Verification of Graph Programs

Christopher M. Poskitt

Submitted for the degree of *Doctor of Philosophy*

The University of York
Department of Computer Science

June 2013

Abstract

This thesis is concerned with verifying the correctness of programs written in GP 2 (for Graph Programs), an experimental, nondeterministic graph manipulation language, in which program states are graphs, and computational steps are applications of graph transformation rules. GP 2 allows for visual programming at a high level of abstraction, with the programmer freed from manipulating low-level data structures and instead solving graph-based problems in a direct, declarative, and rule-based way. To verify that a graph program meets some specification, however, has been – prior to the work described in this thesis – an ad hoc task, detracting from the appeal of using GP 2 to reason about graph algorithms, high-level system specifications, pointer structures, and the many other practical problems in software engineering and programming languages that can be modelled as graph problems. This thesis describes some contributions towards the challenge of verifying graph programs, in particular, Hoare logics with which correctness specifications can be proven in a syntax-directed and compositional manner.

We contribute calculi of proof rules for GP 2 that allow for rigorous reasoning about both partial correctness and termination of graph programs. These are given in an extensional style, i.e. independent of fixed assertion languages. This approach allows for the re-use of proof rules with different assertion languages for graphs, and moreover, allows for properties of the calculi to be inherited: soundness, completeness for termination, and relative completeness (for sufficiently expressive assertion languages).

We propose E-conditions as a graphical, intuitive assertion language for expressing properties of graphs – both about their structure and labelling – generalising the nested conditions of Habel, Pennemann, and Rensink. We instantiate our calculi with this language, explore the relationship between the decidability of the model checking problem and the existence of effective constructions for the extensional assertions, and fix a subclass of graph programs for which we have both. The calculi are then demonstrated by verifying a number of data- and structure-manipulating programs.

We explore the relationship between E-conditions and classical logic, defining translations between the former and a many-sorted predicate logic over graphs; the logic being a potential front end to an implementation of our work in a proof assistant.

Finally, we speculate on several avenues of interesting future work; in particular, a possible extension of E-conditions with transitive closure, for proving specifications involving properties about arbitrary-length paths.

Contents

List of Figures

Definitions and Theorems

Acknowledgements

I remember, shortly prior to commencing my new life as a research student, being somewhat concerned about what I had signed up to do; that I had committed to contributing "something" of my own to computer science. On the one hand this was a rather exciting thing to sign up to do; but on the other, it was quite unnerving and uncertain. I worried about finding myself lost in this journey, or that life as a research student would be lonely.

Happily, on reflection, I can say that neither of these worries came to a head; and that is quite simply because of the people I made this journey with. So many friends, family, and colleagues (some people of which I will undoubtably neglect to mention here) have made these last three(-ish) years amongst my happiest and most fulfilling, and this thesis quite simply would not have been possible without them. I owe them all a great deal of gratitude (and beer/tequila – delete as appropriate).

I am grateful to my supervisor, Detlef Plump, for his invaluable support, guidance, and advice throughout my PhD. His incredible attention to detail, technical understanding, and unending enthusiasm have been of great benefit to me – and this thesis – several times over. Our research discussions were always very lively and enjoyable; already I'm missing them!

I thank my examiners, Barbara König and Jim Woodcock, for their helpful comments on this thesis, and for leading a viva that was both rigorous and enjoyable. Jim, as my internal examiner, regularly provided guidance and encouragement throughout this PhD, also exposing me to a great deal of exciting activities in verification more broadly. For this I am grateful, as well as for the many times we were included in his own activities; the visit to IISc Bangalore being one of the most memorable.

I owe thanks to the Engineering and Physical Sciences Research Council for their financial support throughout my doctoral studies, and to the Department of Computer Science for providing a friendly and stimulating place to undertake them. Very many staff and research students have helped to make this a wonderful place to be. In particular I would to thank my fellow members of PLASMA, the Programming Languages and Systems group (the "MA" has various interesting expansions), for their camaraderie, and for their interest and feedback – especially that of Jeremy Jacob and Colin Runciman – during my (regularly overrunning) seminar slots. Special thanks to my officemates over the years – Chris Bak, Michael Banks, José Calderon, Matthew Naylor, Marco Perez, Jason Reich, Yining Zhao – for making our work environment such a pleasant and supportive place to be, and for forgiving my various oddities (the phase of the giant bouncy ball-seat possibly being amongst the most distracting).

I also had the pleasure of spending three months visiting the group of Tobias Nipkow at Technische Universität München, thanks to the support of a William Gibbs Award. It was fascinating and stimulating to learn about theorem proving and Isabelle, and I hope to use it to make my future work more rigorous; I thank the group for their help and time. While I also extend my thanks to all the members of Lehrstuhl Broy (and later Lehrstuhl Nipkow) for making me feel so welcome in Munich, I would particularly like to thank three friends: Silke Müller, for looking after me and encouraging people to be nice to this language-deficient Brit; Benedikt Hauptmann, for including me in a plethora of social activities and introducing me to the beer connoisseurs of Munich; and Ondřej Kunčar, for the companionship in *that* guesthouse, and the many interesting debates and discussions (which ultimately led me to realise that I don't know the first thing about English).

As well as travelling to Munich I was fortunate enough to travel to a variety of conferences, workshops, meetings – and last but not least – the Marktoberdorf Summer School in 2010. The latter I will never forget: thanks to Katharina Spies (and all the "girls from the office") for your energy and enthusiasm in organising this special event, and thanks also to everyone in the dubiously-named "Team GB" – Luke Herbert, Tuomas Kuismin, James Sharp, Mark Timmer, David Williams, and James Williams (aka Ron) – for all the fun and friendship. Separately, I thank Annegret Habel and Hendrik Radke for hosting a meeting with Detlef and I in Oldenburg, and for the regular feedback on this work whenever we met at ICGT. This gratitude extends also to Karl-Heinz Pennemann, an alumnus of their group, for his comments which helped to improve upon some deficiencies.

I could not have wished for better flatmates during this PhD: David George, Jan Staunton, and James Williams (and there was once a point when Alan Millard was nearly a *de facto* addition to this list!) provided a great deal of fun, support, hugs, and Mario Kart, that I will continue to remember fondly. With James in particular I shared some great times travelling (Amsterdam, Bangalore, Marktoberdorf, to name a few), several far-too-early-in-the-afternoon beers, and many moments of support, all of which I've missed very much since leaving York.

Beyond my flatmates I have a truly wonderful network of friends who enriched my life in many different ways: I daren't try to name them all because no doubt I would be incomplete, but I hope they all know who they are, that they are very special to me, and that I miss them a lot already.

I thank my family for their unwavering love, encouragement, and support throughout these six(-ish) years as a student. I really am incredibly lucky to have you all, and I hope I can get to spend more time with you now that this thesis is done (he says, after moving to Switzerland!).

Finally, last, but of course not least, my deepest gratitude goes to Immy for the love, support, and patience, that got me through to the end.

Author's Declaration

I declare that the work in this thesis is my own, except where otherwise attributed or cited to another author. Some parts of this thesis have been previously published in journal, conference, and workshop papers; the details of which are given in Section 1.4.

C.M. Poskitt, June 2013

Chapter 1

Introduction

We open this thesis by motivating our research problem and summarising our contributions towards addressing it. Then, we describe its structure, giving an overview of the chapters and appendices, and discuss our conventions and assumptions of the reader. Finally, we disclose the peer-reviewed publications in which some of this work has been previously disseminated.

1.1 Motivation

Graphs are ubiquitous in computer science because they model many types of structures and relations in a simple and intuitive way, from pointers in C, to network topologies and high-level system specifications. Many instances of practical problems can be modelled as graph problems, motivating the development and study of algorithms that operate on graphs. One widely studied approach for manipulating graphs – at that same high level of abstraction – is *graph transformation* [EEPT06], in which graphs represent state and *rules* represent computational steps. These are akin to Chomsky string rewriting rules, but lifted to the setting of graphs. Applications of graph transformation to programming languages and software engineering include the semantics and implementation of functional programming languages [Pey87, PvE93], the specification and analysis of pointer structures [BPR04b, BPR04a, RN08], the semantics of the Unified Modelling Language [HZG06, KGKZ09], and the semantics and analysis of model transformations [VVP02, GGZ+05, BET08, HEOG10, KKvT12, GL12].

Applications to the semantics of languages and the analysis of systems naturally raise the question of how to formally verify properties of graph transformation systems. Several verification approaches have emerged in recent years, typically focusing on sets of graph transformation rules or graph grammars; techniques including inductively inferring correctness from a logical characterisation of rules [dCR12], model checking [SV03,

GdMR+12], and abstraction-based approaches for approximating infinite-state spaces [BCK08, BBKR08, RZ12] (these techniques, among others, we review in Section 3.7.2).

Graph transformation languages such as PROGRES [SWZ99], Fujaba [NNZ00], AGG [Tae04], and GrGen [GBG+06], however, provide control constructs on top of graph transformation rules for practical problem solving. To give a simple example, consider the problem of reversing the direction of the edges of an input graph. This requires two loops in sequence: the first applies for as long as possible a rule which reverses an edge and marks it as reversed (to ensure termination), the second applies for as long as possible a rule which removes the auxiliary edge mark.

A first step beyond the verification of graph grammars has been made by Habel, Pennemann, and Rensink [HPR06, HP09], who contributed a weakest precondition verification framework for a graph programming language, with constructs including sequential composition and as-long-as-possible iteration over sets of rules. The authors adopt Dijkstra's approach to program verification: one calculates the weakest precondition for a program and its postcondition, and then needs to prove that the program's precondition implies the weakest precondition. Their language falls short of practical graph transformation languages, however, in that it lacks the ability to compute with labels (or attributes), a capability which is indispensable for many graph algorithms. For example, computing the shortest path between two nodes requires one to compare and add distances (edge labels).

In this thesis we present an approach for verifying programs in the graph programming language GP 2 [Plu12], a nondeterministic language for high-level problem solving in the domain of graphs. GP 2 is based on graph transformation rules and has a simple syntax and semantics, designed to facilitate reasoning about programs. The core of GP 2 consists of just five constructs: single-step application of a set of rules, sequential composition, looping, and two conditional branching constructs. The graph transformation rules in GP 2 are labelled over (sequences of) expressions, and are interpreted over a label alphabet comprising (sequences of) integers and strings. Moreover, they can be equipped with simple Boolean conditions to forbid the existence of certain edges, and require certain relations to hold between the labels.

Instead of adopting the weakest precondition approach to verification, we follow Hoare's seminal paper [Hoa69] and devise calculi of syntax-directed proof rules for three notions of correctness (i.e. three levels of guarantees about termination). Our proof system aims at human-guided verification and the compositional construction of proofs, with future assistance from a mechanical theorem prover. This is in line with work on program verification for languages such as Java [PHM99, HJ00, vO01, Nip02] and Eiffel [NCMM09].

1.2 Research Hypothesis and Contributions

The hypothesis of this thesis is the following:

> *The graph transformation language GP 2 can effectively be equipped with a reasoning system based on Hoare logic, facilitating proofs about both structural properties of graphs and relations between their labels.*

To support this hypothesis, such a reasoning system must satisfy some criteria. In particular, it should be:

- *sound*: every specification one can prove in the system must be valid with respect to the semantics of the language;
- *realistic*: in that it does not require impractical assumptions or restrictions on programs;
- *practical*: it can effectively reason about interesting partial correctness and termination properties of structure- and data-manipulating graph programs;
- *general*: it should not be fixed to particular assertion languages, and hence should be easy to extend.

These criteria will provide the basis with which we evaluate the work to be presented in the following chapters. To this end, the major contributions of this thesis are:

Hoare logic for graph programs. We present Hoare calculi for reasoning about partial correctness and termination of a programming language based on graph transformation. To the best of our knowledge, we are the first to study Hoare logic in this context, having reported an earlier version of this work in [PP10a].

Language-independent approach. We follow an extensional approach and define our calculi independently of a fixed assertion language. Graph specification languages of any kind – whether logical ones, graph grammars, or even graph programs themselves – can be "plugged in" to our axioms and inference rules, subject to satisfying certain requirements, e.g. being able to define a weakest liberal precondition transformation. We prove the calculi to be sound, relatively complete (for sufficiently expressive assertion languages), and complete for termination.

An assertion language for graph programs. We extend the morphism-based nested conditions of Habel, Pennemann, and Rensink to E-conditions, a graphical and intuitive assertion language able to express properties about

labels (drawn from an infinite label alphabet) in addition to properties about structure. We instantiate our extensional calculi with E-conditions by defining transformations for applicability of rules and their weakest liberal preconditions. We prove these transformations correct and thus inherit soundness and completeness for termination from the extensional calculi (relative completeness for E-conditions remains open). We demonstrate the calculi by verifying a number of data- and structure-manipulating programs that up until now have only been verified in an ad-hoc fashion. We also begin to explore the use of an extension of the language with transitive closure, i.e. the ability to express properties about arbitrary length paths.

A logic for proof assistants. While we benefit from the high level of abstraction in E-conditions – e.g. in defining weakest liberal preconditions – the formalism would be complicated to encode directly in off-the-shelf proof assistants. As a first step towards a future implementation of our verification calculi in such tools, we have defined translations between our morphism-based formalism and a classical many-sorted logic for graphs. We prove the translations to be correct, and show how the logic itself can be "plugged in" to our extensional calculi to provide another way of reasoning about graph programs.

1.3 Thesis Structure

The content of this thesis is divided into several chapters, as follows.

- *Chapter 2* introduces several technical preliminaries for graph transformation, and the frameworks in which graph transformation rules are applied. Then, we review the graph programming language GP 2, including its operational semantics, and discuss some example programs.
- *Chapter 3* opens with a review on how to reason with Hoare logic. Then, it introduces Hoare calculi for graph programs that are extensional (i.e. not tied to a particular assertion language), and that can prove specifications to three levels of correctness. Properties of the calculi are studied including soundness, relative completeness, and completeness for termination. The calculi use extensional assertions expressing weakest liberal preconditions as well as successful and failing program executions. The relationship between the existence of effective constructions for such assertions and the decidability of the model checking problem is explored.
- *Chapter 4* instantiates the extensional calculi of Chapter 3 with E-conditions, a morphism-based assertion language for expressing properties about graph labels and structure. Transformations defining the

extensional assertions (for a subclass of programs) are introduced and proven to be correct, with the calculi then inheriting soundness and completeness for termination. The question of whether the calculi with E-conditions are relatively complete remains open and the problem is discussed, before our approach is compared with the closely related work of Oldenburg.

- *Chapter 5* demonstrates the use of our calculi with E-conditions by proving various specifications about data-manipulating graph programs. Then, it goes beyond what E-conditions can be used to prove, and speculates on constructing proofs using an extension of the assertion language with transitive closure.
- *Chapter 6* provides a first step towards an implementation of our work in a proof assistant, by defining translations between E-constraints and a new, many-sorted logic over graphs. Then, it shows how the logic can be "plugged in" to our extensional calculi by taking advantage of these translations.
- *Chapter 7* brings the main part of the thesis to a close, drawing some conclusions, and discussing several potential items of future work.
- *Appendix A* lists the axioms and inference rules of our extensional Hoare calculi (for reference).
- *Appendix B* lists the axioms and inference rules of the Hoare calculi with E-conditions (for reference).
- *Appendix C* lists some definitions and properties of pushouts and pullbacks – used in the algebraic approach to graph transformation – in order to support some proofs in Chapter 4.
- *Appendix D* contains the proofs of some lemmata relied upon in Chapter 6.

Most chapters end with related work and summary sections. The former can be thought of as chapter notes – or bibliographic remarks – in which related and historical publications are discussed and cited. Such discussions, whilst important, are grouped into their own sections to avoid breaking the narrative flow in the main sections of the chapters. The summary sections then highlight the main points to take away from each chapter.

We assume the reader to have a basic knowledge of theoretical computer science, in particular, the basic notations and constructions from discrete mathematics and logic. We also assume a little familiarity with terminology from graph theory (a classic reference is [Har69]), e.g. adjacent nodes, incident edges, loops, connectedness.

1.4 Publication History

The results of this thesis have been disseminated in the journal, conference, and workshop papers listed below (we have indicated which parts of the thesis each paper relates to). The author of this thesis was the primary contributor in all cases.

1. **[PP13]** C.M. Poskitt and D. Plump. Verifying Total Correctness of Graph Programs. In *Selected Revised Papers, Graph Computation Models (GCM 2012)*. Electronic Communications of the EASST, volume 61, 2013.

    - Introduced (weak) total correctness calculi with E-conditions as the assertions (i.e. parts of Chapters 3 and 4), but was also the first of our papers to use GP 2 [Plu12], as opposed to GP 1 [Plu09].

2. **[PP12]** C.M. Poskitt and D. Plump. Hoare-Style Verification of Graph Programs. *Fundamenta Informaticae*, 118(1-2):135-175. IOS Press, 2012.

    - Extended [PP10a] by adding proofs and further examples. Much of Chapter 4 is based on this, but was updated for GP 2, and the proofs (we hope) improved.

3. **[Pos12]** C.M. Poskitt. Verification of Graph Programs. In *Proc. International Conference on Graph Transformation (ICGT 2012)*, volume 7562 of *Lecture Notes in Computer Science*, pages 420-422. Springer-Verlag, 2012.

    - An extended abstract providing an overview of this thesis, especially Chapters 4, 5, and 6.

4. **[PP10a]** C.M. Poskitt and D. Plump. A Hoare Calculus for Graph Programs. In *Proc. International Conference on Graph Transformation (ICGT 2010)*, volume 6372 of *Lecture Notes in Computer Science*, pages 139-154. Springer-Verlag, 2010.

    - Introduced an earlier version of the partial correctness calculus with E-conditions in Chapter 4, including a soundness proof.

5. **[PP10b]** C.M. Poskitt and D. Plump. Hoare Logic for Graph Programs. In *Proc. Theory Workshop at the Third International Conference on Verified Software: Theories, Tools, and Experiments (VS-THEORY 2010)*. 2010.

- Essentially an overview of the work in [PP10a], tailored to a more general audience (i.e. several technical details were described informally).

Chapter 2

Computing by Graph Transformation

This chapter has two primary aims: first, to review and convey an intuition for the fundamentals of graph transformation, providing the necessary foundations for the later chapters of this thesis. Secondly, we review the graph programming language GP 2 that we are setting out to verify, discussing both its syntax and semantics, and illustrating its use with some example programs.

2.1 Fundamentals of Graph Transformation

Fundamentally, in graph transformation we want to be able to write *rules* of the form $L \rightsquigarrow R$, where L is some subgraph to be matched and R is some graph with which to replace the match. One can view this as a generalisation of rules in Chomsky string grammars, but in graph transformation, rule application is not quite so simple. What exactly is a match for L in some graph G? What if 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 under transformation?

These questions have different answers depending on the graph transformation framework applied. In this thesis, rule application is based on the *double-pushout (DPO) approach*. The name comes from its algebraic characterisation, but an important and desirable property of this approach is that rule applications are side-effect free (in particular, rule applications that would leave edges "dangling" are simply forbidden).

We begin by defining graphs and structure-preserving mappings – morphisms – that we frequently use to relate them. Then, we describe the DPO approach to rewriting graphs we use in this thesis, giving both the concrete steps as well as an algebraic characterisation.

We present here only the essentials for this thesis. For a broader technical introduction to graph transformation, we refer the reader to the monograph [EEPT06].

2.1.1 Graphs and Graph Morphisms

We use a definition of graphs in which edges are directed, nodes (resp. edges) are partially (resp. totally) labelled, and in which parallel edges are allowed to exist. Usually, we use graphs that are totally labelled. If in some problem domain a label plays no role, then by convention we label everything with the blank label $\square$ which is not usually drawn. Unlabelled nodes are however required later for technical reasons, to define the application of rules that relabel nodes (this will be explained in Section 2.1.3).

Definition 2.1 (Label alphabet)**.** A *label alphabet* $\mathcal{C} = \langle \mathcal{C}_V, \mathcal{C}_E \rangle$ is a pair comprising a set $\mathcal{C}_V$ of *node labels* and a set $\mathcal{C}_E$ of *edge labels*. $\square$

Definition 2.2 (Graph)**.** A *graph* over a label alphabet $\mathcal{C}$ 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* $l_G \colon V_G \to \mathcal{C}_V$, and a total *edge labelling function* $m_G \colon E_G \to \mathcal{C}_E$. 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. A graph G is *totally labelled* if l_G is a total function. $\square$

It may feel more familiar to see graphs defined as ordered pairs $G = \langle V, E \rangle$ with $E \subseteq V \times V$. Our definition is more general, however, in that it allows for parallel edges and labelling.

Example 2.3 (A graph)**.** 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 in that we abstract from node and edge identities. 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 $\square$.

$\square$

Definition 2.4 (Classes of graphs)**.** We write $\mathcal{G}(\mathcal{C}_\perp)$ (resp. $\mathcal{G}(\mathcal{C})$) to denote the *class* of all (resp. all *totally labelled*) graphs over label alphabet $\mathcal{C}$. $\square$

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, and are important for understanding how rules are applied as well as for how graphs are reasoned about (see e.g. Chapter 4).

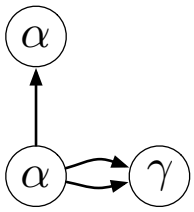


Figure 2.1: A graph

Definition 2.5 (Graph morphism)**.** Given graphs G, H over a label alphabet $\mathcal{C}$, 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 sources, targets, and labels. 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 $l_G(v) \neq \bot$.

Here, $\circ$ 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, surjective, and satisfies $l_H(f_V(v)) = \bot$ for all nodes v with $l_G(v) = \bot$; 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 2.6 (Graph morphisms)**.** 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).

□

Definition 2.7 (Domain and codomain of graph morphisms)**.** Let $f : G \to H$ denote a graph morphism. We call G and H respectively the *domain* and *codomain of* f. □

2.1.2 Rules, Matches, and Direct Derivations

The underlying framework used in this thesis for the application of graph transformation rules is the *double-pushout (DPO) approach*, first described by Ehrig, Pfender, and Schneider in [EPS73], and more recently treated in e.g. [CMR+97, HMP01, EEPT06]. The technical name of this approach comes from its algebraic characterisation with concepts from category theory. Intuitively however, the DPO approach guarantees that rule applications are

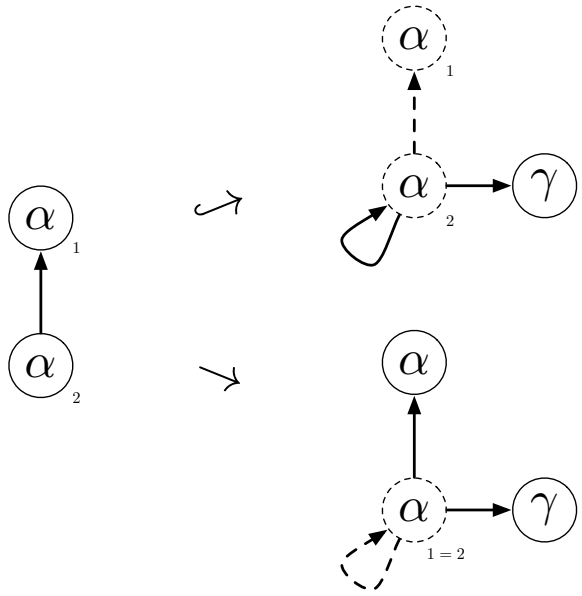


Figure 2.2: An injective and a non-injective graph morphism

local, in the sense that all of the changes made to the graph are described by the rules themselves. This, for example, means that rules can only be applied for matches if the result of the application would not leave edges dangling.

We introduce the DPO approach first by formally defining rules and matches. Then, we consider the derivation of a graph from another via the application of a rule, presenting it from two viewpoints: firstly, as a concrete construction; and secondly, as an algebraic one.

Rules and Rule Applications

In the DPO approach a rule $L \rightsquigarrow R$ is actually described by three components: L, R, and additionally an interface K.

Definition 2.8 (Rule)**.** A *rule* $r : \langle L \hookleftarrow K \hookrightarrow R \rangle$ over a label alphabet $\mathcal{C}$ comprises totally labelled graphs $L, K, R \in \mathcal{G}(\mathcal{C})$, 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. □

In rules, the graph L describes what is to be matched, and R describes what the match should be replaced with. The interface graph K describes a part of the graph that is required to exist for the rule to be applicable, but is not actually changed by the application of the rule. The part of the graph to be deleted is described by $L - K$, and $R - K$ describes the part of the graph to be "glued on".

We will often give rules without the interface, simply writing $r : \langle L \Rightarrow R \rangle$. In such cases we number nodes that correspond in L and R, i.e. they are not created or deleted, but must be matched and must be preserved. We establish the convention that K comprises exactly these nodes, and that $E_K = \emptyset$; hence L and R contain all the information necessary with which to infer K.

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*, intuitively meaning that if r deletes a node in the match, then all edges incident to it are also in the match and would be deleted by r.

Definition 2.9 (Dangling condition; match)**.** Given a rule $r : \langle L \hookleftarrow K \hookrightarrow R \rangle$, a totally labelled 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 $g(L - K)$. We say that $g : L \hookrightarrow G$ is a *match* for r if it satisfies the dangling condition. □

The application of a rule to a graph roughly involves deleting everything in the match that is not preserved by the interface, then adding everything in R that is not also in the interface.

Definition 2.10 (Rule application)**.** Given a rule $r = \langle L \hookleftarrow K \hookrightarrow R \rangle$ and a match $g : L \hookrightarrow G$, we write $G \Rightarrow_{r,g} M$ if $H \cong M$, where H is constructed from G as follows:

1. Remove all nodes and edges in $g(L) - g(K)$, obtaining a graph D.
2. Add disjointly to D all nodes and edges from $R - K$ retaining their labels, obtaining a graph H. For $e \in E_R - E_K$, $s_H(e) = s_R(e)$ if $s_R(e) \in V_R - V_K$, otherwise $s_H(e) = g_V(s_R(e))$.
3. Target functions are defined analogously.

□

Example 2.11 (Rule application)**.** Figure 2.3 shows an example of a rule application. The top three graphs are respectively L, K, and R. The bottom three graphs are respectively G, D, and H. The rule matches a pair of adjacent α-labelled nodes (the edge labelled with the blank symbol □), removes one of the nodes and the edge, and replaces them with a looping edge. The part of the graph being rewritten is indicated by dotted lines.

□

Pushouts and Direct Derivations

Our definition of rule application is expressed in operational steps, and in terms of the individual nodes and edges mapped to by a match. Such a

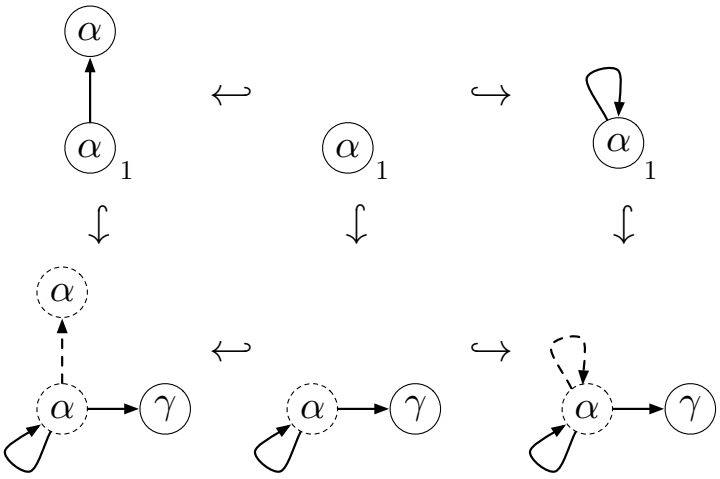


Figure 2.3: A rule application

definition is of course important for practical reasons, but is harder for reasoning about at an abstract level. Rule applications in the DPO approach have an abstract, algebraic characterisation – one that gives the approach its name – in which graphs are treated as algebras, and in which rule applications are defined by an algebraic construction modelled as two pushouts [CMR+97]. As well as providing an abstract framework to allow for formal reasoning about rule applications, this approach allows for the use of general results and properties from the branch of mathematics known as category theory.

We give in this subsection a brief introduction to the essentials of the algebraic approach that are needed to understand the technical parts of this thesis. First, we review pushouts, the gluing construction for graphs in the DPO approach. Then, we review a construction comprising two pushouts for directly deriving a graph via a rule and a match. Further technical fundamentals from the DPO approach (e.g. properties, decompositions) are given together in Appendix C, and are referred to in some proofs in later chapters.

Pushouts are used in this framework as a gluing construction for two morphisms with a common domain [EEPT06]. Informally, a pushout graph is formed from the disjoint union of the two codomains, but with nodes and edges also present in the common domain identified.

Definition 2.12 (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 2.4 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 $C \to D \to D' = C \to D'$.

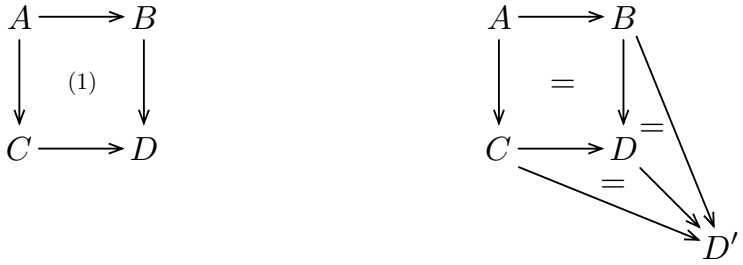


Figure 2.4: A pushout and the universal property of pushouts

□

Pushouts have a number of important properties. First, every item in D has a so-called 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). Another property of interest is the uniqueness of D (up to isomorphism), 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 $A \to C$ if and only if 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 2.13 (Pushout)**.** Figure 2.5 depicts a simple pushout, in which all the morphisms are injective. All nodes and edges are labelled with the blank symbol (the numbers indicate the mappings of the morphisms).

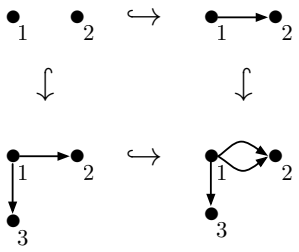


Figure 2.5: A pushout

□

If we have a pair of graph morphisms $A \to B$ and $B \to D$, a pushout complement is the graph and two morphisms needed to form a pushout as in (1) of Figure 2.4.

Definition 2.14 (Pushout complement)**.** Given graph morphisms $A \to B$ and $B \to D$, a *pushout complement* of these morphisms is a graph C together with two morphisms $A \to C$ and $C \to D$ if the resulting diagram (as in (1) of Figure 2.4) is a pushout. □

Rule applications in the DPO approach can be considered as the construction of a pushout complement followed by a pushout, as in the two squares of Figure 2.3. Given a graph G and rule r with match g, we say that H is directly derived from G via r and g if a double-pushout diagram exists as described in the following.

Definition 2.15 (Direct derivation; comatch)**.** Given a rule $r : \langle L \hookleftarrow K \hookrightarrow R \rangle$, graph G, and a match $g : L \hookrightarrow G$, a *direct derivation* $G \Rightarrow_{r,g} H$ from graph G to graph H is given by the double-pushout diagram in Figure 2.6 with pushouts (1) and (2). The morphism $R \hookrightarrow H$ is called the *comatch* of r. □

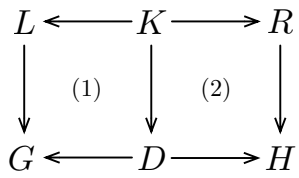


Figure 2.6: A direct derivation

The pushouts (1) and (2) only exist if g satisfies the dangling condition, i.e. is a match. One can show that D and H are determined uniquely up to isomorphism, i.e. direct derivations $G \Rightarrow_{r,g} H$ are unique up to isomorphism. (Note that H is the same graph, up to isomorphism, as the one constructed in Definition 2.10.)

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 $G \Rightarrow_{\mathcal{R}} H$ if $G \Rightarrow_r H$ for some $r \in \mathcal{R}$.

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.

2.1.3 Rules with Relabelling

The traditional DPO approach is not ideal for relabelling nodes, due to the requirement that L, K, and R are totally labelled graphs. Whilst an edge can easily be "relabelled" in an arbitrary context – by deleting and recreating the edge with a new label – the same is not true for nodes, because the

requirement to satisfy the dangling condition prevents a node from being deleted unless 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, whilst preserving the uniqueness (up to isomorphism) 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 for graph programs by insisting that L and R are totally labelled, but allowing the nodes of K to be partially labelled. We introduce this tailored approach which later provides the semantic foundation of rule applications in graph programs. Further technical details are given in Appendix C.

Definition 2.16 (Rule with relabelling)**.** A *rule (with relabelling)* $r : \langle L \hookleftarrow K \hookrightarrow R \rangle$ over label alphabet $\mathcal{C}$ comprises totally labelled graphs $L, R \in \mathcal{G}(\mathcal{C})$, a partially labelled graph $K \in \mathcal{G}(\mathcal{C}_\perp)$, and inclusions $K \hookrightarrow L$, $K \hookrightarrow R$. (Terminology and abbreviations are the same as in Definition 2.8.) □

Note that in this definition, unlabelled nodes in K can have different labels in L and R.

Again, we will frequently omit the interface in rules. We establish the convention that K comprises exactly the numbered nodes in L and R (the numbers indicating that nodes correspond, as before), that $E_K = \emptyset$, and additionally that all nodes in K are unlabelled.

A rule application with relabelling proceeds exactly as in Definition 2.10 but with additional treatment for unlabelled nodes in the interface.

Definition 2.17 (Rule application with relabelling)**.** Given a rule $r = \langle L \hookleftarrow K \hookrightarrow R \rangle$ and a match $g \colon L \hookrightarrow G$, we write $G \Rightarrow_{r,g} M$ if $H \cong M$, where H is the graph that is constructed from G as follows:

1. Remove all nodes and edges in $g(L) - g(K)$. Then, nodes that are images of the unlabelled nodes in K are made unlabelled themselves, obtaining a graph D.

2. Add disjointly to D all nodes and edges from $R - K$ retaining their labels, obtaining a graph H. For $e \in E_R - E_K$, $s_H(e) = s_R(e)$ if $s_R(e) \in V_R - V_K$, otherwise $s_H(e) = g_V(s_R(e))$.

3. Target functions are defined analogously.

4. For each unlabelled node v in K, $l_H(g_V(v))$ becomes $l_R(v)$.

□

Example 2.18 (Rule application with relabelling)**.** An example rule and a possible application of it are given together in Figure 2.7, where both nodes

and edges are labelled over the alphabet $\{$  ,  , $\square\}$. The top row displays the left- and right-hand sides of the rule, with the interface in the middle (note that the nodes are unlabelled). The bottom left graph is the one to which the rule is being applied, the bottom right graph is the outcome of applying the rule with the particular match g.

□

We give rule applications with relabelling an algebraic characterisation using the framework of [HP02], which is a conservative extension of the traditional DPO approach. The key requirement is that the two pushout squares are also *pullbacks* (the dual of pushouts). Squares that are both pushouts and pullbacks are called *natural pushouts*.

Definition 2.19 (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 2.8 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 2.20 (Natural pushout)**.** A diagram (1) as in Figure 2.8 is a *natural pushout* if it is both a pushout and a pullback. □

Habel and Plump [HP02] give a characterisation of natural pushouts. Consider the pushout diagram (1) of Figure 2.4. The diagram 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 also unlabelled.

Direct derivations for rules with unlabelled interface nodes are defined as in the traditional DPO approach, but with the requirement just discussed that pushouts are natural.

Definition 2.21 (Direct derivation with relabelling)**.** Given a rule $r : \langle L \hookleftarrow K \hookrightarrow R \rangle$ with L, R (resp. K) totally (resp. partially) labelled, and a match $g : L \hookrightarrow G$, a *direct derivation (with relabelling)* $G \Rightarrow_{r,g} H$ from graph G to graph H is given in Figure 2.9, where (1) and (2) are natural pushouts.

□

The graph H is unique up to isomorphism, and is totally labelled if G is totally labelled [HP02]. (We remark that the graphs we later define graph programs to operate on are totally labelled.) Moreover, graph H is also the same as the H constructed in Definition 2.17.

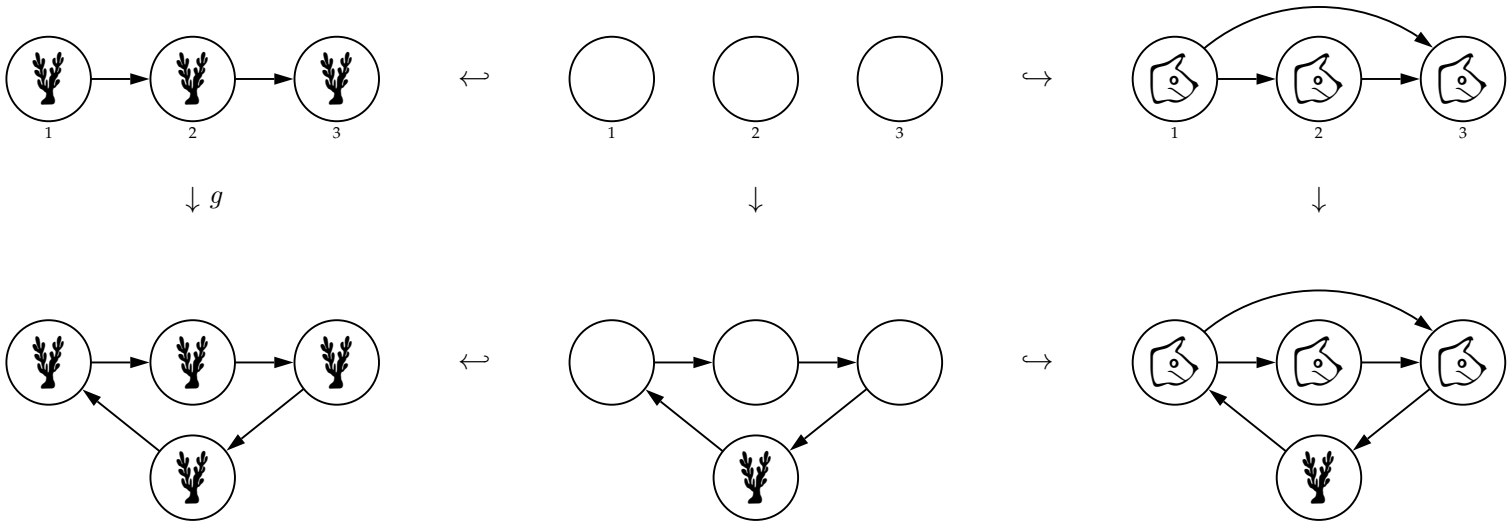


Figure 2.7: A rule application with relabelling

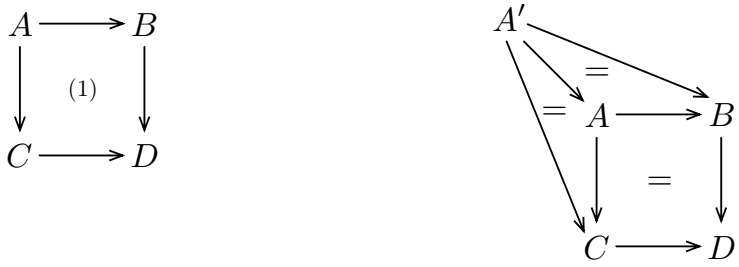


Figure 2.8: A pullback and the universal property of pullbacks

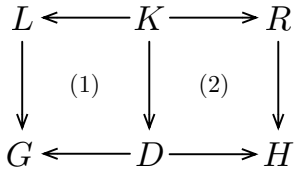


Figure 2.9: A direct derivation with relabelling

Example 2.22. Consider the pushout diagrams in Figure 2.10. The diagram on the left comprises natural pushouts, whereas the diagram on the right comprises non-natural pushouts (example from [Plu09]).

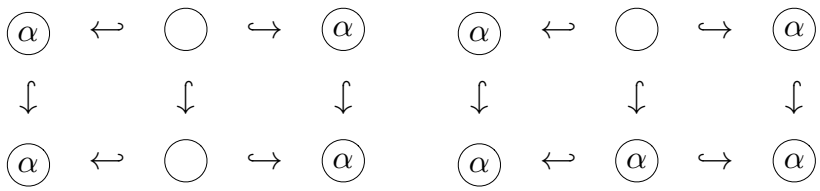


Figure 2.10: Natural (left) and non-natural (right) double-pushout diagrams

□

2.2 Graph Programs

Thus far this chapter has focused on the basic definitions, mechanisms, and frameworks by which a rule application can rewrite a graph – whether with node relabelling or not. Analogously to Chomsky string grammars for defining languages or classes of strings, we can create graph grammars for defining classes of graphs, with a wide variety of applications (see e.g. the handbook [HER99a]). Indeed, much of the research into verifying graph transformations has focused primarily on the verification of such grammars (see Section 3.7.2). In this thesis however, we focus on a more general model of computation than graph grammars, in that instead

of applying sets of rules an arbitrary number of times to some fixed starting graph, we allow users to write programs expressing how those rules should be applied, and moreover, we allow these programs to be executed on graphs provided by users. That is, we focus on a graph transformation programming language allowing for graph-like problems to be solved at a high level of abstraction, and thus in a more natural way than if one were required to work directly with the low level data structures that traditional languages typically use to implement graphs.

The graph transformation language we introduce in this section, and work with for the rest of the thesis, is GP 2 [Plu12] – the initialism standing for Graph Programs. A graph program comprises two components: (1) a set of rules, based upon (but more general than) those in the DPO approach with relabelling; and (2) some program text expressing how the rules are to be applied to an input graph. The former, more precisely, are called *rule schemata*, which are labelled over syntactic expressions and describe sets of rules with concrete labels. The latter is built from a small core of control constructs: single-step application of a set of rule schemata, sequential composition, conditional branching, and iterative looping. The language is equipped with an operational semantics, intended by the designers to facilitate formal reasoning and verification – and later used in this thesis for proving the soundness of our proof calculi.

Remark 2.23 (GP and GP 2)**.** Most of our previously published work focused on the version of GP as described in [Plu09]. This thesis however uses the revised syntax and semantics of GP 2 [Plu12], and thus previously published work has been updated to be compatible (we give a short summary of the changes and new features of GP 2 in Section 2.3 for the interested reader). For brevity we will typically write "graph programs" or "GP", as opposed to "GP 2". Unless we explicitly make reference to the older versions of the language, it can be understood that every instance of GP in this thesis refers to the language as described in [Plu12]. □

The rest of this section is organised as follows. First, we define the building blocks of GP – (conditional) rule schemata – and the class of graphs that they operate on. Then, we give the abstract syntax of program text, and discuss some examples of graph programs. Finally, we give the operational semantics of both core and derived commands in GP.

Note that the definitions presented in the rest of this section are based on those given in [Plu12].

2.2.1 Graphs and Conditional Rule Schemata

In GP, the graphs that programs are executed on are labelled differently to the graphs that make up the rule schemata. The former are labelled with (lists of) integers and strings, and the latter with (lists of) expressions. The

former can be thought of as "semantic" graphs, in that they are the graphs provided as input to programs. The latter can by thought of as "syntactic" graphs, representing possibly infinite sets of rules at the semantic level. We fix the two label alphabets in what follows: both are designed to allow general-purpose computation on graphs, e.g. lists to allow information to be appended to existing labels, and integers to allow counting and arithmetic. Both label alphabets also encode the ability to "mark" nodes and edges.

Labels in semantic graphs comprise a list component and a mark component. The list component of a label comprises a (possibly empty) list of *atoms*, each atom an integer or string. The mark component is a Boolean value (true for marked, false for unmarked): a marked node is displayed as shaded whereas a marked edge is dashed. Marks were added to GP 2 to facilitate more natural implementations of graph algorithms that need, for example, a mechanism for indicating that a node or edge has been visited.

In both syntactic and semantic graphs, we will use the same label alphabet for both nodes and edges.

Definition 2.24 (Label alphabet $\mathcal{L}$)**.** Let $\mathbb{Z}$ be the set of integers, Char be a finite set of characters and $\mathbb{L} = (\mathbb{Z} \cup \text{Char}^*)^*$ the set of all lists of integers and character strings. We denote with $\mathcal{L}$ the label alphabet for *semantic graphs*:

$$\mathcal{L} = \mathbb{L} \times \mathbb{B}$$

where $\mathbb{B} = \{\text{true}, \text{false}\}$. □

For clarity, we use a colon ':' to delimit the atoms in a list. For example, the list comprising the integer 25 followed by the strings "York" and "Zürich" would be displayed as 25:"York":"Zürich".

Remark 2.25 (Marks and labelling functions)**.** Though labels in $\mathcal{L}$ comprise both a list component and a mark component, in several parts of this thesis we will only be concerned with the list component. Usually we will be explicit, for example by using the symbol $\mathbb{L}$ (denoting the set of all lists) as opposed to $\mathcal{L}$ (the set of all labels). For simplicity however, we will occasionally abuse our notations by omitting the mark component. For example, in writing $l_G(v) = 25$, we mean that node v in graph G has list component 25 with some unspecified mark component. □

Labels in rule schemata again comprise a list and a mark component. The list components however are now lists of *expressions*. Each expression is built up from constant symbols (representing semantic values in $\mathbb{Z}$ and Char^*), variables, and function symbols. Variables are typed, each from a disjoint set of variable identifiers: IVar, SVar, AVar, and LVar respectively for integer, string, atomic, and list expressions. The abstract syntax of these expressions is defined by a grammar. (The grammar is ambiguous: we use parentheses to disambiguate expressions.)

Definition 2.26 (Graph labels for rule schemata)**.** Let RS (short for Rule Schemata) denote the label alphabet containing all expressions that can be derived from the syntactic class RSLabel of Figure 2.11. Here, Digit denotes the set $\{0, 1, \ldots, 9\}$, and IVar, SVar, AVar, and LVar are respectively sets of variables of type Integer, String, Atom, and List. By $\mathcal{G}(\mathrm{RS})$ we denote the class of all graphs labelled over RS.

Integer	::=	Digit {Digit} | IVar | '−' Integer
		| Integer ArithOp Integer
ArithOp	::=	'+' | '−' | '⋆' | '/'
String	::=	'"' {Char} '"' | SVar | String '.' String
Atom	::=	Integer | String | AVar
List	::=	`empty` | Atom | LVar | List ':' List
RSLabel	::=	List Mark
Mark	::=	`true` | `false`

Figure 2.11: Abstract syntax of rule schema labels

□

Expressions represent possibly infinite sets of lists in $\mathbb{L}$, which are obtained by assigning values to variables (respecting their types) and evaluating expressions. We identify lists of length one with the atomic expressions they contain, allowing us to divide semantic and syntactic lists into four types and establish a hierarchical connection between them. This *subtype hierarchy* is shown in Figure 2.12. (We use the non-terminals of the grammar to denote the syntactic classes of expressions that can be derived from them.)

Figure 2.12: Subtype hierarchy for lists

The intended meaning of most symbols should be clear, e.g. + denotes addition of integers, and . denotes string concatenation. A precise semantics will be given with the semantics of rule schema application.

Example 2.27 (List expressions)**.** Examples of expressions include `25 + x`, `"Yo"."rk"`, and `y : 0 : "Zuer".z`. These expressions respectively belong to Integer, String, and List (the first two also belong to both Atom and List because of the hierarchical structure of types), provided that `x` $\in$ IVar, and `z` $\in$ SVar (the variable `y` can be of any type). □

Remark 2.28 (Indegree and outdegree expressions)**.** The grammar of Figure 2.11 is a slightly restricted version of the one in [Plu12], which additionally allows integer expressions about the indegrees and outdegrees of nodes. These integer expressions were introduced only in GP 2, allowing for labels such as `outdeg(1)`, which would be interpreted as the outdegree of node 1 in the graph that the expression is used. These differ from all other list expressions in that their interpretations rely on information about the structure of a graph; all other expressions are interpreted only with respect to an assignment of variables. We omit expressions about indegrees and outdegrees to preserve the separation of expressions and graph structure that was present in GP 1 [Plu09], which was a design choice mirrored in our assertion language for verification (introduced in [PP10a, PP12]). To revise the assertion language to support in- and outdegree expressions would require fundamental changes to its definition, which would likely break many of the constructions and proofs that assume a separation of expressions and graph structure. Hence, we leave such an investigation as future work. □

Rule schemata are rules as defined in Definition 2.16, but with the left- and right-hand side graphs from $\mathcal{G}(\mathrm{RS})$ and the interface graph consisting of unlabelled nodes only. They represent (usually) infinite sets of rules with graphs in $\mathcal{G}(\mathcal{L})$, obtained by assigning variables to values and evaluating expressions. Because this is done during graph matching, we require that the expressions in the left-hand graph have a simple shape.

Definition 2.29 (Simple list)**.** A list expression $l \in$ List is *simple* if:

1. l contains no arithmetic operators;
2. l contains at most one occurrence of a list variable; and
3. each occurrence of a string expression in l contains at most one occurrence of a string variable.

□

Definition 2.30 (Rule schema)**.** A *rule schema* $r : \langle L \hookleftarrow K \hookrightarrow R \rangle$ comprises two totally labelled graphs $L, R \in \mathcal{G}(\mathrm{RS})$, a graph K containing only unlabelled nodes, and inclusions $K \hookrightarrow L$, $K \hookrightarrow R$. We require that all list expressions in L are simple and that all variables in R also occur in L. □

Rather than drawing the interface K explicitly in rule schemata, we leave it implicit by writing node identifiers in L and R to indicate which nodes are the same. Nodes without identifiers in L are to be deleted, and those without identifiers in R are to be created. Hence, K comprises those nodes with identifiers (but with their labels removed).

Example 2.31 (Rule schema)**.** An example of a rule schema is `propagate`, given in Figure 2.13. A possible match would be a pair of adjacent nodes such that: (1) the source (resp. target) node is marked (resp. unmarked); (2) the edge is unmarked; and (3) the list components of the edge and source node are integers. In an execution of `propagate`, the edge and target node would become marked, and the list component of the target node would be replaced by the sum of integers `i` and `j`. Note that this rule simply performs a relabelling: both nodes on both sides are given with identifiers (1 and 2), hence no node is deleted and no node is created.

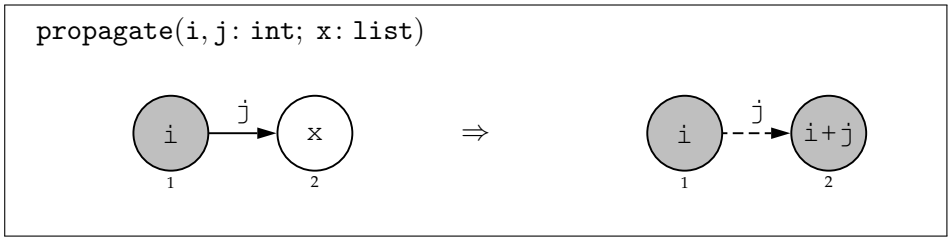


Figure 2.13: A rule schema

If given, the interface graph K for `propagate` would be drawn as in Figure 2.14. Note that nodes in K are *unlabelled*, hence neither marked nor unmarked.

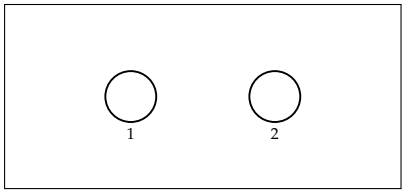


Figure 2.14: The interface of `propagate`

□

We can equip rule schemata with textual conditions that allow for relations between labels to be expressed. These conditions can also be used to express the *absence* of edges between nodes in the match.

Definition 2.32 (Rule schema condition)**.** A *rule schema condition* is an expression that can be derived from Condition in the grammar of Figure 2.15. Here, Node denotes a set of node identifiers.

Condition	::=	Type '(' List ')' | List ('=' | '\=') List
		| Integer IntRel Integer
		| `edge` '(' Node ',' Node [',' List] ')'
		| `not` Condition | Condition (`and` | `or`) Condition
IntRel	::=	'>' | '<' | '>=' | '<='
Type	::=	`int` | `string` | `atom`

Figure 2.15: Abstract syntax of rule schema conditions

□

As for list expressions, we will give a semantics for rule schema conditions with the semantics of (conditional) rule schema application. However, much of the intended meaning should be clear already from the syntax. The Type conditions are used to express that list expressions should evaluate to particular types. The `edge` conditions are used to express the existence of edges between particular nodes in a left-hand graph (the third optional parameter allows one to require a particular list expression for it). These are most useful when negated, forbidding rule applications in contexts where certain edges exist outside of (but "incident to") the match.

A conditional rule schema is a rule schema together with a condition on the expressions of its left-hand graph.

Definition 2.33 (Conditional rule schema)**.** A *conditional rule schema* is a pair $\langle r, \Gamma \rangle$ with r a rule schema and $\Gamma \in$ Condition a rule schema condition, such that all variables occurring in Γ also occur in the left-hand graph of r. □

The condition of a conditional rule schema is written underneath the graphs, and after the keyword `where`.

Example 2.34 (Conditional rule schema)**.** Figure 2.16 contains `bridge`, an example of a conditional rule schema. In an application of `bridge`, the (unmarked) nodes and edges in a path of length two would be suitable for a match, provided the condition is satisfied, i.e. there is no direct edge linking the first and third node. If this is the case, `bridge` adds an edge between these two nodes, the list component of which is constructed from the list components of the nodes. If applied repeatedly to an (unmarked) input graph, eventually we would compute its transitive closure.

□

For the sake of brevity, we will occasionally refer to conditional rule schemata as simply rule schemata. If the presence (or not) of a condition is significant in some context, we will be explicit about it.

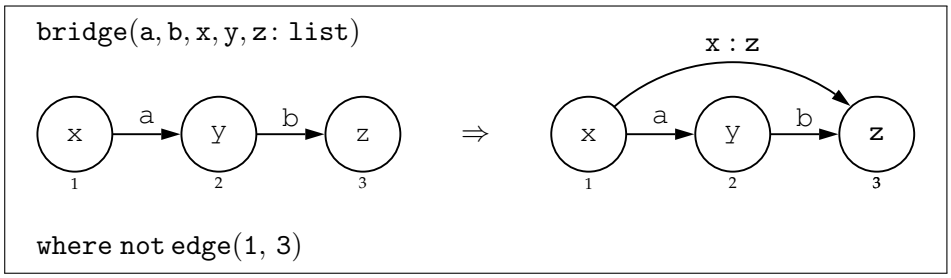


Figure 2.16: A conditional rule schema

2.2.2 Semantics of Rule Schema Application

In this subsection we define the semantics of (conditional) rule schema application which proceeds roughly as follows. For a rule schema r with condition Γ applied to a graph $G \in \mathcal{G}(\mathcal{L})$:

1. match the left-hand graph L of r with a subgraph of G, ignoring labels;
2. check whether there is an assignment α mapping variables to values in $\mathbb{L}$ such that after evaluating expressions in L, the match is label preserving;
3. check whether the condition Γ evaluates to true under the assignment and match;
4. apply to G the rule obtained from r by evaluating all expressions in the left and right graph.

We formalise these steps in what follows. First we define *premorphisms*, which are graph morphisms that disregard labels. Then, we define assignments for the evaluation of expressions, showing how to obtain a graph in $\mathcal{G}(\mathcal{L})$ from a graph in $\mathcal{G}(\text{RS})$. Finally, we define the evaluation of rule schema conditions before putting everything together and defining (conditional) rule schema application.

Since premorphisms are only required in this thesis for graph matching, we tailor our definition to the particular label alphabets, and require the functions to be injective.

Definition 2.35 (Premorphism)**.** Given graphs $L \in \mathcal{G}(\text{RS})$ and $G \in \mathcal{G}(\mathcal{L})$, a *premorphism* $g\colon L \hookrightarrow G$ consists of two injective functions $g_V : V_L \hookrightarrow V_G$ and $g_E\colon E_L \hookrightarrow E_G$ that preserve sources and targets, i.e. $s_G \circ g_E = g_V \circ s_L$ and $t_G \circ g_E = g_V \circ t_L$. □

Each graph in $\mathcal{G}(\text{RS})$ represents a possibly infinite set of graphs in $\mathcal{G}(\mathcal{L})$. The latter are obtained by mapping variables to values from $\mathbb{L}$ and evaluating expressions.

Definition 2.36 (Assignment for typed variables)**.** An *assignment* is a family of partial[1] functions $\alpha = (\alpha_X)_{X \in \{\mathrm{I,S,A,L}\}}$ where $\alpha_\mathrm{I} : \mathrm{IVar} \to \mathbb{Z}$, $\alpha_\mathrm{S} : \mathrm{SVar} \to \mathrm{Char}^*$, $\alpha_\mathrm{A} : \mathrm{AVar} \to \mathbb{Z} \cup \mathrm{Char}^*$, and $\alpha_\mathrm{L} : \mathrm{LVar} \to \mathbb{L}$. For notational convenience we often omit the subscripts of these mappings since exactly one of them is applicable to each variable. □

Definition 2.37 (Domain of typed assignments)**.** Given an assignment α, we denote by $\mathrm{dom}(\alpha)$ the *domain of* α, that is, the set of all variables `x` for which there is a partial function α_X defined for `x`. □

Definition 2.38 (Application of assignments to labels and graphs)**.** Let $(l\ b)$ denote some label with $l \in$ List and $b \in$ Mark, and let α denote some assignment defined for all variables in l. We define the *application of* α *to* $(l\ b)$, denoted by $(l\ b)^\alpha$ as follows. Define $(l\ b)^\alpha = (l^\alpha, b^\alpha) \in \mathcal{L}$, where $l^\alpha \in \mathbb{L}$ and $b^\alpha \in \mathbb{B}$ are defined inductively:

1. if $l =$ `empty`, then l^α is the empty sequence. If l is a numeral or sequence of characters, then l^α is the integer or character string[2] represented by l. If l is a variable identifier, then $l^\alpha = \alpha(l)$. If l is the expression $-i$ for some i in Integer, then $l^\alpha = -i^\alpha$. If l has the form $i_1 \oplus i_2$ with i_1, i_2 in Integer and $\oplus$ in ArithOp, then $l^\alpha = i_1^\alpha \oplus_\mathbb{Z} i_2^\alpha$ where $\oplus_\mathbb{Z}$ is the integer operation represented by $\oplus$[3]. If l has the form $s_1 \,.\, s_2$ with s_1, s_2 in String, then l^α is the string resulting from concatenating s_1^α and s_2^α. Finally, if l is a list $l_1 : l_2$ with l_1, l_2 in List, then l^α is the sequence l_1^α followed by l_2^α.

2. if $b =$ `true` (resp. `false`), then $b^\alpha =$ true (resp. false) in $\mathbb{B}$.

Given a graph G in $\mathcal{G}(\mathrm{RS})$ and an assignment α defined for all variables in G, we write G^α for the graph in $\mathcal{G}(\mathcal{L})$ that is obtained from G by replacing each label $(l\ b)$ with $(l\ b)^\alpha$ (note that G^α has the same nodes, edges, source and target functions as G). If $g\colon G \to H$ is a graph morphism with $G, H \in \mathcal{G}(\mathrm{RS})$, then g^α denotes the morphism $\langle g_V, g_E \rangle\colon G^\alpha \to H^\alpha$. □

Definition 2.39 (Application of assignments to conditions)**.** Let Γ denote some rule schema condition, $g : L \hookrightarrow G$ denote a premorphism, and α an assignment defined for all variables in Γ and L. We define the *application of* α, g *to* Γ, denoted by $\Gamma^{g,\alpha} \in \mathbb{B}$, inductively as follows.

If Γ has the form $t\ (l)$ with t in Type and l in List, then $\Gamma^{g,\alpha} =$ true if $t =$ `int` (resp. `string`, `atom`) and $l^\alpha \in \mathbb{Z}$ (resp. Char^*, $\mathbb{Z} \cup \mathrm{Char}^*$), otherwise false. If Γ has the form $l_1 = l_2$ (resp. $l_1 \mathrel{\backslash =} l_2$) with l_1, l_2 in List, then $\Gamma^{g,\alpha} =$ true if $l_1^\alpha = l_2^\alpha$ (resp. $l_1^\alpha \neq l_2^\alpha$), otherwise false. If Γ has the form $i_1 \bowtie i_2$

[1] Partiality is not needed for the purposes of this chapter but assignments will later be used for assertions, too, where partiality is convenient.

[2] Note that the empty list and empty character string are distinct values.

[3] The effect of division by zero is undefined, i.e. left to the implementation.

with i_1, i_2 in Integer and $\bowtie$ in IntRel, then $\Gamma^{g,\alpha}$ = true if $i_1^\alpha \bowtie_{\mathbb{B}} i_2^\alpha$ where $\bowtie_{\mathbb{B}}$ is the obvious Boolean-valued function on integers corresponding to $\bowtie$, otherwise false.

If Γ has the form `edge` (n_1, n_2) (resp. `edge` (n_1, n_2, l)) with n_1, n_2 node identifiers in L (resp. l in List), then $\Gamma^{g,\alpha}$ = true if there is an edge in G from $g_V(n_1)$ to $g_V(n_2)$ (resp. with list component l^α), otherwise false.

If Γ has the form `not` Γ' with Γ' in Condition, then $\Gamma^{g,\alpha}$ = true if $(\Gamma')^{g,\alpha}$ = false, otherwise false. If Γ has the form Γ_1 `and` Γ_2 (resp. Γ_1 `or` Γ_2) with Γ_1, Γ_2 in Condition, then $\Gamma^{g,\alpha}$ = true if $\Gamma_1^{g,\alpha} = \Gamma_2^{g,\alpha}$ = true (resp. $\Gamma_1^{g,\alpha}$ or $\Gamma_2^{g,\alpha}$ is true), otherwise false. □

Finally, we define the application of a (conditional) rule schema to a graph. Essentially, it requires the existence of a premorphism g and assignment α such that $r^{g,\alpha} = \langle\langle L^\alpha \hookleftarrow K \hookrightarrow R^\alpha\rangle, \Gamma^{g,\alpha}\rangle$, an *instance* of r, can be applied in the DPO approach with relabelling.

Definition 2.40 (Application of a (conditional) rule schema)**.** Given a conditional rule schema $r = \langle\langle L \hookleftarrow K \hookrightarrow R\rangle, \Gamma\rangle$ and graphs $G, H \in \mathcal{G}(\mathcal{L})$, we write $G \Rightarrow_{r,g} H$ if there is a premorphism $g\colon L \hookrightarrow G$ and an assignment α such that:

1. $g\colon L^\alpha \hookrightarrow G$ is a graph morphism;
2. $\Gamma^{g,\alpha}$ = true; and
3. $G \Rightarrow_{r^{g,\alpha},g} H$.

Here, $G \Rightarrow_{r^{g,\alpha},g} H$ denotes the application of $r^{g,\alpha}$ with match g to G in the DPO approach with relabelling (note that $\Rightarrow$ is used for both the application of conditional rule schemata as well as rules). For simplicity, we often abbreviate $G \Rightarrow_{r,g} H$ to $G \Rightarrow_r H$, and write $G \Rightarrow_{\mathcal{R}} H$ for a set of (conditional) rule schemata $\mathcal{R}$ if there is some $r \in \mathcal{R}$ such that $G \Rightarrow_r H$. □

Because the left-hand graphs of rule schemata contain only simple list expressions, for a given rule schema r and premorphism g, there is at most one instance of r that can be applied with match g (see [Plu12]).

Example 2.41 (Application of a conditional rule schema)**.** A possible application of the conditional rule schema `bridge` is shown in Figure 2.17, where:

$$\alpha_L = \{\mathtt{a} \mapsto \square, \mathtt{b} \mapsto \square, \mathtt{x} \mapsto 0\colon 1\colon 2, \mathtt{y} \mapsto 3, \mathtt{z} \mapsto 4\}.$$

The first row of the diagram displays the conditional rule schema. The second row displays the rule obtained after evaluating labels with respect to α. The third row displays a direct derivation $G \Rightarrow_{r^{g,\alpha}} H$. Note that there are three other possible matches for `bridge` in G. Note also that `bridge` could not be applied again to H with the same premorphism, because the condition would no longer be satisfied.

□

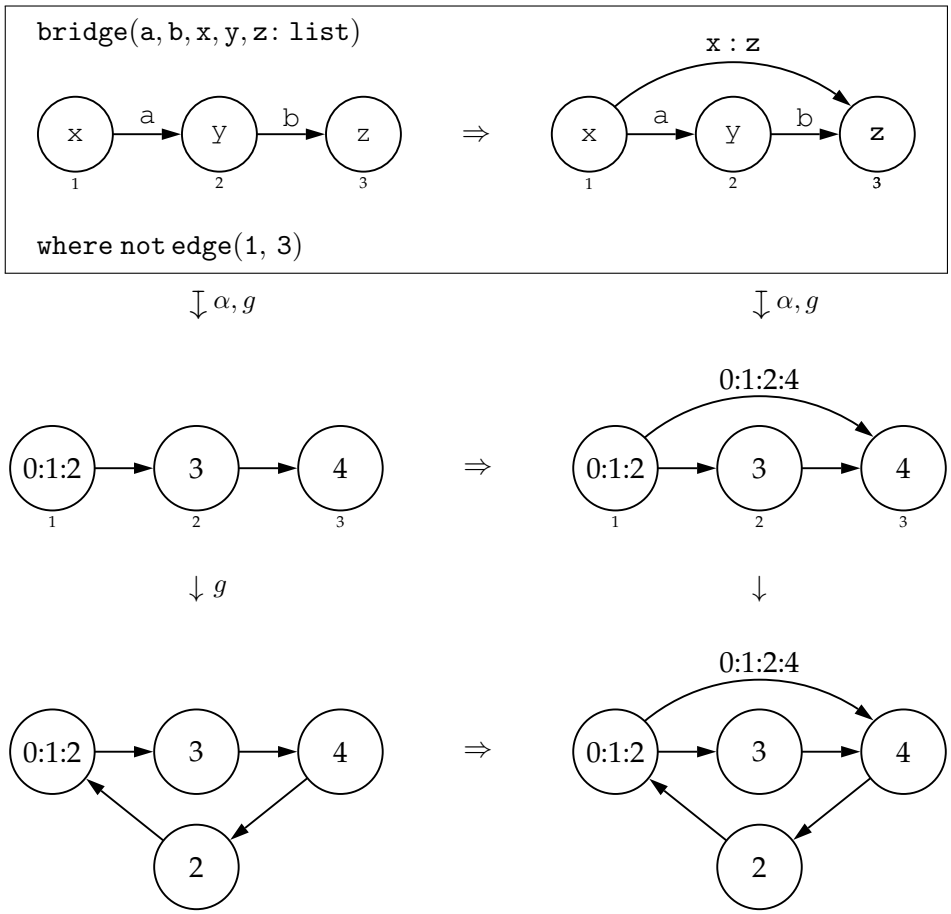


Figure 2.17: A conditional rule schema and a possible application of it

2.2.3 Abstract Syntax of Programs

Recall that graph programs consist of declarations of (conditional) rule schemata and some program text (which might be organised into macros). In this subsection, we give a grammar defining the abstract syntax of programs in GP, before giving a basic intuition into the meaning of the control constructs (a formal operational semantics will be given later).

Definition 2.42 (Abstract syntax of programs)**.** Figure 2.18 contains the grammar for the abstract syntax of graph program. The identifiers of category RuleId occurring in a RuleSetCall refer to declarations of (conditional) rule schemata in RuleDecl, which we have discussed in the previous sections. As usual, ambiguity is resolved by the use of parentheses.

□

The most fundamental command in GP is the application of a set of (conditional) rule schemata, represented in the grammar by RuleSetCall.

Prog	::=	Decl {Decl}
Decl	::=	RuleDecl | MacroDecl | MainDecl
MacroDecl	::=	MacroId '=' ComSeq
MainDecl	::=	`main` '=' ComSeq
ComSeq	::=	Com {';' Com}
Com	::=	RuleSetCall | MacroCall
		| `if` ComSeq `then` ComSeq [`else` ComSeq]
		| `try` ComSeq `then` ComSeq [`else` ComSeq]
		| ComSeq '!'
		| ComSeq `or` ComSeq
		| `skip` | `fail`
RuleSetCall	::=	RuleId | '{' [RuleId {',' RuleId}] '}'
MacroCall	::=	MacroId

Figure 2.18: Abstract syntax of graph programs

When executed, a rule schema from the set is nondeterministically applied to the input graph. If no rule schema in the set can be applied, then the execution fails. Commands can be executed sequentially using ';'. The `if` and `try` commands are the conditional constructs of GP. Rather than evaluate some Boolean function to determine whether to follow the branch after `then` or the branch after `else`, the branching is determined by executing the "guard" program given immediately after the `if` or `try`, taking the first branch if its execution results in a graph, and the other if it fails. (Note that due to nondeterminism, some guard programs may both be able to succeed and fail.) The difference between `if` and `try` is that the former executes the guard program on a *copy* of the current graph, whereas the latter does not, and hence retains all changes in a non-failing execution. Finally, as-long-as-possible iteration is represented by '!'. A command preceded by this symbol will be executed for as long as it can be done so without failure. Of course, as-long-as-possible iteration may introduce non-termination into a graph program.

The commands `or`, `skip`, and `fail` can be expressed by semantically equivalent programs made up of the other commands we have described. However, they are made available to programmers for convenience. Respectively, they represent nondeterministic choice of programs, do-nothing, and program failure.

2.2.4 Example Programs

In this subsection, we present some example programs in order to give some intuition into graph programming, before later introducing a formal operational semantics. The first program manipulates the labels of nodes to compute a graph colouring; the second marks nodes in order to check connectedness; and the third reduces an input graph to check whether it was a tree or not.

Example 2.43 (Computing a graph colouring)**.** The program `colouring` in Figure 2.19 produces a colouring (an assignment of integers to nodes such that adjacent nodes have different colours) for every input graph that is unmarked and atom-labelled, recording colours as the second elements in the list components of nodes.

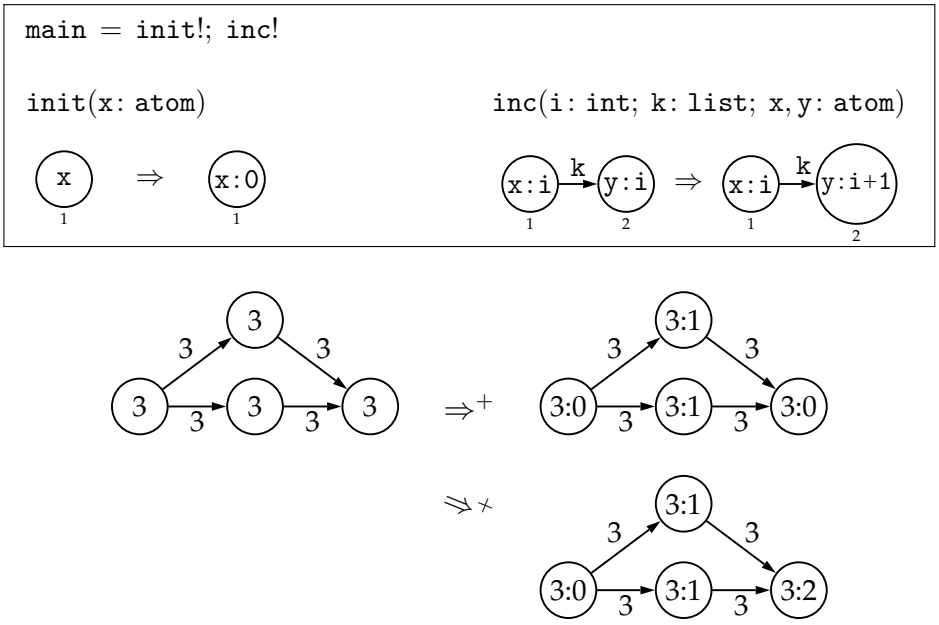


Figure 2.19: The program `colouring` and two of its executions

The program initially colours each node with 0 by applying the rule schema `init` as long as possible, using the iteration operator '!'. It then iterates `inc` for as long as possible, which matches adjacent nodes with the same colour, and increments the colour of the target node by 1. Observe that the iteration of `inc` will only end once the graph is correctly coloured, otherwise `inc` would be applied again and the iteration would continue. (Note that for simplicity the program only operates on unmarked nodes and edges, but could easily be extended for marked ones were it necessary.) □

Example 2.44 (Checking connectedness)**.** The program `connected?` in Figure 2.20 takes a graph (in which all nodes and edges are unmarked) and checks whether it is connected or not, i.e. whether or not there exists a path between any two pairs of nodes. The program provides the result – "yes" or "no" – as the label of a fresh node that it creates. In this example, we ignore the directions of edges, so a path is allowed to contain edges that are followed from the target node to the source node.

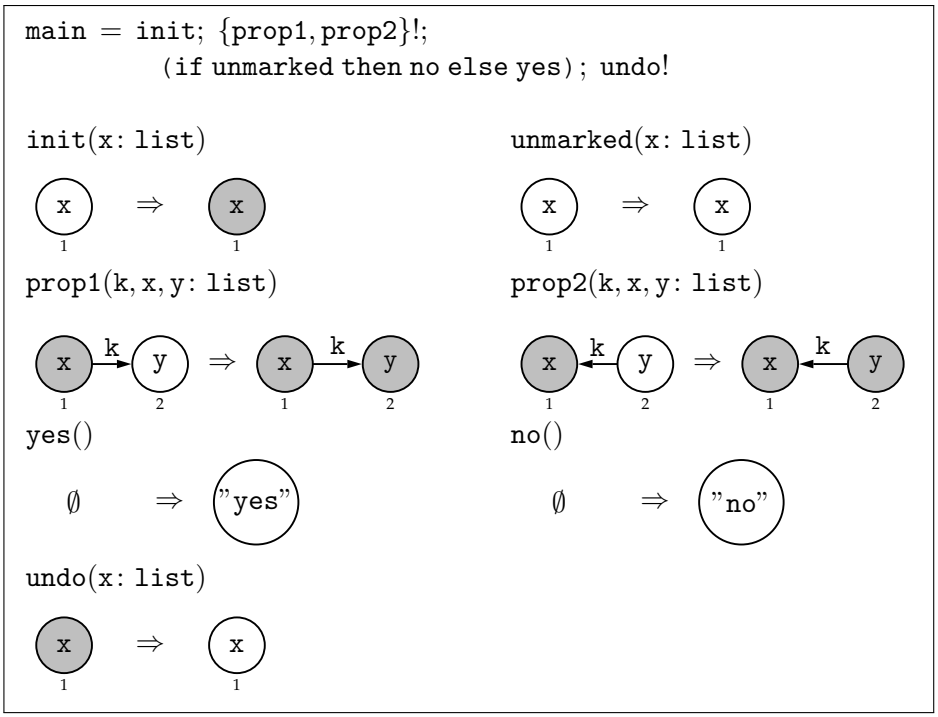


Figure 2.20: The program `connected?`

The program begins applying the rule schema `init` exactly once, which nondeterministically marks a single node in the graph. (It would fail at this point if the input graph is the empty graph $\emptyset$, or if no node is unmarked.) Then, the program iterates applications of `prop1` and `prop2`, which propagate node marks along edges in both directions. When neither of these rule schemata can be applied, the program tries to apply `unmarked` in the guard of a conditional. The rule schema does not change the graph, however, if it can be applied, then there must be an unmarked node in the graph. This means that there is some node that cannot be reached from the initially marked one, hence the graph is not connected. In this case, `no` is applied which creates a new node indicating this result. In the other case, `yes` creates a new node indicating that the graph is connected. In both cases, the program then iterates `undo` to remove all of the marks used in the computation.

A more elegant version of the program – that does not need `undo`! – would be:

$$\texttt{if init; } \{\texttt{prop1}, \texttt{prop2}\}\texttt{!; unmarked then no else yes.}$$

This version of the program performs the main part of the computation on a *copy* of the input graph, because it is wholly within the guard of an if-then-else conditional (the next example program discusses this construct further). When guards of conditionals are more than single sets of rule schemata, however, the program becomes harder to verify (see Chapters 3 and 4). □

Example 2.45 (Was it a tree?)**.** The program `tree`? of Figure 2.21 takes a graph (containing only unmarked nodes and edges) and checks whether or not it is a tree, i.e. an acyclic graph in which all nodes have indegree of at most 1, and exactly one node has indegree 0. Unlike the programs `colouring` and `connected`? which only perform label manipulations, this program performs a destructive test on the graph (i.e. deletes nodes and edges) in order to establish whether or not the original input graph was a tree or not. However, it does so within the guard of an if-then-else construct, which hides the effect of the rules. As for `connected`?, a fresh node labelled with "yes" or "no" is created to indicate the result.

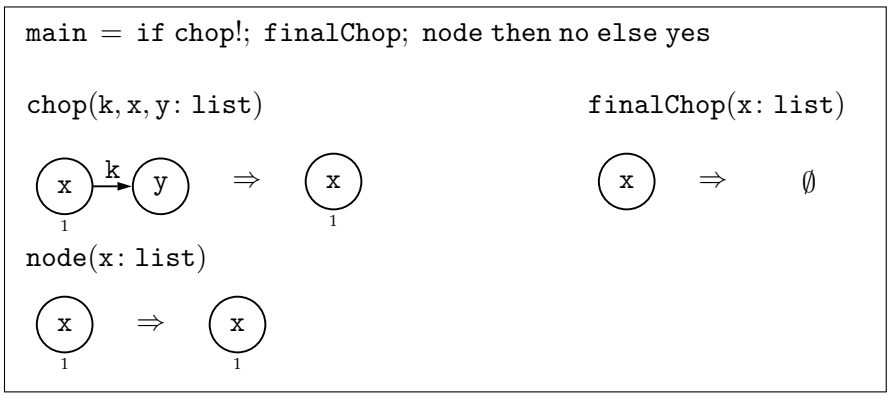


Figure 2.21: The program `tree`?

The whole program is just an instance of an if-then-else conditional. Upon execution, the program in the guard is executed on a copy of the input graph; a successful execution then leading to an execution of `no`, and a failing one leading to an execution of `yes`. Intuitively, the guard program applies a rule akin to what might be seen in a graph grammar for trees – but in *reverse* – and then determines whether the original input was a graph or not based on what the rule was not able to reduce.

The guard program begins by "chopping" the tree down to its root. It iterates the rule schema `chop`, each application of which removes a leaf

from the tree (as well as the edge pointing to it). Note that this rule schema exploits the DPO approach to match the leaves of the tree, since the node labelled with `y` can only be deleted if the only edge pointing to it is the edge labelled with `k` – otherwise edges would be left dangling (and the node would not have been a leaf in the first place). Note also that this iteration preserves an important invariant: if the graph is a tree before an application of `chop`, then it remains a tree (albeit a smaller one) after every possible application. Moreover, if the graph is *not* a tree, applying `chop` cannot transform it into a graph that is a tree.

After the iteration completes, if the original input graph was a tree, then only the root should remain. Then, the program attempts to delete this node, and checks whether or not this results in the empty graph, reporting "yes" if it is and "no" if it is not (these rule schemata are the same as in `connected`?). If the graph is not empty, then it means that there was either a node that `chop` could not delete because it was incident to more than one incoming edge (hence the graph was not a tree), or the graph was a forest of trees and hence the reduction left more than root node. □

2.2.5 Operational Semantics

GP has an operational semantics in the style of Plotkin [Plo04], which precisely describes the meaning of the language's control constructs, and which is used in the soundness proofs of our verification work in later chapters. The semantics of GP consists of inference rules, which inductively define a small-step transition relation $\to$ on *configurations*. Intuitively, configurations represent the current state (a graph or a special failure state) paired with a command sequence that remains to be executed (if any).

Definition 2.46 (Configuration)**.** A *program configuration* is either a command sequence with a graph in ComSeq $\times$ $\mathcal{G}(\mathcal{L})$, just a graph in $\mathcal{G}(\mathcal{L})$, or the special element fail. □

Definition 2.47 (Transition relation)**.** A *small-step transition relation*

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

over configurations defines the individual steps of computation. The transitive and reflexive-transitive closures of $\to$ are written $\to^+$ and $\to^*$ respectively. □

Configurations in ComSeq$\times\mathcal{G}(\mathcal{L})$ represent states of unfinished computations, whereas graphs in $\mathcal{G}(\mathcal{L})$ are proper results. The configuration fail represents a failure state. A configuration γ is said to be *terminal* if there is no configuration δ such that $\gamma \to \delta$.

We provide semantic inference rules for both the core and the derived commands of GP ("derived" in the sense that the commands have semantically equivalent programs consisting of only core commands). Each inference rule has a premise and conclusion, separated by a horizontal bar. Both contain (implicitly) universally quantified meta-variables for command sequences and graphs, where $\mathcal{R}$ stands for a call in RuleSetCall, C, P, P', Q for command sequences in ComSeq, and G, H for graphs in $\mathcal{G}(\mathcal{L})$.

Definition 2.48 (Semantic inference rules for core commands)**.** The *inference rules for core commands* of GP are given in Figure 2.22. The notation $G \not\Rightarrow_{\mathcal{R}}$ expresses that for a graph $G \in \mathcal{G}(\mathcal{L})$, there is no graph H such that $G \Rightarrow_{\mathcal{R}} H$.

$$[\mathrm{call}_1]_{\mathrm{OS}}\ \frac{G \Rightarrow_{\mathcal{R}} H}{\langle \mathcal{R},\, G\rangle \to H} \qquad [\mathrm{call}_2]_{\mathrm{OS}}\ \frac{G \not\Rightarrow_{\mathcal{R}}}{\langle \mathcal{R},\, G\rangle \to \mathrm{fail}}$$

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

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

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

$$[\mathrm{if}_2]_{\mathrm{OS}}\ \frac{\langle C,\, G\rangle \to^{+} \mathrm{fail}}{\langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q,\, G\rangle \to \langle Q,\, G\rangle}$$

$$[\mathrm{try}_1]_{\mathrm{OS}}\ \frac{\langle C,\, G\rangle \to^{+} H}{\langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q,\, G\rangle \to \langle P,\, H\rangle}$$

$$[\mathrm{try}_2]_{\mathrm{OS}}\ \frac{\langle C,\, G\rangle \to^{+} \mathrm{fail}}{\langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q,\, G\rangle \to \langle Q,\, G\rangle}$$

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

Figure 2.22: Inference rules for core commands

□

To convey an intuition as to how the rules should be read, consider the rule $[\mathrm{call}_1]_{\mathrm{OS}}$. This reads: "for all sets of (conditional) rule schemata $\mathcal{R}$ and all graphs $G, H \in \mathcal{G}(\mathcal{L})$, $G \Rightarrow_{\mathcal{R}} H$ implies that $\langle \mathcal{R}, G\rangle \to H$".

Note how if-then-else and try-then-else are distinguished. In the former, successful executions of the "guard" program C are hidden in the sense that P is executed on the original input graph G. In the latter, successful executions of C are not discarded, and P is executed on the graph

that is returned. Note also the nondeterminism with regards to failing executions in the rules for if-then-else, try-then-else, and as-long-as-possible iteration. There may be cases when a program execution might either result in a proper state or fail, depending on the execution path taken, and hence either the first or second rule for each construct is nondeterministically applied.

By inspection of the inference rules, we note that a program execution can only result in a failure state if a set of conditional rule schemata is applied to a graph for which no rule schema in the set is applicable.

Definition 2.49 (Semantic inference rules for derived commands)**.** The *inference rules for derived commands* of GP are given in Figure 2.23.

$$[\mathrm{or}_1]_{\mathrm{OS}}\ \langle P\ \texttt{or}\ Q,\ G\rangle \to \langle P,\ G\rangle \qquad [\mathrm{or}_2]_{\mathrm{OS}}\ \langle P\ \texttt{or}\ Q,\ G\rangle \to \langle Q,\ G\rangle$$

$$[\mathrm{skip}]_{\mathrm{OS}}\ \langle \texttt{skip},\ G\rangle \to G \qquad [\mathrm{fail}]_{\mathrm{OS}}\ \langle \texttt{fail},\ G\rangle \to \mathrm{fail}$$

$$[\mathrm{if}_3]_{\mathrm{OS}}\ \frac{\langle C,\ G\rangle \to^+ H}{\langle \texttt{if}\ C\ \texttt{then}\ P,\ G\rangle \to \langle P,\ G\rangle} \qquad [\mathrm{if}_4]_{\mathrm{OS}}\ \frac{\langle C,\ G\rangle \to^+ \mathrm{fail}}{\langle \texttt{if}\ C\ \texttt{then}\ P,\ G\rangle \to G}$$

$$[\mathrm{try}_3]_{\mathrm{OS}}\ \frac{\langle C,\ G\rangle \to^+ H}{\langle \texttt{try}\ C\ \texttt{then}\ P,\ G\rangle \to \langle P,\ H\rangle} \qquad [\mathrm{try}_4]_{\mathrm{OS}}\ \frac{\langle C,\ G\rangle \to^+ \mathrm{fail}}{\langle \texttt{try}\ C\ \texttt{then}\ P,\ G\rangle \to G}$$

$$[\mathrm{try}_5]_{\mathrm{OS}}\ \frac{\langle C,\ G\rangle \to^+ H}{\langle \texttt{try}\ C,\ G\rangle \to H} \qquad [\mathrm{try}_6]_{\mathrm{OS}}\ \frac{\langle C,\ G\rangle \to^+ \mathrm{fail}}{\langle \texttt{try}\ C,\ G\rangle \to G}$$

Figure 2.23: Inference rules for derived commands

□

The derived commands can be defined by the core commands. We do not go into details, but refer the reader to [Plu12].

The meaning of programs is given by the semantic function $[\![\_]\!]$, which assigns to each program P the function $[\![P]\!]$ mapping an input graph $G \in \mathcal{G}(\mathcal{L})$ to the set of all possible results of executing P on G. The application of function $[\![P]\!]$ to graph G is denoted $[\![P]\!]G$. As well as graphs, this set may contain the special values fail and $\perp$. The former indicates a program run ending in failure, whereas $\perp$ indicates that at least one execution diverges (does not terminate), or "gets stuck".

Definition 2.50 (Divergence)**.** A program P *can diverge from* graph G if there is an infinite sequence:

$$\langle P, G\rangle \to \langle P_1, G_1\rangle \to \langle P_2, G_2\rangle \to \dots$$

□

Definition 2.51 (Getting stuck)**.** A program P *can get stuck from* graph G if there is a terminal configuration $\langle Q, H\rangle$ such that $\langle P, G\rangle \to^* \langle Q, H\rangle$. □

A program can get stuck if the guard program C of a conditional can diverge on some graph G, neither producing a graph nor failing, or if the same property is true for a program that is iterated. The execution in these cases gets stuck because none of the inference rules for conditionals and iteration can be applied.

Definition 2.52 (Semantic function)**.** The *semantic function* $[\![\_]\!] : \mathrm{ComSeq} \to (\mathcal{G}(\mathcal{L}) \to 2^{\mathcal{G}(\mathcal{L}) \cup \{\mathrm{fail}, \bot\}})$, given a graph $G \in \mathcal{G}(\mathcal{L})$ and a program P, is defined by:

$$[\![P]\!]G = \{X \in \mathcal{G}(\mathcal{L}) \cup \{\mathrm{fail}\} \mid \langle P, G\rangle \to^+ X\} \\ \cup \{\bot \mid P \text{ can diverge or get stuck from } G\}.$$

□

Finally, we provide a straightforward definition of program equivalence which is based on the definition of semantic functions.

Definition 2.53 (Semantic equivalence)**.** Two graph programs P and Q are *semantically equivalent*, denoted by $P \equiv Q$, if $[\![P]\!] = [\![Q]\!]$. □

2.3 Related Work

This chapter has only touched the surface of the algebraic approach to graph transformation, presenting only what we need as a technical basis for this thesis. The recent monograph [EEPT06] provides a (rather technical) introduction to further topics, and the handbook volumes survey a range of foundations and applications [Roz97, HER99a, HER99b].

The algebraic constructions we have considered have all been in the context of graphs and graph morphisms, specifically to provide a formal basis for GP. However, much effort in graph transformation research has been focused towards generalising these frameworks to more abstract settings so that results become applicable not just for graphs, but for several other graph-like structures, e.g. hypergraphs, Petri-nets. The DPO approach (without relabelling) has been studied extensively in such abstract settings (we refer again to the monograph [EEPT06]). Recent work has considered DPO rewriting with relabelling – the formal basis of rule application in GP – in an abstract categorical setting [HP12], and generalising the results of this thesis into that framework is a possibility for future work.

GP 2, as defined in this paper, has been evolving for more than a decade since the minimal and computationally complete core presented in [HP01]. The original incarnation of conditional rule schemata was introduced by Plump and Steinert in [PS04], using the DPO with relabelling approach of [HP02] as the formal basis for their application. Steinert presented a thorough account of "GP 1" in her thesis [Ste07], which included a structural operational semantics and a number of case studies; this language, Manning began to implement [MP08]. Plump published a paper summarising the state of the GP project in 2009 [Plu09], which we used as a basis for our first papers on verification. Following lessons learnt from case studies and our verification work, the language was revised further to GP 2 [Plu12], the version used in this thesis. Among the changes were new types for rule schema labels (atoms, lists), marked nodes and edges, the try-then-else construct, and the removal of forced backtracking in the semantics of failing loop bodies and guard programs of conditionals (in order to allow a more efficient implementation). Bak and Plump have begun to address the "bottleneck" of graph matching by considering so-called rooted graph programs [BP12], in which rule schemata and host graphs contain and manipulate distinguished nodes (roots) to facilitate graph matching in constant time. The authors are proposing to implement this work and thus supersede the (now outdated) GP prototype of Manning.

GP is not the only graph transformation language: AGG [Tae04], Fujaba [NNZ00], and GrGen [GBG$^+$06] are some well-known examples. Unlike GP, these languages lack formal semantics making them less attractive to use in the context of our verification work. PROGRES [SWZ99] is another well-known graph transformation language: this *does* have a complete formal semantics due to Schürr, but unlike the semantics of GP, it is very complicated.

2.4 Summary

In this chapter we have:

- given some preliminary definitions: label alphabets, graphs, graph morphisms;
- reviewed the DPO approach to applying rules in graph transformation (i.e. no side effects in rule application), giving both concrete steps and an algebraic construction formed from two pushouts;
- reviewed the DPO approach with relabelling;
- defined conditional rule schemata, the building blocks of GP, which generalise the above approach to rewriting with expressions as labels;

- explained the control constructs of graph programs, and demonstrated them in a number of example programs;
- given a formal operational semantics for GP.

Chapter 3

Hoare Calculi for Graph Programs

In this chapter we introduce our verification calculi for graph programs, which follow the approach initiated by Hoare [Hoa69]. We introduce the proof rules of the calculi in the extensional style – i.e. independent of a fixed assertion language – before defining three notions of correctness, and showing the calculi to be sound (that is, everything provable is valid). We show too that the calculi are relatively complete (roughly, everything valid is provable), and complete for termination (if a program iteration eventually terminates, we can prove it). Finally, we briefly review related work on the verification of graph transformations.

3.1 Reasoning with Hoare Logic

Hoare logics are calculi of axioms and inference rules (collectively referred to as "proof rules") for rigorously reasoning about the correctness of programs. Central to the approach is the Hoare triple, $\{pre\}\ P\ \{post\}$, which is a specification about the behaviour of a program P when executed on a program state satisfying some precondition pre. (For now, we do not fix how to assert properties of program states.) In such situations – and ignoring for now the issue of termination – executing P establishes the postcondition $post$. The idea of a Hoare logic is to allow the derivation of such triples – such specifications – directly from the syntax of a program.

From the most fundamental program constructs, we derive such triples by instantiating axioms[1]. Take, for example, some program construct denoted by `do-nothing`, which upon execution terminates immediately without altering the program state. An obvious axiom would be:

[1] These are really "axiom schemata", but we follow other authors in simply referring to them as the axioms of the calculi.

$$\text{[do-nothing]}\ \frac{}{\{pre\}\ \texttt{do-nothing}\ \{pre\}}$$

Given the (informal) semantics of this construct, whatever is asserted in pre about the state, we can be sure that after `do-nothing` is executed the assertion pre still holds. Hence from this axiom, and in a classical setting where program states comprise a variable store, we can "prove" triples like $\vdash \{\texttt{x > 0}\}\ \texttt{do-nothing}\ \{\texttt{x > 0}\}$ or $\vdash \{\texttt{x*y = 30}\}\ \texttt{do-nothing}\ \{\texttt{x*y = 30}\}$. Here, the symbol $\vdash$ is used to indicate that the triple has been proven within some calculus, but it is often omitted when the context is clear. (Later, we use another symbol $\models$, to indicate the truth of a triple under some interpretation.)

For less fundamental constructs, Hoare logics contain inference rules. Unlike axioms, to derive a triple from these, one must first prove some other triples in the calculus using other inference rules and axioms. Take for example a sequential composition construct denoted by ';'. Informally, a program $P;\ Q$ is executed by first executing P, and upon its termination, executing Q. To yield a triple about $P;\ Q$, an inference rule might require a proof about both P and Q individually relative to some "midpoint" assertion, as follows:

$$\text{[seq-comp]}\ \frac{\{pre\}\ P\ \{mid\} \qquad \{mid\}\ Q\ \{post\}}{\{pre\}\ P;\ Q\ \{post\}}$$

Constructing and using Hoare logics is appealing because it allows reasoning around individual parts and components of programs. This is in contrast to e.g. a weakest precondition semantics (as described by Dijkstra [Dij75, Dij76]), in which a program and postcondition are recursively transformed into an assertion describing the "weakest" requirements on a state for the program to establish the postcondition. Whereas verification in the latter approach is reduced to proving that a precondition implies the weakest precondition of a program (which is often large and non-trivial), a Hoare logic approach allows for the isolation of difficult aspects of the proof, for example, in finding and proving invariants of looping programs.

To be useful, a Hoare logic needs to be *sound*. That is, if some triple can be proven in the calculus, that triple must be valid according to some notion of correctness. Typically, this involves appealing to the formal semantics of the programming language, and showing that the axioms and inference rules faithfully adhere to them. In a sense, Hoare logics capture the *essential properties* of the semantics whilst avoiding the need to work with them directly [NN07].

Another desirable property of a Hoare logic is *completeness*. That is, if one can denote a triple that is valid according to some notion of correctness, then it is possible to prove that triple within the calculus. Unfortunately, because most calculi require – for practicality – inference rules that allow

preconditions to be strengthened and postconditions to be weakened (or either to be replaced by semantically equivalent but syntactically distinct assertions), completeness is relative to the assertion language. Take such an inference rule for deriving new assertions:

$$\text{[consequence]}\ \frac{pre \text{ implies } pre' \qquad \{pre'\}\ P\ \{post'\} \qquad post' \text{ implies } post}{\{pre\}\ P\ \{post\}}$$

To derive $\vdash \{pre\}\ P\ \{post\}$, one must first prove $\vdash \{pre'\}\ P\ \{post'\}$, and then also show that the two implications in the assertion language are valid, i.e. true in all states. Unless the assertion language is very simple, the latter task is usually a source of incompleteness. For example, if the assertion language can express arithmetical expressions over the integers, then by *Gödel's incompleteness theorem* there is no complete calculus for deducing valid assertions [Cro72]. Separately, the assertion language may also cause incompleteness by not being *expressive* enough, i.e. unable to express certain preconditions required in correctness proofs.

A more realistic completeness property to aim for, due to Cook [Coo78], is *relative completeness* (or *Cook completeness*). The idea is to separate incompleteness due to the axioms and inference rules from any incompleteness in deducing valid assertions. A Hoare logic is relatively complete, if all valid triples can be proven in the calculus, relative to an oracle for deciding the validity of assertions.

3.2 Assertions for Graph Programs

In the previous section we discussed Hoare logic without much attention to assertion languages for expressing the pre- and postconditions of triples. That is, with the exception of how we separate incompleteness due to inadequate proof calculi from incompleteness due to the assertion language. This idea of separating proof rules and assertion languages is one that we continue in this chapter, even as we come to consider graph programs specifically.

Rather than fixing assertion languages for graph programs, we define our axioms, inference rules, and notions of correctness in a general way that allows for languages to be "plugged in" later. According to Nielson and Nielson [NN07], this is an *extensional approach* to Hoare logic. The approach allows us to focus on capturing the semantics of GP's various control constructs in the proof rules of our calculi without having to concern ourselves with the limitations of a particular assertion language.

Formally, we require only that assertion languages are associated with a satisfaction relation, which holds when a state (i.e. a graph) "satisfies" an assertion in some sense. What the assertions actually are (and they need

not be restricted to logical formulae), and what constitutes "satisfaction", is left open to users.

Definition 3.1 (Assertion language)**.** An *assertion language* is a pair $\mathfrak{A} = \langle A, \models_{\mathfrak{A}} \rangle$, where A is a (possibly infinite) set of *assertions*, and $\models_{\mathfrak{A}} \subseteq \mathcal{G}(\mathcal{L}) \times A$ is a *satisfaction relation*. If for a graph $G \in \mathcal{G}(\mathcal{L})$ and assertion $a \in A$ we have that $G \models_{\mathfrak{A}} a$ holds, we say that G *satisfies* a. □

Example 3.2. An example of a weak assertion language is one that can express three properties: that a graph is empty, that it is not, or that either of those properties hold. This can be defined by:

$$\mathfrak{A}_1 = \langle \{\texttt{emp}, \texttt{notemp}, \texttt{either}\}, \models_{\mathfrak{A}_1} \rangle$$

where given a graph $G \in \mathcal{G}(\mathcal{L})$,

$$\begin{aligned} G &\models_{\mathfrak{A}_1} \texttt{emp} \text{ if } V_G = \emptyset, \\ G &\models_{\mathfrak{A}_1} \texttt{notemp} \text{ if } V_G \neq \emptyset, \text{ and} \\ G &\models_{\mathfrak{A}_1} \texttt{either} \text{ always.} \end{aligned}$$

□

Example 3.3. A more powerful assertion language is one in which the assertions are DPO graph grammars and the satisfaction relation holds when a graph can be generated by such a grammar. This can be defined by $\mathfrak{A}_2 = \langle \mathcal{GG}, \models_{\mathfrak{A}_2} \rangle$ where $\mathcal{GG}$ is a class of DPO graph grammars, and $G \models_{\mathfrak{A}_2}$ GRA for $G \in \mathcal{G}(\mathcal{L}), \text{GRA} \in \mathcal{GG}$ if G can be generated by GRA.

A closely related idea is to use Graph Reduction Specifications (GRSs) [BPR04b, BPR03] as an assertion language. Unlike grammars, where one checks whether a graph can be generated from some starting graph, in GRSs, one checks whether a graph can be reduced to some accepting graph. An assertion language based on GRSs would be useful for reasoning abstractly about shape safety, e.g. manipulations of pointer structures. □

Examples 3.2 and 3.3 describe respectively weak and powerful assertion languages. The former can only express two properties, whereas the latter ones can be used to express any recursively enumerable set of graphs (see the result of Uesu [Ues78]). Whilst expressiveness is generally important, other factors are also important in choosing an assertion language for a Hoare logic. For example, is there a decision procedure for checking whether a graph satisfies an assertion? And is the language expressive enough for the calculus?

3.3 Notions of Correctness

In this section we fix our semantics of Hoare triples, i.e. what it means for $\{pre\}\ P\ \{post\}$ to be "correct". To do this we look at the semantics of graph programs, and in particular, recall the semantic function from Definition 2.52:

$$\begin{aligned}[\![P]\!]G = \{X \in \mathcal{G}(\mathcal{L}) \cup \{\text{fail}\} \mid \langle P, G\rangle \to^+ X\} \\ \cup \{\bot \mid P \text{ can diverge or get stuck from } G\}.\end{aligned}$$

For a program P executed on a graph G, the set $[\![P]\!]G$ might contain graphs, the element fail, and possibly also $\bot$. Of these possible elements, the graphs can be seen as the results of "successful" executions, i.e. they are the *proper* states. In particular, there might be several (or none) of them due to the nondeterminism, and this must be accounted for in definitions of correctness. Following standard approaches (see e.g. [AdO09]), we define our weakest notion of correctness – *partial correctness* – to consider only the proper states (i.e. graphs) that might result, without any guarantee that any will actually be returned.

Definition 3.4 (Partial correctness)**.** Let $\mathfrak{A} = \langle A, \models_{\mathfrak{A}}\rangle$ be an assertion language, and c, d be assertions in A. A graph program P is *partially correct* with respect to a precondition c and postcondition d, denoted $\models_{\text{par}} \{c\}\ P\ \{d\}$, if for every graph $G \in \mathcal{G}(\mathcal{L})$, $G \models_{\mathfrak{A}} c$ implies $H \models_{\mathfrak{A}} d$ for every H in $[\![P]\!]G$. □

With partial correctness we can make guarantees about the properties of graphs in $[\![P]\!]G$, but the definition is not strong enough to guarantee that $\bot$ (for diverging executions and those that get stuck) is absent from the set, nor strong enough to guarantee the absence of fail (for failing executions). We follow [Apt84] and define two notions of correctness to account for these distinct issues. When we refer to *weak total correctness*, we mean partial correctness as well as absence of divergence and executions that get stuck. When we refer to *total correctness*, we mean weak total correctness as well as absence of failing executions.

Definition 3.5 (Weak total correctness)**.** Let $\mathfrak{A} = \langle A, \models_{\mathfrak{A}}\rangle$ be an assertion language, and c, d be assertions in A. A graph program P is *weakly totally correct* with respect to a precondition c and postcondition d, denoted $\models_{\text{wtot}} \{c\}\ P\ \{d\}$, if $\models_{\text{par}} \{c\}\ P\ \{d\}$ and if for every graph $G \in \mathcal{G}(\mathcal{L})$ such that $G \models_{\mathfrak{A}} c$, there is no infinite sequence $\langle P, G\rangle \to \langle P_1, G_1\rangle \to \langle P_2, G_2\rangle \to \cdots$ (divergence), and there is no terminal configuration $\langle Q, H\rangle$ such that $\langle P, G\rangle \to^* \langle Q, H\rangle$ (getting stuck). □

Definition 3.6 (Total correctness)**.** Let $\mathfrak{A} = \langle A, \models_{\mathfrak{A}} \rangle$ be an assertion language, and c, d be assertions in A. A graph program P is *totally correct* with respect to a precondition c and postcondition d, denoted $\models_{\text{tot}} \{c\}\ P\ \{d\}$, if $\models_{\text{wtot}} \{c\}\ P\ \{d\}$, and if for every graph $G \in \mathcal{G}(\mathcal{L})$ such that $G \models_{\mathfrak{A}} c$, there is no derivation $\langle P, G\rangle \to^* \text{fail}$. □

We remark that in GP 1 [Plu09], it is possible for $[\![P]\!]G$ to be empty if P terminates on G but all its execution paths result in failure (this was referred to as *finite failure*). With the GP 2 semantics we use in this thesis, $[\![P]\!]G$ in this case would be a singleton set containing only fail (we refer to the semantic function and semantic inference rules in Section 2.2.5).

3.4 Partial Correctness Calculus

Having fixed three notions of correctness for Hoare triples, in this section we give the axioms and inference rules of a proof system for the first of them – partial correctness. The proof rules are introduced individually in this section, alongside informal discussions of the semantics of GP which directly motivated the choices we made. They are also presented together for reference in Figures A.1 and A.2 of the appendix.

We define the provability of triples in our partial, weak total, and total correctness calculi in the standard way. Note that we allow premises of inference rules to require provability in a different calculus; this is especially useful in the total correctness calculus, where occasionally total correctness is too strong to require of a premise. If a triple is provable, we can display a proof for it by means of a proof tree (see e.g. Example 3.13).

Definition 3.7 (Provability; proof tree)**.** Given an proof system I, a triple $\vdash_I \{c\}\ P\ \{d\}$ is *provable in* I if one can construct a *proof tree* from the axioms and inference rules of I with that triple as the root. If $\vdash_I \{c\}\ P\ \{d\}$ is an instance of an axiom X, then:

$$X\ \frac{}{\vdash_I \{c\}\ P\ \{d\}}$$

is a proof tree, and the triple is provable in I. If $\vdash_I \{c\}\ P\ \{d\}$ can be instantiated from the conclusion of an inference rule X, and there are proof trees $T_1, \ldots, T_n$ with conclusions that are instances of the n premises of X, then:

$$X\ \frac{T_1 \quad \cdots \quad T_n}{\vdash_I \{c\}\ P\ \{d\}}$$

is a proof tree, and the triple is provable in I. □

Let $\mathfrak{A} = \langle A, \models_{\mathfrak{A}} \rangle$ be an assertion language. Let r (resp. $\mathcal{R}$) range over rule schemata (resp. sets of rule schemata), c, c', d, d', e, inv over assertions

in A, and C, P, Q over graph programs. If a triple $\{c\}\ P\ \{d\}$ can be proved with our axioms and inference rules for partial correctness, we denote this by $\vdash_{\text{par}} \{c\}\ P\ \{d\}$. (The connections between *provability*, i.e. $\vdash_{\text{par}}$, and *validity*, i.e. $\models_{\text{par}}$, are made explicitly clear later in Section 3.6.)

Notation 3.8. Note that all the triples in the premises and conclusions of our proof rules for partial correctness can be assumed to be preceded by $\vdash_{\text{par}}$. □

First we consider the axiom $[\text{ruleapp}]_{\text{wlp}}$ which allows reasoning about the application of a single (conditional) rule schema.

$$[\text{ruleapp}]_{\text{wlp}}\ \frac{}{\{\text{Wlp}_{\mathfrak{A}}[r, c]\}\ r\ \{c\}}$$

Rule schema application directly manipulates the state (a graph), and hence the $[\text{ruleapp}]_{\text{wlp}}$ axiom can be seen as core to our calculus as the assignment axiom might be in a calculus for a traditional imperative language. The axiom rests entirely on the precondition $\text{Wlp}_{\mathfrak{A}}[r, c]$, which is an assertion in A that in this extensional approach we do not define, but rather characterise in Definition 3.10. Informally, $\text{Wlp}_{\mathfrak{A}}[r, c]$ should express a *weakest liberal precondition* relative to r and c, i.e. the weakest restrictions on a graph such that any successful application of r results in a graph that establishes the postcondition c (note that a successful application of r is not guaranteed by satisfying the precondition).

Definition 3.9 (Liberal precondition)**.** Given an assertion language $\mathfrak{A} = \langle A, \models_{\mathfrak{A}} \rangle$, a program P, and a postcondition c in A, an assertion a in A is a *liberal precondition relative to* P, c if for all graphs G such that $G \models_{\mathfrak{A}} a$,

$$H \in [\![P]\!]G \ \text{ implies } \ H \models_{\mathfrak{A}} c.$$

□

Definition 3.10 (Weakest liberal precondition $\text{Wlp}_{\mathfrak{A}}$)**.** Given an assertion language $\mathfrak{A} = \langle A, \models_{\mathfrak{A}} \rangle$, a program P, and a postcondition c in A, $\text{Wlp}_{\mathfrak{A}}[P, c]$ denotes a *weakest liberal precondition relative to* P, c, that is, a liberal precondition in A such that for all graphs G and all other liberal preconditions $a \in A$,

$$G \models_{\mathfrak{A}} a \ \text{ implies } \ G \models_{\mathfrak{A}} \text{Wlp}_{\mathfrak{A}}[P, c].$$

□

An important question to ask is whether a particular assertion language $\mathfrak{A}$ is expressive enough to actually define the assertion $\mathrm{Wlp}_{\mathfrak{A}}[r, c]$ – as we have characterised it – for all combinations of r, c. If there is not such an assertion in A, then the assertion language introduces incompleteness. Whether a construction actually exists for transforming an arbitrary rule schema and postcondition into an assertion $\mathrm{Wlp}_{\mathfrak{A}}[r, c]$ is thus an important consideration for choosing an assertion language.

A set of rule schemata $\mathcal{R}$ is applied to a graph by nondeterministically choosing an applicable rule schema from the set and applying it to the graph. Hence to derive a triple about $\mathcal{R}$, we must prove the same triple but for each each of the individual rule schemata in $\mathcal{R}$ – this is captured by the proof rule [ruleset]:

$$\text{[ruleset]}\ \frac{\{c\}\ r\ \{d\} \text{ for each } r \in \mathcal{R}}{\{c\}\ \mathcal{R}\ \{d\}}$$

For each $\{c\}\ r\ \{d\}$ it might be the case that c is a suitable assertion for $\mathrm{Wlp}_{\mathfrak{A}}[r, d]$, and so each triple is proved simply by instantiating $[\mathrm{ruleapp}]_{\mathrm{wlp}}$. But it might also be the case that c is stronger than the weakest liberal precondition, and so we need a rule of consequence:

$$\text{[cons]}\ \frac{\mathrm{impl}(c, c') \quad \{c'\}\ P\ \{d'\} \quad \mathrm{impl}(d', d)}{\{c\}\ P\ \{d\}}$$

$\mathrm{impl}(a_1, a_2)$: prove that $G \models_{\mathfrak{A}} a_1$ implies $G \models_{\mathfrak{A}} a_2$ for all $G \in \mathcal{G}(\mathcal{L})$

The aim of our [cons] rule is no different from rules of consequence in other Hoare logics: it allows for preconditions to be strengthened, postconditions to be weakened, or either to be replaced by semantically equivalent but syntactically distinct assertions. It requires two proofs – outside of the calculus – about the relationships between pre- and postconditions in the premise and the conclusion of the rule. If the assertion language is logical, then what is required is essentially to prove the validity of $c \Rightarrow c'$ and $d' \Rightarrow d$. But in our more general setting where assertion languages could be non-logical (e.g. graph grammars), we describe – via the abbreviation $\mathrm{impl}(a_1, a_2)$ – the essential property that needs to be shown. That is, regardless of state, satisfying c is sufficient for satisfying c', and satisfying d' is sufficient for satisfying d. The difficulty of checking these implications (which would ideally be automatic), is an important consideration in choosing an assertion language.

Sequential composition is captured by the [comp] rule:

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

In an execution of $P; Q$, the program Q is not executed until after the execution of P has terminated. So to prove a partial correctness triple about $P; Q$, it suffices to prove triples about P and Q separately relative to some "midpoint", i.e. some assertion e that holds for a graph after a successful execution of P and before an execution of Q.

Next we consider the proof rule [if] for the first of GP's conditional constructs:

$$\text{[if]}\ \frac{\{\text{SE}_{\mathfrak{A}}[c, C]\}\ P\ \{d\} \quad \{\text{FE}_{\mathfrak{A}}[c, C]\}\ Q\ \{d\}}{\{c\}\ \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}}$$

The [if] rule requires two further special assertions, which we characterise more formally shortly. Intuitively, $\text{SE}_{\mathfrak{A}}[c, C]$ is an assertion satisfied by graphs for which assertion c holds, and for which program C has an execution path that yields a graph ("Successful Execution"). On the other hand, $\text{FE}_{\mathfrak{A}}[c, C]$ is an assertion satisfied by graphs for which c holds, and for which C has an execution path that yields a fail state ("Failing Execution"). These assertions are used in the premises of [if] to appropriately capture the semantics of `if` C `then` P `else` Q. Recall that in an execution of this program on a graph G, the program C is first executed on a *copy* of G. If C terminates and results in a graph, then P is executed on G. If C terminates and results in a fail state, then Q is executed on G instead. (Note that $[\![C]\!]G$ could contain both a graph and fail; in such a case either P or Q could be executed.)

Definition 3.11 (Assertion $\text{SE}_{\mathfrak{A}}$)**.** Given an assertion language $\mathfrak{A} = \langle A, \models_{\mathfrak{A}} \rangle$, a graph program C, and an assertion c in A, $\text{SE}_{\mathfrak{A}}[c, C]$ is any assertion in A such that:

$$G \models_{\mathfrak{A}} \text{SE}_{\mathfrak{A}}[c, C] \ \text{ if and only if } \ (G \models_{\mathfrak{A}} c \text{ and } H \in [\![C]\!]G)$$

for some $H \in \mathcal{G}(\mathcal{L})$. □

Definition 3.12 (Assertion $\text{FE}_{\mathfrak{A}}$)**.** Given an assertion language $\mathfrak{A} = \langle A, \models_{\mathfrak{A}} \rangle$, a graph program C, and an assertion c in A, $\text{FE}_{\mathfrak{A}}[c, C]$ is any assertion in A such that:

$$G \models_{\mathfrak{A}} \text{FE}_{\mathfrak{A}}[c, C] \ \text{ if and only if } \ (G \models_{\mathfrak{A}} c \text{ and fail} \in [\![C]\!]G)\,.$$

□

As for $\mathrm{Wlp}_{\mathfrak{A}}$, it is important to consider whether the assertion language is expressive enough to define $\mathrm{SE}_{\mathfrak{A}}[c, C]$ and $\mathrm{FE}_{\mathfrak{A}}[c, C]$ for all combinations of c and C. Actually, if the language is expressive enough to construct them effectively, the problem of checking whether a graph satisfies an assertion becomes undecidable. This problem, which we discuss and prove later in Section 3.6.1, is due to the fact that C can be an arbitrary program – including one that employs as-long-as-possible iteration. When we define a particular assertion language in Chapter 4, we get around this problem by restricting C to a particular class of programs – one for which $\mathrm{SE}_{\mathfrak{A}}[c, C]$ and $\mathrm{FE}_{\mathfrak{A}}[c, C]$ can be constructed, and one for which the problem of checking whether a graph satisfies an assertion can remain decidable.

We remark too that in several proofs it is the assertion c that is needed to establish the postcondition d, and not an assertion about the executions of C. In such cases, a simpler version of [if] could be used in which the premises are $\vdash_{\text{par}} \{c\}\ P\ \{d\}$ and $\vdash_{\text{par}} \{c\}\ Q\ \{d\}$, and then the issues of defining $\mathrm{SE}_{\mathfrak{A}}[c, C]$ and $\mathrm{FE}_{\mathfrak{A}}[c, C]$ disappear.

The proof rule for GP's other conditional construct – `try` – is similar to that for [if]:

$$[\text{try}]\ \frac{\{\mathrm{SE}_{\mathfrak{A}}[c, C]\}\ C; P\ \{d\} \quad \{\mathrm{FE}_{\mathfrak{A}}[c, C]\}\ Q\ \{d\}}{\{c\}\ \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}}$$

The program `try` C `then` P `else` Q behaves at execution in the same way as `if` C `then` P `else` Q, except that C is not executed on a copy of the graph. Rather, if its execution is successful, the program P is executed on the new graph; the changes retained. This is captured by sequential composition in the first of the premises.

The remaining core construct in GP is as-long-as-possible iteration, captured by the proof rule [!] below:

$$[!]\ \frac{\{inv\}\ P\ \{inv\}}{\{inv\}\ P!\ \{\mathrm{FE}_{\mathfrak{A}}[inv, P]\}}$$

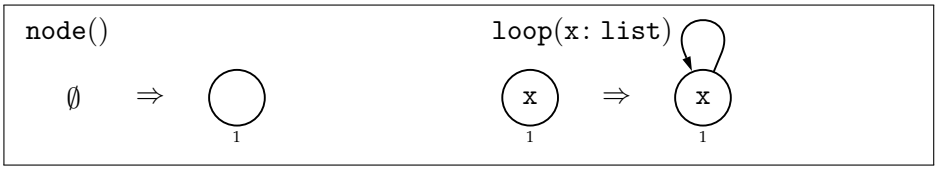
The intuition behind the rule should be simple: if an assertion inv (for invariant) holds before and after a single execution of P, then it should also hold after any number of executions of P. If $P!$ terminates, then from the semantics of the construct we know that the final execution of P resulted in a fail state, and so $\mathrm{FE}_{\mathfrak{A}}[inv, P]$ must hold too. We remark that if a proof requires only the invariant, an application of [cons] yields $\vdash_{\text{par}} \{inc\}\ P!\ \{inv\}$. We remark also that this construct is a potential source of

non-termination (i.e. divergence or getting stuck), and the proof rule does not prove anything about its absence.

The partial correctness proof rules discussed are given together in Figure A.1 of the appendix. Additionally, in Figure A.2 we present some further proof rules for the derived commands of GP. These are given for convenience, but are not required for completeness since derived commands all have semantically equivalent programs comprised only of core commands.

Example 3.13 (Partial correctness proof)**.** We construct a partial correctness proof of a very simple program, using $\mathfrak{A}_1$ of Example 3.2 as the assertion language. Recall that this language comprises three assertions: `emp`, `notemp`, and `either` which respectively correspond to expressing that a graph is empty, non-empty, or either (clearly a property that all graphs satisfy). The assertion language is too weak to allow us to prove anything interesting – this will come in Chapters 4 and 5 – but allows us to demonstrate how to instantiate the proof rules, and moreover how one can display a proof in the form of a proof tree.

Consider the program {`node`, `loop`}, the rule schemata of which are:



We will prove $\vdash_{\text{par}} \{\texttt{emp}\}\ \{\texttt{node}, \texttt{loop}\}\ \{\texttt{notemp}\}$, that is, if the program is executed on an empty graph, any graph resulting is non-empty. A proof of this triple is presented below, in the form of a proof tree. Here, the "root" is at the bottom of the tree and is the triple we desire to prove. The "leaves" are instances of the axioms, with the rest of the proof tree comprising instances of inference rules.

$$\dfrac{\dfrac{[\text{ruleapp}]_{\text{wlp}}\ \dfrac{}{\{\texttt{either}\}\ \texttt{node}\ \{\texttt{notemp}\}}}{[\text{cons}]\ \dfrac{}{\{\texttt{emp}\}\ \texttt{node}\ \{\texttt{notemp}\}}} \qquad \dfrac{[\text{ruleapp}]_{\text{wlp}}\ \dfrac{}{\{\texttt{either}\}\ \texttt{loop}\ \{\texttt{notemp}\}}}{[\text{cons}]\ \dfrac{}{\{\texttt{emp}\}\ \texttt{loop}\ \{\texttt{notemp}\}}}}{[\text{ruleset}]\ \vdash_{\text{par}} \{\texttt{emp}\}\ \{\texttt{node}, \texttt{loop}\}\ \{\texttt{notemp}\}}$$

Here, we have taken `either` to define both $\text{Pre}_{\mathfrak{A}_1}[\texttt{node}, \texttt{notemp}]$ and $\text{Pre}_{\mathfrak{A}_1}[\texttt{loop}, \texttt{notemp}]$, although we have not attempted to define a general construction for arbitrary rule schemata and assertions. Rather, we can argue that in these two cases `either` fits the characterisation in Definition 3.10. For `node` this is obvious, but for `loop` one must remember that we are working with partial correctness. A graph can only ever result if the rule schema is executed on a non-empty graph; in this case it is clear that the graph cannot become non-empty. We remark that {`emp`} `loop` {`notemp`} is

not true in the sense of total correctness, since executing `loop` on the empty graph always results in failure. We remark also on our convention that we do not display the implications required in the premises of [cons]. These are treated separately when they are non-trivial (in this example they obviously hold). □

Example 3.14 (Partial correctness proof)**.** In this example we will prove a property of a graph program relative to assertions from a much more expressive language – one in which the assertions are also graph programs! Define the assertion language $\mathfrak{P} = \langle \mathcal{P}, \models_{\mathfrak{P}} \rangle$ where $\mathcal{P}$ is the class of all graph programs, and $G \models_{\mathfrak{P}} P$ holds for $G \in \mathcal{G}(\mathcal{L})$, $P \in \mathcal{P}$ if $G \in [\![P]\!]\emptyset$, i.e. G is a possible result of executing P on the empty graph. (Of course, in general, checking $G \models_{\mathfrak{P}} P$ is undecidable.)

In this example we consider the program `makecomplete` in Figure 3.1 that transforms a loop-free graph containing n nodes into a complete graph of n nodes (i.e. a graph in which all pairs of nodes are linked by an edge in each direction).

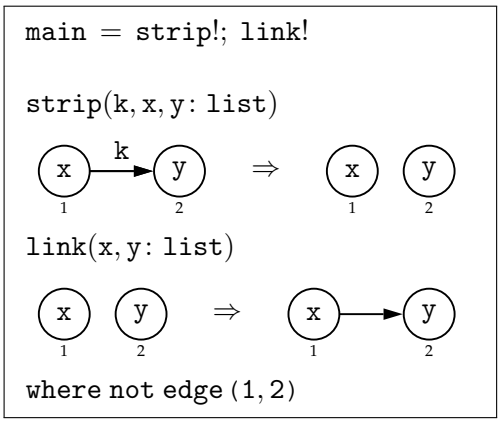


Figure 3.1: The program `makecomplete`

We will prove the triple $\vdash_{\text{par}} \{\texttt{noloop}\}\ \texttt{strip!; link!}\ \{\texttt{connected}\}$ where `noloop` is a program generating all non-empty loop-free graphs labelled over the empty list, and `connected` is a program generating all non-empty connected graphs labelled over the empty list (i.e. there is at least one path between every pair of nodes, ignoring the directions of edges). Note that we do not consider marked nodes or edges in this example. We display a proof of the triple below, again with a proof tree.

$$\dfrac{\dfrac{[\text{ruleapp}]_{\text{wlp}}\ \dfrac{}{\{\texttt{noloop}\}\ \texttt{strip}\ \{\texttt{noloop}\}}}{[!]\ \dfrac{}{\{\texttt{noloop}\}\ \texttt{strip!}\ \{\texttt{noedge}\}}} \quad [\text{cons}]\ \dfrac{[\text{ruleapp}]_{\text{wlp}}\ \dfrac{}{\{\texttt{nopara}\}\ \texttt{link}\ \{\texttt{nopara}\}}}{[!]\ \dfrac{}{\{\texttt{nopara}\}\ \texttt{link!}\ \{\texttt{complete}\}}}}{[\text{comp}]\ \dfrac{}{\vdash_{\text{par}} \{\texttt{noloop}\}\ \texttt{strip!; link!}\ \{\texttt{connected}\}}}$$

The programs used as assertions are given in the table below:

Program Name	Program Text
`noloop`	`main` $=$ `node`; $(\{$`node`, `edge`$\}$ `or fail`$)!$
`noedge`	`main` $=$ `node`; $($`node or fail`$)!$
`nopara`	`main` $=$ `node`; $(\{$`node`, `link`$\}$ `or fail`$)!$
`complete`	`main` $=$ `node`; $($`node or fail`$)!$; `bilink`$!$
`connnected`	`main` $=$ `node`; $(\{$`expand1`, `expand2`, `edge`$\}$ `or fail`$)!$

where the rule schemata are as given in Figure 3.2. The programs are intended for execution on the empty graph, each of them initialising by creating a single node. The programs then behave essentially as graph grammars, applying a set of rules repeatedly via the as-long-as-possible construct, with "`or fail`" used to allow for nondeterministic terminations of the iterations. The program `complete` has a slightly different form in that there are two iterations: the first creates some number of nodes (determined nondeterministically) and the second creates edges between all pairs of nodes until the graph is complete.

To complete the proof we must justify the implications in the instance of [cons], and the special assertions demanded by $[\text{ruleapp}]_{\text{wlp}}$ and [!]. To avoid bloating the example, we present the justifications in this case very informally.

The instance of [cons] requires that graphs satisfying `noedge` also satisfy `nopara`, and that graphs satisfying `complete` also satisfy `connected`. For the former, this is clear from the program texts: every graph computed by `noedge` can be computed by `nopara` by nondeterministically choosing `node` in each iteration. The latter case is more difficult to show by appealing to the program text, but assuming that `complete` does correctly generate complete graphs and `connected` does correctly generate connected graphs, it should be intuitively clear that complete graphs are also connected (there are paths of length one between all pairs of nodes).

In the instances of [!] we must argue that `noedge` is a suitable assertion for $\text{FE}_{\mathfrak{P}}[\texttt{noloop}, \texttt{strip}]$ and that `complete` is a suitable assertion for $\text{FE}_{\mathfrak{P}}[\texttt{nopara}, \texttt{link}]$. To do this we appeal to Definition 3.12 and consider the assertions in turn.

$$G \models_{\mathfrak{P}} \texttt{noedge} \text{ if and only if } (G \models_{\mathfrak{P}} \texttt{noloop} \text{ and fail} \in [\![\texttt{strip}]\!]G)$$

Satisfying `noedge` implies the satisfaction of `noloop` (by examination of the program texts), and `strip` will fail since there are no edges in the graph. Conversely, satisfying `noloop` guarantees there are no loops, and having $\text{fail} \in [\![\texttt{strip}]\!]G$ means there is also an absence of non-looping edges. Together, the graph must also satisfy `noedge`.

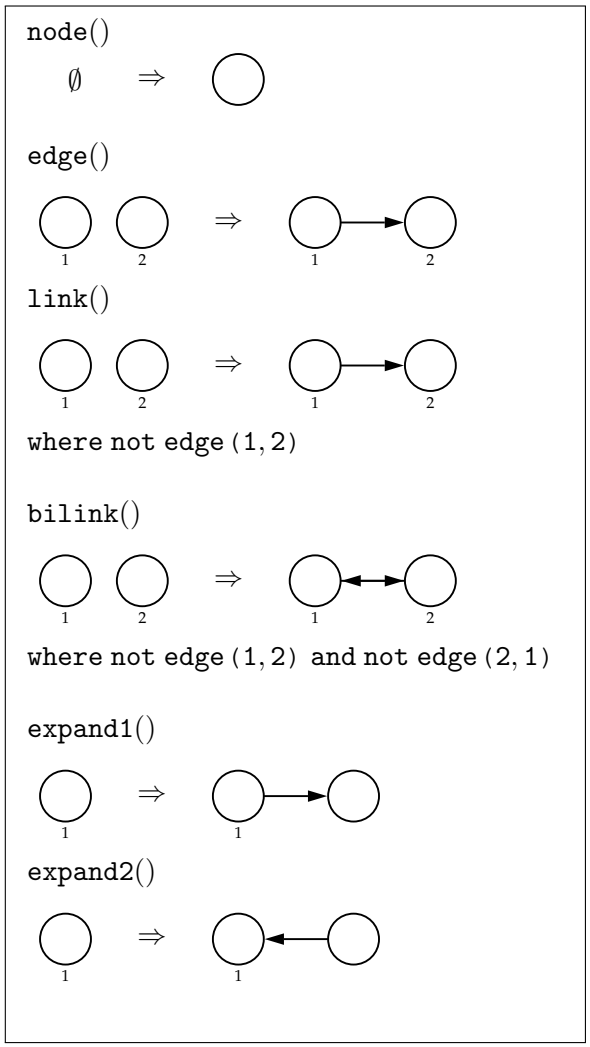


Figure 3.2: Rule schemata for the `makecomplete` proof tree

$$G \models_{\mathfrak{P}} \texttt{complete} \text{ if and only if } (G \models_{\mathfrak{P}} \texttt{nopara} \text{ and fail} \in [\![\texttt{link}]\!]G)$$

Satisfying `complete` implies the satisfaction of `nopara`, due to the condition in the rule schema `bilink`, and `link` will fail since the iterated application of `bilink` will have already added edges between all pairs of nodes, meaning that the condition of `link` will not be satisfied by any pair of nodes. Conversely, satisfying `nopara` guarantees the absence of parallel edges and loops, and having fail $\in [\![\texttt{link}]\!]G$ means that there is not a pair of nodes without an edge between them. Together, the graph must be complete and hence also satisfy `complete`.

Finally, it should be argued that `noloop` (resp. `nopara`) is a suitable assertion for $\mathrm{Wlp}_{\mathfrak{P}}[\texttt{strip}, \texttt{noloop}]$ (resp. $\mathrm{Wlp}_{\mathfrak{P}}[\texttt{link}, \texttt{nopara}]$). Informally, this can be seen by observing that the rule schemata are independent of the assertions, in that they can only establish the desired postconditions if they already hold before an execution. They cannot cause them to stop holding if they held before, and if they did not hold before, they can never cause them to hold after their executions. □

Note that in both examples we had to apply creativity to determine suitable assertions for [!] and $[\text{ruleapp}]_{\text{wlp}}$, then justify them on a case-by-case basis. We alleviate this challenge in Chapter 4 when we instantiate the proof calculi with an assertion language for which there are transformations defining the special assertions Wlp, SE, and FE.

3.5 Total Correctness Calculi

In this section we extend our proof system for partial correctness with proof rules for reasoning about termination. Recall from Section 3.3 the two notions correctness we defined for this – weak total correctness then total correctness – respectively adding guarantees about termination and absence of failure. We introduce proof systems for the two notions of correctness in this order. Again, we introduce the new proof rules one-by-one, with all of them displayed together for reference in Figures A.3 and A.5 of the appendix.

First, we extend our calculus to account for weak total correctness. Let $\mathfrak{A} = \langle A, \models_{\mathfrak{A}} \rangle$ be an assertion language. Let r (resp. $\mathcal{R}$) range over rule schemata (resp. sets of rule schemata), c, c', d, d', e, inv over assertions in A, and C, P, Q over graph programs. If a triple $\{c\}\ P\ \{d\}$ can be proven in our system for weak total correctness, we denote this by $\vdash_{\text{wtot}} \{c\}\ P\ \{d\}$.

Notation 3.15. Note that unless explicitly stated otherwise, all the triples in the premises and conclusions of our proof rules for weak total correctness can be assumed to be preceded by $\vdash_{\text{wtot}}$. □

Our proof system for weak total correctness includes already some partial correctness proof rules of Figure A.1, but with each $\vdash_{\text{par}}$ replaced with $\vdash_{\text{wtot}}$: $[\text{ruleapp}]_{\text{wlp}}$, [ruleset], [comp], and [cons]. These proof rules are already sound in the sense of weak total correctness.

The new proof rules for the conditional constructs of GP are as they are for partial correctness, but with an additional premise:

$$[\text{if}]_{\text{wtot}}\ \frac{\{c\}\ C\ \{\top_{\mathfrak{A}}\}\quad \{\text{SE}_{\mathfrak{A}}[c,C]\}\ P\ \{d\}\quad \{\text{FE}_{\mathfrak{A}}[c,C]\}\ Q\ \{d\}}{\{c\}\ \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}}$$

$$[\text{try}]_{\text{wtot}}\ \frac{\{c\}\ C\ \{\top_{\mathfrak{A}}\}\quad \{\text{SE}_{\mathfrak{A}}[c,C]\}\ C;P\ \{d\}\quad \{\text{FE}_{\mathfrak{A}}[c,C]\}\ Q\ \{d\}}{\{c\}\ \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}}$$

In these proof rules, $\top_{\mathfrak{A}}$ is an assertion satisfied by every graph $G \in \mathcal{G}(\mathcal{L})$, i.e. the weakest possible assertion. In a logical language, true would be such an assertion. (Actually, the rules remain sound for any assertion in place of $\top_{\mathfrak{A}}$. But completeness is affected if we unnecessarily constrain the permissible results of executing C.) The intuition behind the new premises is this: in proving $\vdash_{\text{wtot}} \{c\}\ C\ \{\top_{\mathfrak{A}}\}$, we can be sure that no execution of the program C on a graph satisfying c will diverge or get stuck. Hence we can be sure that P or Q will always be executed eventually, and the termination of these programs is then guaranteed by the other premises.

The construct that can introduce non-termination to a graph program is as-long-as-possible iteration, for which we introduce a new proof rule below:

$$[!]_{\text{wtot}}\ \frac{\{inv\}\ P\ \{inv\}\quad P \text{ is } \#\text{-decreasing under } inv}{\{inv\}\ P!\ \{\text{FE}_{\mathfrak{A}}[inv,P]\}}$$

The first premise requires a derivation of $\vdash_{\text{wtot}} \{inv\}\ P\ \{inv\}$ in our proof system for weak total correctness, which establishes inv as an invariant (as in [!]), but also establishes that P will terminate (this excludes the possibility of $P!$ getting stuck). The second premise requires that there is a function mapping graphs to natural numbers, such that if an execution of P on G yields a graph, that graph is mapped to a smaller number than G. Such a function $\#$ is called a *termination function*. If this property holds for graphs G satisfying inv, we say that P is *$\#$-decreasing under inv*. These definitions are given more precisely below.

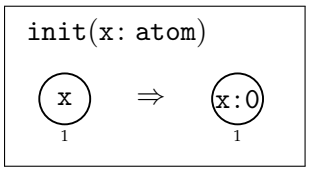
Definition 3.16 (Termination function; $\#$-decreasing)**.** A *termination function* is a mapping $\# : \mathcal{G}(\mathcal{L}) \to \mathbb{N}$ from (semantic) graphs to natural numbers. Let $\mathfrak{A} = \langle A, \models_{\mathfrak{A}} \rangle$ be an assertion language. Given an assertion a in A, a graph program P is *$\#$-decreasing (under a)* if for all graphs G, H in $\mathcal{G}(\mathcal{L})$ (such that $G \models_{\mathfrak{A}} a$ and $H \models_{\mathfrak{A}} a$),

$$H \in [\![P]\!]G \text{ implies } \#G > \#H.$$

□

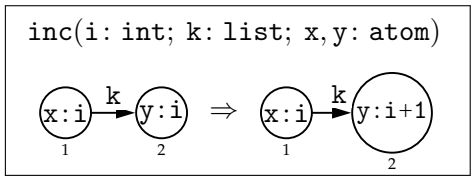
In an application of $[!]_{\text{wtot}}$, one must find a suitable termination function $\#$ that returns smaller natural numbers along the graphs of program executions. A simple, intuitive termination function would be one that maps a graph to its size (e.g. total number of nodes and edges). If a program is reducing the size of a graph upon each application, then clearly the iteration cannot continue indefinitely, and this is reflected by the output of $\#$ tending towards zero. However, in cases when programs are not necessarily decreasing the size of the graph, much less trivial termination functions may be needed. (We mention that the problem of deciding whether a program – or even just a set of rule schemata – is terminating or not, is undecidable in general [Plu98].) Note that the rule $[!]_{\text{wtot}}$ requires only that $\#$ is decreasing for graphs that satisfy the invariant inv, i.e. it need not be decreasing for graphs outside of the particular context described by the invariant.

Example 3.17 (Termination function for wtot proof)**.** Recall the rule schema `init` from the `colouring` program of Figure 2.19:



For `init`, we can define a termination function $\#_{\texttt{init}} : \mathcal{G}(\mathcal{L}) \to \mathbb{N}$ to map graphs to the number of their nodes labelled by a single atom. The rule schema is clearly $\#_{\texttt{init}}$-decreasing, since every application of `init` reduces by one the number of nodes with such a label. □

Example 3.18 (Termination function for wtot proof)**.** Recall the rule schema `inc` from the `colouring` program of Figure 2.19:



This rule schema requires a less obvious termination function than the one proposed in Example 3.17. For simplicity, we will define (in English, for now) an invariant inv, and define a termination function for `inc` that assumes the invariant holds. We denote by inv the graph property described in this box:

> "every node label has the form $x:i$ for some atom x, and colour i (which is a natural number). Moreover, there is a sequence $0 \trianglelefteq \cdots \trianglelefteq n$ for $n \geq 0$, including each of the colours used in the graph, where $p \trianglelefteq q$ holds if $q = p$ or $q = p + 1$."

Given any graph $G \in \mathcal{G}(\mathcal{L})$, we define the termination function $\#_{\mathtt{inc}} : \mathcal{G}(\mathcal{L}) \to \mathbb{N}$ by:

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

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

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

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

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

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

We remark that `inc`! will terminate on any graph – not just those satisfying inv. A termination function however is harder to write without the assumptions the invariant allows us to make about the graphs. Of course, in our Hoare logic approach, we are generally interested in proving more than termination alone. Hence it will often be possible to exploit partial correctness properties (in particular, invariants) to help us to write (weak) total correctness proofs. □

The weak total correctness proof rules discussed are given together in Figure A.3 of the appendix. Additionally, in Figure A.4 we present some further proof rules for the derived commands of GP. (Again, these are not necessary for completeness since derived commands all have semantically equivalent programs comprised only of core commands.)

Finally, we introduce our proof system for the strongest of our notions of correctness – total correctness – which guarantees in addition to termination the absence of failing executions. If a triple $\{c\}\ P\ \{d\}$ can be proven in our system for total correctness, we denote this by $\vdash_{\text{tot}} \{c\}\ P\ \{d\}$.

Notation 3.19. Note that unless explicitly stated otherwise, all the triples in the premises and conclusions of our proof rules for total correctness can be assumed to be preceded by $\vdash_{\text{tot}}$. □

Our proof system for total correctness includes [comp] and [cons] of Figure A.1 but with $\vdash_{\text{tot}}$ preceding the triples. These proof rules are already sound in the sense of total correctness. The proof rules for many of the other constructs are very similar to those we have discussed already, except that they explicitly demand certain premises to be proved for a weaker notion of correctness (which we shall see immediately in the first of the proof rules).

First, we add new proof rules for the conditional and iteration constructs:

$$[\text{if}]_{\text{tot}}\ \frac{\vdash_{\text{wtot}} \{c\}\ C\ \{\top_{\mathfrak{A}}\} \quad \{\text{SE}_{\mathfrak{A}}[c, C]\}\ P\ \{d\} \quad \{\text{FE}_{\mathfrak{A}}[c, C]\}\ Q\ \{d\}}{\{c\}\ \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}}$$

$$[\text{try}]_{\text{tot}}\ \frac{\vdash_{\text{wtot}} \{c\}\ C\ \{\top_{\mathfrak{A}}\} \quad \{\text{SE}_{\mathfrak{A}}[c, C]\}\ C; P\ \{d\} \quad \{\text{FE}_{\mathfrak{A}}[c, C]\}\ Q\ \{d\}}{\{c\}\ \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}}$$

$$[!]_{\text{tot}}\ \frac{\vdash_{\text{wtot}} \{inv\}\ P\ \{inv\} \quad P \text{ is } \#\text{-decreasing under } inv}{\{inv\}\ P!\ \{\text{FE}_{\mathfrak{A}}[inv, P]\}}$$

The proof rules $[\text{if}]_{\text{tot}}$, $[\text{try}]_{\text{tot}}$, and $[!]_{\text{tot}}$ each contain a premise that must be shown to hold in our calculus for weak total correctness. In $[\text{if}]_{\text{tot}}$ and $[\text{try}]_{\text{tot}}$ it is required to show that executions of program C on a graph satisfying the precondition always terminate. However, proving that the execution will not lead to failure is too strong a demand; failure simply informs the branch of the conditional to execute next. Suppose that the premise was instead $\vdash_{\text{tot}} \{c\}\ C\ \{t\}$ with t defining $\top_{\mathfrak{A}}$. Then, for example, for the following program F:

```
if fail then fail else skip
```

we would not be able to prove $\vdash_{\text{tot}} \{a\}\ F\ \{a\}$ for any assertion a. Any such triple is true because F is equivalent to the derived command `skip`: the "guard" program always fails, meaning that `skip` is always executed. But the supposed premise is too strong, requiring us to prove that the guard program never fails.

In $[!]_{\text{tot}}$, we do not want to prove $\vdash_{\text{tot}} \{inv\}\ P\ \{inv\}$ as a premise, since we expect that at some point that P will fail and allow the iteration to terminate.

Finally, what remains to be addressed is the core operation that can explicitly introduce failure: the application of (sets of) rule schemata. We add the following new proof rule to address this:

$$[\text{ruleset}]_{\text{tot}}\ \frac{\text{impl}(c, \text{SE}_{\mathfrak{A}}[c, \mathcal{R}]) \quad \vdash_{\text{par}} \{c\}\ r\ \{d\} \text{ for each } r \in \mathcal{R}}{\{c\}\ \mathcal{R}\ \{d\}}$$

$\text{impl}(a_1, a_2)$: prove that $G \models_{\mathfrak{A}} a_1$ implies $G \models_{\mathfrak{A}} a_2$ for all $G \in \mathcal{G}(\mathcal{L})$

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

We do not include a proof rule for a program that is just a single rule schema r, because this case is captured by proving $\vdash_{\text{tot}} \{c\}\ \{r\}\ \{d\}$.

The total correctness proof rules discussed are given together in Figure A.5 in the appendix. Additionally, in Figure A.6 we present some further proof rules for the derived commands of GP.

3.6 Properties of the Calculi

In this section we consider some important properties of the calculi we have introduced. First, we consider an inherent property of assertion languages able to construct SE and FE relative to all assertions and programs: that the problem of checking whether an assertion is satisfied by a graph becomes undecidable. Then, we follow with technical results about the soundness and completeness of the calculi.

3.6.1 Definability and Decidability

In the proof rules of our calculi we rely upon some assertions expressing the existence of successful and failing executions. One might get the feeling – correctly – that being able to define such assertions for arbitrary programs has consequences in decidability. Consider the model checking problem, i.e. the problem of checking whether a graph satisfies a given assertion:

Model Checking Problem (MCP) for $\mathfrak{A}$	
Input:	graph G, assertion $a \in A$
Question:	does $G \models_{\mathfrak{A}} a$?

The model checking problem is undecidable if one can construct the assertions SE or FE for arbitrary programs (but not a class of restricted ones; see Chapter 4). We sketch a proof of this, by (informally) encoding Turing machines and strings respectively as graph transformation systems and graphs representing configurations, and illustrate how the model checking problem cannot be decidable because it could be used to construct a decider for the halting problem.

Proposition 3.20 (Constructing $\text{SE}_{\mathfrak{A}}$; MCP undecidability)**.** Let $\mathfrak{A} = \langle A, \models_{\mathfrak{A}} \rangle$ be an assertion language. Suppose that there is a transformation $\text{SE}(a, P)$ defining $\text{SE}_{\mathfrak{A}}[a, P]$ for all programs P and all assertions a in A. Then the model checking problem for $\mathfrak{A}$ is undecidable in general. □

Proof sketch. Assume that the model checking problem is decidable, i.e. that there is a decider for the question of whether $G \models_{\mathfrak{A}} a$. We construct a decider for the halting problem using an MCP decider as a component and hence derive a contradiction. For simplicity, we assume that the assertion language defines $\top_{\mathfrak{A}}$, i.e. an assertion satisfied by all graphs.

Let M denote a deterministic Turing machine and w a string composed of M's alphabet. We assume without loss of generality that M does not crash on any input (can always be ensured). Let $\text{enc}(M)$ denote a set of rule schemata simulating the transitions of M (along the lines of [HP01, Plu13]), and $\text{enc}(w)$ denote a graph encoding the initial configuration of M on string w.

We construct a decider for the halting problem in Figure 3.3: here, E_1 and E_2 denote algorithms encoding w and M, and E_3 represents the application of transformation $\text{SE}(\top_{\mathfrak{A}}, \text{enc}(M))$ to get an assertion defining $\text{SE}_{\mathfrak{A}}[\top_{\mathfrak{A}}, \text{enc}(M)]$.

If $\text{enc}(w) \models_{\mathfrak{A}} \text{SE}(\top_{\mathfrak{A}}, \text{enc}(M))$, then by the definition of SE, we have that there exists some graph $G \in [\![\text{enc}(M)]\!]\text{enc}(w)$, corresponding to some accepting configuration of M. Moreover, $[\![\text{enc}(M)]\!]\text{enc}(w) = \{G\}$ since M is deterministic. Hence M halts on w if $\text{enc}(w) \models_{\mathfrak{A}} \text{SE}(\top_{\mathfrak{A}}, \text{enc}(M))$. If the satisfaction relation does not hold then M will loop on w (by assumption it will not crash). Termination of M on w is equivalent to checking $\text{enc}(w) \models_{\mathfrak{A}} \text{SE}(\top_{\mathfrak{A}}, \text{enc}(M))$, so we use the answer of the MCP decider to solve the halting problem for M, w. But the halting problem is not decidable. A contradiction. The model checking problem cannot be decidable if there is a construction defining SE for arbitrary programs.

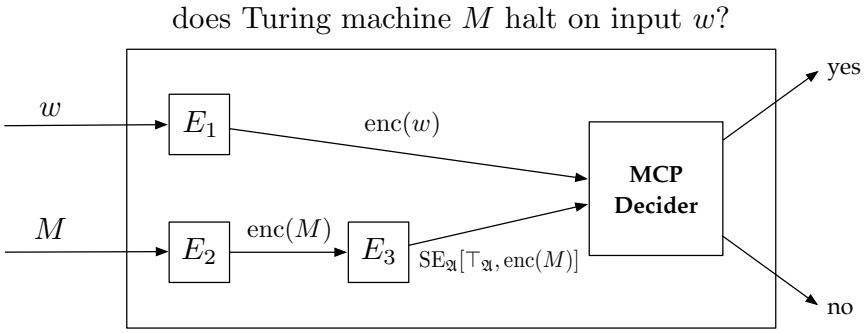


Figure 3.3: Decider for the halting problem

□

Proposition 3.21 (Constructing $\mathrm{FE}_{\mathfrak{A}}$; MCP undecidability)**.** Let $\mathfrak{A} = \langle A, \models_{\mathfrak{A}} \rangle$ be an assertion language. Suppose that there is a transformation $\mathrm{FE}(a, P)$ defining $\mathrm{FE}_{\mathfrak{A}}[a, P]$ for all programs P and all assertions a in A. Then the model checking problem for $\mathfrak{A}$ is undecidable in general. □

Proof sketch. A similar argument to that in the proof of Proposition 3.20 can be used; instead, we replace program enc(M) with enc(M); `fail`. Then, we use the fact that this program will fail on enc(w) if and only if enc(M) halts on enc(w). □

3.6.2 Soundness

Now, we prove that the three proof calculi are sound with respect to the operational semantics of GP. That is, we show that if a triple is derivable in a proof system for partial, weak total, or total correctness, then that triple is valid in the sense of the corresponding notion of correctness.

First, we prove the soundness of the partial correctness proof calculus summarised in Figure A.1.

Theorem 3.22 (Soundness for partial correctness)**.** Given a program P and assertions c, d from an arbitrary assertion language $\mathfrak{A} = \langle A, \models_{\mathfrak{A}} \rangle$,

$$\vdash_{\text{par}} \{c\}\ P\ \{d\} \text{ implies } \models_{\text{par}} \{c\}\ P\ \{d\}.$$

□

Proof. To prove soundness, we show that the implication holds for each of the axioms and inference rules by appealing to the operational semantics of programs (see Section 2.2.5). The result then follows by structural induction on proof trees.

Let r (resp. $\mathcal{R}$) range over rule schemata (resp. sets of rule schemata), c, c', d, d', e, inv over assertions in A, G, G', H, H' over graphs in $\mathcal{G}(\mathcal{L})$, and

C, P, Q over graph programs. Recall that the symbol $\to$ denotes a small-step transition relation on configurations of graphs and programs.

[ruleapp]$_{\text{wlp}}$. Suppose that $\vdash_{\text{par}} \{c\}\ r\ \{d\}$ where c is a suitable assertion for $\text{Wlp}_{\mathfrak{A}}[r, d]$. Suppose that $G \models_{\mathfrak{A}} c$. Then by Definitions 3.9 and 3.10 we have that $H \models_{\mathfrak{A}} d$ for every direct derivation $G \Rightarrow_r H$, i.e. every $H \in [\![r]\!]G$, and the result that $\models_{\text{par}} \{c\}\ r\ \{d\}$.

[ruleset]. Suppose that $\vdash_{\text{par}} \{c\}\ \mathcal{R}\ \{d\}$ and $G \models_{\mathfrak{A}} c$. By induction hypothesis we have $\models_{\text{par}} \{c\}\ r\ \{d\}$ for each $r \in \mathcal{R}$, and by definition of partial correctness we have that $H \models_{\mathfrak{A}} d$ for each $H \in [\![r]\!]G = \{H \in \mathcal{G}(\mathcal{L}) \mid G \Rightarrow_r H\}$ where the equality is clear from the semantic rule $[\text{call}_1]_{\text{OS}}$. By the definition of rule schema set application, it is clear that $[\![\mathcal{R}]\!]G = \bigcup_{r \in \mathcal{R}} [\![r]\!]G$, so every graph H in $[\![\mathcal{R}]\!]G$ must satisfy d. With this we get that result that $\models_{\text{par}} \{c\}\ \mathcal{R}\ \{d\}$.

[comp]. Suppose that $\vdash_{\text{par}} \{c\}\ P;\ Q\ \{d\}$ and $G \models_{\mathfrak{A}} c$. By induction hypothesis we have $\models_{\text{par}} \{c\}\ P\ \{e\}$, $\models_{\text{par}} \{e\}\ Q\ \{d\}$, and together with the semantic rules $[\text{seq}_1]_{\text{OS}}$, $[\text{seq}_2]_{\text{OS}}$ we have that

$$[\![P;\ Q]\!]G - \{\text{fail}\}^2 = \{H \in \mathcal{G}(\mathcal{L}) \mid \langle P;\ Q, G\rangle \to^+ \langle Q, G'\rangle \to^+ H\}$$

where each G' (resp. H) in the sequence of configurations satisfies e (resp. d). It follows that $\models_{\text{par}} \{c\}\ P;\ Q\ \{d\}$.

[if]. Suppose that $\vdash_{\text{par}} \{c\}\ \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}$ and $G \models_{\mathfrak{A}} c$. By induction hypothesis we have $\models_{\text{par}} \{c_s\}\ P\ \{d\}$, $\models_{\text{par}} \{c_f\}\ Q\ \{d\}$ where c_s (resp. c_f) is an assertion defining $\text{SE}_{\mathfrak{A}}[c, C]$ (resp. $\text{FE}_{\mathfrak{A}}[c, C]$). We consider the three possible outcomes of executing C on a graph G and show that soundness holds for all of them.

Case A. There is a successful execution of C, i.e. there is some graph G' in $[\![C]\!]G$, and by Definition 3.11, $G \models_{\mathfrak{A}} c_s$. By the semantic rule $[\text{if}_1]_{\text{OS}}$, P is executed on G. Then by induction hypothesis and definition of partial correctness, we have that every graph $H \in [\![P]\!]G$ satisfies d.

Case B. There is a failing execution of C, i.e. fail $\in [\![C]\!]G$, and by Definition 3.12, $G \models_{\mathfrak{A}} c_f$. By the semantic rule $[\text{if}_2]_{\text{OS}}$, Q is executed on G. Then by induction hypothesis and definition of partial correctness, we have that every graph $H \in [\![Q]\!]G$ satisfies d.

Case C. There is neither a successful nor a failing execution of C. No semantic rules can be applied; the execution is stuck. No graph will result to check against the postcondition d.

Putting all three cases together we obtain our soundness result for [if], i.e. $\models_{\text{par}} \{c\}\ \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}$.

[2] We are not concerned with the element fail in partial correctness.

[try]. Suppose that $\vdash_{\text{par}} \{c\}$ `try` C `then` P `else` Q $\{d\}$ and $G \models_{\mathfrak{A}} c$. By induction hypothesis we have $\models_{\text{par}} \{c_s\}\ C;\ P\ \{d\}$, $\models_{\text{par}} \{c_f\}\ Q\ \{d\}$ where c_s (resp. c_f) is an assertion defining $\text{SE}_{\mathfrak{A}}[c, C]$ (resp. $\text{FE}_{\mathfrak{A}}[c, C]$). We consider the three possible outcomes of executing C on a graph G and show that soundness holds for all of them.

Case A. There is a successful execution of C, i.e. there is some graph G' in $[\![C]\!]G$, and by Definition 3.11, $G \models_{\mathfrak{A}} c_s$. By the semantic rule $[\text{try}_1]_{\text{OS}}$, P is executed on G'. By induction hypothesis and the definition of partial correctness, we have that every graph H in

$$[\![C;\ P]\!]G - \{\text{fail}\} = \{H \in \mathcal{G}(\mathcal{L}) \mid \langle C;\ P, G\rangle \to^+ \langle P, G'\rangle \to^+ H\}$$

satisfies the postcondition d.

Case B. There is a failing execution of C. Case follows as for [if].

Case C. There is neither a successful nor a failing execution of C. Case follows as for [if].

Putting all three cases together we obtain our soundness result for [try], i.e. $\models_{\text{par}} \{c\}$ `try` C `then` P `else` Q $\{d\}$.

[!]. Suppose that $\vdash_{\text{par}} \{inv\}\ P!\ \{inv_f\}$ where inv_f is an assertion defining $\text{FE}_{\mathfrak{A}}[inv, P]$, and suppose that $G \models_{\mathfrak{A}} inv$. Given any graph H in $[\![P!]\!]G$, it must result from a sequence of configurations:

$$\langle P!, G\rangle \to^* H$$

where the sequence contains zero or more applications of $[\text{alap}_1]_{\text{OS}}$ (case zero: $H = G$) followed by a single application of $[\text{alap}_2]_{\text{OS}}$. Sequences of $[\text{alap}_1]_{\text{OS}}$ instances contain configurations $\langle P!, X\rangle$ where X is a graph resulting from applying P once in each instance of the sequence. By induction hypothesis we have $\models_{\text{par}} \{inv\}\ P\ \{inv\}$, and so each graph along the sequence satisfies the invariant inv. The single instance of $[\text{alap}_2]_{\text{OS}}$ does not transform the final graph, H, hence it satisfies the invariant. But the premise of $[\text{alap}_2]_{\text{OS}}$ states that there is a failing execution of P on H – which satisfies inv – so by Definition 3.12 it must be the case that $H \models_{\mathfrak{A}} inv_f$. Hence the result that $\models_{\text{par}} \{inv\}\ P\ \{inv_f\}$.

[cons]. Suppose that $\vdash_{\text{par}} \{c\}\ P\ \{d\}$ and $G \models_{\mathfrak{A}} c$. By induction hypothesis we have $\models_{\text{par}} \{c'\}\ P\ \{d'\}$, $G \models_{\mathfrak{A}} c'$, $H \models_{\mathfrak{A}} d'$, and $H \models_{\mathfrak{A}} d$ for every H in $[\![P]\!]G$. Hence immediately the result that $\models_{\text{par}} \{c\}\ P\ \{d\}$. □

Next, we prove the soundness of the proof calculus for weak total correctness, the proof rules of which are summarised in Figure A.3.

Theorem 3.23 (Soundness for weak total correctness)**.** Given a program P and assertions c, d from an arbitrary assertion language $\mathfrak{A} = \langle A, \models_{\mathfrak{A}} \rangle$,

$$\vdash_{\text{wtot}} \{c\}\ P\ \{d\} \text{ implies } \models_{\text{wtot}} \{c\}\ P\ \{d\}.$$

□

Proof. To prove soundness, we show that the implication holds for each of the axioms and inference rules by appealing to the operational semantics of programs (see Section 2.2.5). The result then follows by structural induction on proof trees.

Let r (resp. $\mathcal{R}$) range over rule schemata (resp. sets of rule schemata), c, c', d, d', e, inv over assertions in A, G, G', H, H' over graphs in $\mathcal{G}(\mathcal{L})$, and C, P, Q over graph programs. Recall that the symbol $\to$ denotes a small-step transition relation on configurations of graphs and programs.

[ruleapp]$_{\text{wlp}}$, [ruleset]. We have the partial correctness of both rules from Theorem 3.22. The semantics of rule schemata application imply weak total correctness in both cases: executions of the programs either transform the graph, or fail, in exactly one transition.

[comp]. Suppose that $\vdash_{\text{wtot}} \{c\}\ P;\ Q\ \{d\}$ and $G \models_{\mathfrak{A}} c$. By induction hypothesis we have $\models_{\text{wtot}} \{c\}\ P\ \{e\}$ and $\models_{\text{wtot}} \{e\}\ Q\ \{d\}$, i.e. the executions of P and Q always terminate with regards to their pre- and postconditions. If the execution of P results in fail, then [seq$_3$]$_{\text{OS}}$ will apply and the execution of $P; Q$ results in fail. Should the execution of P result in a graph H, then by [seq$_2$]$_{\text{OS}}$ program Q will be executed on H and will result in a graph or fail. Together with Theorem 3.22 we get the result that $\models_{\text{wtot}} \{c\}\ P;\ Q\ \{d\}$.

[cons]. The conclusion of the proof rule is clearly weakly totally correct if the premise is: the program in both triples remains the same. The rule simply allows manipulation of the pre- and postconditions, the soundness of which was shown in Theorem 3.22.

[if]$_{\text{wtot}}$. Suppose that $\vdash_{\text{wtot}} \{c\}$ `if` C `then` P `else` $Q\ \{d\}$ and $G \models_{\mathfrak{A}} c$. By induction hypothesis, $\models_{\text{wtot}} \{c\}\ C\ \{t\}$ where t is an assertion defining $\top_{\mathfrak{A}}$. All executions of C on graphs satisfying c result in a graph or fail (i.e. no divergence or getting stuck), resulting respectively in the application of [if$_1$]$_{\text{OS}}$ or [if$_2$]$_{\text{OS}}$, i.e. the execution of P or Q on the original graph G. By induction hypothesis, $\models_{\text{wtot}} \{c_s\}\ P\ \{d\}$ (resp. $\models_{\text{wtot}} \{c_f\}\ Q\ \{d\}$) where c_s (resp. c_f) defines $\text{SE}_{\mathfrak{A}}[c, C]$ (resp. $\text{FE}_{\mathfrak{A}}[c, C]$), i.e. the programs P and Q are guaranteed to terminate when executed on a graph satisfying c. Together, and with Theorem 3.22 we get the result that $\models_{\text{wtot}} \{c\}$ `if` C `then` P `else` $Q\ \{d\}$.

$[\text{try}]_{\text{wtot}}$. Similar to the argument for $[\text{if}]_{\text{wtot}}$.

$[!]_{\text{wtot}}$. Suppose that $\vdash_{\text{wtot}} \{inv\}\ P!\ \{inv_f\}$ where inv_f is an assertion defining $\text{FE}_{\mathfrak{A}}[inv, P]$, and $G \models_{\mathfrak{A}} inv$. By induction hypothesis, we have $\models_{\text{wtot}} \{inv\}\ P\ \{inv\}$, i.e. each application of P maintains the invariant and also terminates. Since every iteration of P will yield a graph or fail state, the execution of the program will not get stuck in some configuration $\langle P!, X\rangle$ for $X \in \mathcal{G}(\mathcal{L})$ since either $[\text{alap}_1]_{\text{OS}}$ or $[\text{alap}_2]_{\text{OS}}$ will always eventually be applicable. What remains to show is that there will be a strictly finite number of iterations of P, i.e. no diverging sequence $\langle P!, G\rangle \to \langle P!, G'\rangle \to \cdots$.

Suppose that there is a termination function $\#$ such that program P is $\#$-decreasing under inv. By Definition 3.16, for all graphs G, H in $\mathcal{G}(\mathcal{L})$ satisfying inv, we have that $H \in [\![P]\!]G$ implies $\#G > \#H$. We show by contradiction that this guarantees a finitely long iteration of P. Assume that $P!$ diverges on any such G. Since P is $\#$-decreasing under inv, each complete execution of P yields a graph for which $\#$ returns a smaller natural number. Since $P!$ diverges, there are infinitely many executions of P, each resulting in a graph for which $\#$ returns a smaller number. But there are only finitely many smaller natural numbers than any natural number n. A contradiction. It cannot be the case that $P!$ diverges from any such G. Together, and with Theorem 3.22, we get the result that $\models_{\text{wtot}} \{inv\}\ P!\ \{inv_f\}$. □

Finally, we prove soundness of our calculus for total correctness. The proof rules of this calculus can be found together in Figure A.5.

Theorem 3.24 (Soundness for total correctness)**.** Given a program P and assertions c, d from an arbitrary assertion language $\mathfrak{A} = \langle A, \models_{\mathfrak{A}}\rangle$,

$$\vdash_{\text{tot}} \{c\}\ P\ \{d\} \text{ implies } \models_{\text{tot}} \{c\}\ P\ \{d\}.$$

□

Proof. To prove soundness, we show that the implication holds for each of the axioms and inference rules by appealing to the operational semantics of programs (see Section 2.2.5). The result then follows by structural induction on proof trees.

Let r (resp. $\mathcal{R}$) range over rule schemata (resp. sets of rule schemata), c, c', d, d', e, inv over assertions in A, G, G', H, H' over graphs in $\mathcal{G}(\mathcal{L})$, and C, P, Q over graph programs. Recall that the symbol $\to$ denotes a small-step transition relation on configurations of graphs and programs.

[comp]. Suppose that $\vdash_{\text{tot}} \{c\}\ P;\ Q\ \{d\}$ and $G \models_{\mathfrak{A}} c$. By induction hypothesis we have $\models_{\text{tot}} \{c\}\ P\ \{e\}$ and $\models_{\text{tot}} \{e\}\ Q\ \{d\}$. By the definition of total correctness and Theorem 3.23, we have $\models_{\text{wtot}} \{c\}\ P;\ Q\ \{d\}$, which we strengthen to $\models_{\text{tot}} \{c\}\ P;\ Q\ \{d\}$ since the semantic rule $[\text{seq}_3]_{\text{OS}}$ cannot

be applied (by induction hypothesis there is no possibility that $\langle P, G\rangle \to^+$ fail), and by induction hypothesis Q will not fail on graphs satisfying e.

[cons]. The conclusion of the proof rule is clearly totally correct if the premise is: the program in both triples remains the same. The rule simply allows manipulation of the pre- and postconditions, the soundness of which was shown in Theorem 3.22.

[ruleset]$_{\text{tot}}$. Suppose that $\vdash_{\text{tot}} \{c\}\ \mathcal{R}\ \{d\}$ and $G \models_{\mathfrak{A}} c$. By induction hypothesis and Theorem 3.22 we have that $\models_{\text{par}} \{c\}\ r\ \{d\}$ for each $r \in \mathcal{R}$, which gives us $\models_{\text{par}} \{c\}\ \mathcal{R}\ \{d\}$. Also by induction hypothesis, we have that $G \models_{\mathfrak{A}} c_s$ where c_s is an assertion defining $\text{SE}_{\mathfrak{A}}[c, \mathcal{R}]$. By Definition 3.11, there is some $H \in [\![\mathcal{R}]\!]G$, and by the semantic rule $[\text{call}_1]_{\text{OS}}$ it must be the case that $G \Rightarrow_{\mathcal{R}} H$. This contradicts the premise of $[\text{call}_2]_{\text{OS}}$, meaning that the semantic rule cannot be applied and fail $\notin [\![\mathcal{R}]\!]G$. Together, and with the observation that $\mathcal{R}$ cannot diverge or get stuck on any graph, we get the result that $\models_{\text{tot}} \{c\}\ \mathcal{R}\ \{d\}$.

[if]$_{\text{tot}}$. Suppose that $\vdash_{\text{tot}} \{c\}\ \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}$ and $G \models_{\mathfrak{A}} c$. By induction hypothesis we have $\models_{\text{wtot}} \{c\}\ C\ \{t\}$, $\models_{\text{tot}} \{c_s\}\ P\ \{d\}$, and $\models_{\text{tot}} \{c_f\}\ Q\ \{d\}$ where t, c_s, c_f respectively define $\top_{\mathfrak{A}}, \text{SE}_{\mathfrak{A}}[c, C], \text{FE}_{\mathfrak{A}}[c, C]$. By Theorem 3.23 we have that $\models_{\text{wtot}} \{c\}\ \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}$, which we strengthen to $\models_{\text{tot}}$ by the semantics of if-then-else and the hypothesis that P and Q do not fail – and are weakly totally correct – on graphs respectively satisfying c_s, c_f.

[try]$_{\text{tot}}$. The argument is similar to that for [if]$_{\text{tot}}$.

[!]$_{\text{tot}}$. Suppose that $\vdash_{\text{tot}} \{inv\}\ P!\ \{inv_f\}$ where inv_f defines $\text{FE}_{\mathfrak{A}}[inv, P]$, and $G \models_{\mathfrak{A}} inv$. By induction hypothesis we have $\models_{\text{wtot}} \{inv\}\ P\ \{inv\}$ and that P is $\#$-decreasing under inv. Together with Theorem 3.23, we have that $\models_{\text{wtot}} \{inv\}\ P!\ \{inv_f\}$. From the semantic rules $[\text{alap}_1]_{\text{OS}}$ and $[\text{alap}_2]_{\text{OS}}$, there is no possibility of an execution $\langle P!, G\rangle \to^*$ fail. Hence the result that $\models_{\text{tot}} \{inv\}\ P!\ \{inv_f\}$. □

3.6.3 Completeness

In this subsection, we consider two important completeness properties. First, we consider the completeness of the partial correctness rules, i.e. if a triple is valid, can one prove it in our calculus? Next, we consider the issue of termination, and prove that if a program under as-long-as-possible iteration will terminate, then a termination function exists that we can use to prove it.

Relative Completeness

In Section 3.1 we discussed the notion of completeness, and in particular the notion of relative completeness, which treats incompleteness due to the assertion language as a separate issue. Here, we prove the relative completeness of our partial correctness calculus in a similar style to that of Winskel [Win93].

The proof relies on weakest liberal preconditions relative to commands and postconditions, the idea being that stronger liberal preconditions can be proven simply by an application of [cons]. This requires that the assertion language we are using is *expressive*, i.e. able to define weakest liberal preconditions for arbitrary combinations of programs and postconditions.

Definition 3.25 (Expressive assertion languages)**.** An assertion language $\mathfrak{A} = \langle A, \models_{\mathfrak{A}} \rangle$ is *expressive* (resp. *expressive for* $\mathcal{P}$) if for every program P (resp. from a restricted class of programs $\mathcal{P}$) and assertion a in A, there is a weakest liberal precondition $\mathsf{Wlp}_{\mathfrak{A}}[P, a]$ in A. □

For our extensional calculi, we simply assume that the assertion language is expressive. If an assertion language is fixed, the relative completeness proof for our calculi can still be used, but in addition one must also prove that the assertion language is expressive (the biggest challenge is to show that it can express the weakest liberal precondition relative to an iterated program and postcondition).

Lemma 3.26 (Provability of Wlp)**.** Let $\mathfrak{A} = \langle A, \models_{\mathfrak{A}} \rangle$ denote an expressive assertion language, P a program, and a an assertion in A. Then,

$$\vdash_{\text{par}} \{\mathsf{Wlp}_{\mathfrak{A}}[P, c]\}\ P\ \{c\}.$$

□

Proof. We proceed by structural induction on programs, showing that the lemma holds for all commands. Let r range over conditional rule schemata, $\mathcal{R}$ over sets of conditional rule schemata, and C, P_1, P_2 over programs.

Case $P = r$. We get $\vdash_{\text{par}} \{\mathsf{Wlp}_{\mathfrak{A}}[r, c]\}\ r\ \{c\}$ directly from the axiom $[\text{ruleapp}]_{\text{wlp}}$.

Case $P = \mathcal{R}$. If $\mathcal{R} = \emptyset$, then [ruleset] has no premises to prove and so we axiomatically deduce $\vdash_{\text{par}} \{\mathsf{Wlp}_{\mathfrak{A}}[\mathcal{R}, c]\}\ \mathcal{R}\ \{c\}$. If $\mathcal{R} \neq \emptyset$, then:

$$\begin{aligned} & G \models_{\mathfrak{A}} \mathsf{Wlp}_{\mathfrak{A}}[\mathcal{R}, c] \\ \text{iff}\ \ & H \models_{\mathfrak{A}} c \text{ for all } G \Rightarrow_{\mathcal{R}} H \\ \text{iff}\ \ & H \models_{\mathfrak{A}} c \text{ for all } G \Rightarrow_{r} H \text{ and } r \in \mathcal{R} \\ \text{iff}\ \ & G \models_{\mathfrak{A}} \mathsf{Wlp}_{\mathfrak{A}}[r, c] \text{ for all } r \in \mathcal{R} \end{aligned}$$

By the induction hypothesis we have $\vdash_{\text{par}} \{\text{Wlp}_{\mathfrak{A}}[r, c]\}\ r\ \{c\}$ for all $r \in \mathcal{R}$. With [cons] we deduce from each of these $\vdash_{\text{par}} \{\text{Wlp}_{\mathfrak{A}}[\mathcal{R}, c]\}\ r\ \{c\}$, and then finally, with the rule [ruleset], we deduce the result:

$$\vdash_{\text{par}} \{\text{Wlp}_{\mathfrak{A}}[\mathcal{R}, c]\}\ \mathcal{R}\ \{c\}.$$

Case $P = P_1; P_2$. In this case,

$$\begin{aligned}
& G \models_{\mathfrak{A}} \text{Wlp}_{\mathfrak{A}}[P_1; P_2, c] \\
\text{iff}\ \ & H \models_{\mathfrak{A}} c \text{ for all } H \in [\![P_1; P_2]\!]G \\
\text{iff}\ \ & H \models_{\mathfrak{A}} c \text{ for all } H \in [\![P_2]\!]G' \text{ and all } G' \in [\![P_1]\!]G \\
\text{iff}\ \ & G' \models_{\mathfrak{A}} \text{Wlp}_{\mathfrak{A}}[P_2, c] \text{ for all } G' \in [\![P_1]\!]G \\
\text{iff}\ \ & G \models_{\mathfrak{A}} \text{Wlp}_{\mathfrak{A}}[P_1, \text{Wlp}_{\mathfrak{A}}[P_2, c]]
\end{aligned}$$

By the induction hypothesis we have:

$$\vdash_{\text{par}} \{\text{Wlp}_{\mathfrak{A}}[P_1, \text{Wlp}_{\mathfrak{A}}[P_2, c]]\}\ P_1\ \{\text{Wlp}_{\mathfrak{A}}[P_2, c]\}$$

and:

$$\vdash_{\text{par}} \{\text{Wlp}_{\mathfrak{A}}[P_2, c]\}\ P_2\ \{c\}.$$

By [comp] we get:

$$\vdash_{\text{par}} \{\text{Wlp}_{\mathfrak{A}}[P_1, \text{Wlp}_{\mathfrak{A}}[P_2, c]]\}\ P_1; P_2\ \{c\}$$

and then by [cons] the result that:

$$\vdash_{\text{par}} \{\text{Wlp}_{\mathfrak{A}}[P_1; P_2, c]\}\ P_1; P_2\ \{c\}.$$

Case $P = \texttt{if}\ C\ \texttt{then}\ P_1\ \texttt{else}\ P_2$. In this case,

$$\begin{aligned}
& G \models_{\mathfrak{A}} \text{Wlp}_{\mathfrak{A}}[\texttt{if}\ C\ \texttt{then}\ P_1\ \texttt{else}\ P_2, c] \\
\text{iff}\ \ & (\text{some } G' \in [\![C]\!]G \text{ implies } H \models_{\mathfrak{A}} c \text{ for all } H \in [\![P_1]\!]G \\
& \text{and fail} \in [\![C]\!]G \text{ implies } H \models_{\mathfrak{A}} c \text{ for all } H \in [\![P_2]\!]G) \\
\text{iff}\ \ & (G \models_{\mathfrak{A}} \text{SE}_{\mathfrak{A}}[\texttt{true}, C] \text{ implies } G \models_{\mathfrak{A}} \text{Wlp}_{\mathfrak{A}}[P_1, c] \\
& \text{and } G \models_{\mathfrak{A}} \text{FE}_{\mathfrak{A}}[\texttt{true}, C] \text{ implies } G \models_{\mathfrak{A}} \text{Wlp}_{\mathfrak{A}}[P_2, c])
\end{aligned}$$

By the induction hypothesis we have $\vdash_{\text{par}} \{\text{Wlp}_{\mathfrak{A}}[P_1, c]\}\ P_1\ \{c\}$ and $\vdash_{\text{par}} \{\text{Wlp}_{\mathfrak{A}}[P_2, c]\}\ P_2\ \{c\}$. Applying [cons] and the definitions of SE, FE we get:

$$\vdash_{\text{par}} \{\text{SE}_{\mathfrak{A}}[\text{Wlp}_{\mathfrak{A}}[\texttt{if}\ C\ \texttt{then}\ P_1\ \texttt{else}\ P_2, c], C]\}\ P_1\ \{c\}$$

and:

$$\vdash_{\text{par}} \{\text{FE}_{\mathfrak{A}}[\text{Wlp}_{\mathfrak{A}}[\texttt{if } C \texttt{ then } P_1 \texttt{ else } P_2, c], C]\}\ P_2\ \{c\}.$$

By [if] we get the result that:

$$\vdash_{\text{par}} \{\text{Wlp}_{\mathfrak{A}}[\texttt{if } C \texttt{ then } P_1 \texttt{ else } P_2, c]\}\ \texttt{if } C \texttt{ then } P_1 \texttt{ else } P_2\ \{c\}.$$

Case $P = \texttt{try } C \texttt{ then } P_1 \texttt{ else } P_2$. In this case,

$$\begin{aligned}
& G \models_{\mathfrak{A}} \text{Wlp}_{\mathfrak{A}}[\texttt{try } C \texttt{ then } P_1 \texttt{ else } P_2, c] \\
\text{iff } & (\text{some } G' \in [\![C]\!]G \text{ implies } H \models_{\mathfrak{A}} c \text{ for all } H \in [\![P_1]\!]G' \\
& \text{and fail} \in [\![C]\!]G \text{ implies } H \models_{\mathfrak{A}} c \text{ for all } H \in [\![P_2]\!]G) \\
\text{iff } & (\text{some } G' \in [\![C]\!]G \text{ implies } G' \models_{\mathfrak{A}} \text{Wlp}_{\mathfrak{A}}[P_1, c] \\
& \text{and fail} \in [\![C]\!]G \text{ implies } H \models_{\mathfrak{A}} c \text{ for all } H \in [\![P_2]\!]G) \\
\text{iff } & (G \models_{\mathfrak{A}} \text{SE}_{\mathfrak{A}}[\texttt{true}, C] \text{ implies } G \models_{\mathfrak{A}} \text{Wlp}_{\mathfrak{A}}[C, \text{Wlp}_{\mathfrak{A}}[P_1, c]] \\
& \text{and } G \models_{\mathfrak{A}} \text{FE}_{\mathfrak{A}}[\texttt{true}, C] \text{ implies } G \models_{\mathfrak{A}} \text{Wlp}_{\mathfrak{A}}[P_2, c])
\end{aligned}$$

By the induction hypothesis and with [comp] we get:

$$\vdash_{\text{par}} \{\text{Wlp}_{\mathfrak{A}}[C, \text{Wlp}_{\mathfrak{A}}[P_1, c]]\}\ C;\ P_1\ \{c\}$$

and:

$$\vdash_{\text{par}} \{\text{Wlp}_{\mathfrak{A}}[P_2, c]\}\ P_2\ \{c\}.$$

Applying [cons] and the definitions of SE, FE we get:

$$\vdash_{\text{par}} \{\text{SE}_{\mathfrak{A}}[\text{Wlp}_{\mathfrak{A}}[\texttt{try } C \texttt{ then } P_1 \texttt{ else } P_2, c], C]\}\ C;\ P_1\ \{c\}$$

and:

$$\vdash_{\text{par}} \{\text{FE}_{\mathfrak{A}}[\text{Wlp}_{\mathfrak{A}}[\texttt{try } C \texttt{ then } P_1 \texttt{ else } P_2, c], C]\}\ P_2\ \{c\}.$$

With [try] we get the result that:

$$\vdash_{\text{par}} \{\text{Wlp}_{\mathfrak{A}}[\texttt{try } C \texttt{ then } P_1 \texttt{ else } P_2, c]\}\ \texttt{try } C \texttt{ then } P_1 \texttt{ else } P_2\ \{c\}.$$

Case $P = P_1!$. The induction hypothesis gives us:

$$\vdash_{\text{par}} \{\text{Wlp}_{\mathfrak{A}}[P_1, \text{Wlp}_{\mathfrak{A}}[P_1!, c]]\}\ P_1\ \{\text{Wlp}_{\mathfrak{A}}[P_1!, c]\}.$$

With the following reasoning,

$$\begin{aligned} & G \models_{\mathfrak{A}} \mathrm{Wlp}_{\mathfrak{A}}[P_1!, c] \\ \text{implies } & H \models_{\mathfrak{A}} c \text{ for all } H \in [\![P_1!]\!]G \\ \text{implies } & H \models_{\mathfrak{A}} c \text{ for all } H \in [\![P_1!]\!]G' \text{ and } G' \in [\![P_1]\!]G \\ \text{implies } & G' \models_{\mathfrak{A}} \mathrm{Wlp}_{\mathfrak{A}}[P_1!, c] \text{ for all } G' \in [\![P_1]\!]G \\ \text{implies } & G \models_{\mathfrak{A}} \mathrm{Wlp}_{\mathfrak{A}}[P_1, \mathrm{Wlp}_{\mathfrak{A}}[P_1!, c]] \end{aligned}$$

and an application of [cons], we get:

$$\vdash_{\text{par}} \{\mathrm{Wlp}_{\mathfrak{A}}[P_1!, c]\}\ P_1\ \{\mathrm{Wlp}_{\mathfrak{A}}[P_1!, c]\}.$$

Applying [!] we get:

$$\vdash_{\text{par}} \{\mathrm{Wlp}_{\mathfrak{A}}[P_1!, c]\}\ P_1!\ \{\mathrm{FE}_{\mathfrak{A}}[\mathrm{Wlp}_{\mathfrak{A}}[P_1!, c], P_1]\}.$$

We need to show that satisfying $\mathrm{FE}_{\mathfrak{A}}[\mathrm{Wlp}_{\mathfrak{A}}[P_1!, c], P_1]$ implies the satisfaction of c. Suppose that $G \models_{\mathfrak{A}} \mathrm{FE}_{\mathfrak{A}}[\mathrm{Wlp}_{\mathfrak{A}}[P_1!, c], P_1]$. Then by definition of FE, fail $\in [\![P_1]\!]G$ and $G \models_{\mathfrak{A}} \mathrm{Wlp}_{\mathfrak{A}}[P_1!, c]$. Then $G \models_{\mathfrak{A}} c$ because $G \in [\![P_1!]\!]G$ and for all $H \in [\![P_1!]\!]G$, $H \models_{\mathfrak{A}} c$. With [cons] we derive the result:

$$\vdash_{\text{par}} \{\mathrm{Wlp}_{\mathfrak{A}}[P_1!, c]\}\ P_1!\ \{c\}.$$

With all cases considered, the lemma is proven by structural induction. □

The main theorem – that our calculus is relatively complete – follows easily from this lemma. We simply use the proof rule [cons] and the definition of weakest liberal preconditions.

Theorem 3.27 (Relative completeness)**.** Let $\mathfrak{A} = \langle A, \models_{\mathfrak{A}} \rangle$ denote an expressive assertion language, P a program, and c, d assertions in A. Then,

$$\models_{\text{par}} \{c\}\ P\ \{d\} \text{ implies } \vdash_{\text{par}} \{c\}\ P\ \{d\}.$$

□

Proof. Suppose that $\models_{\text{par}} \{c\}\ P\ \{d\}$. By Lemma 3.26 we have:

$$\vdash_{\text{par}} \{\mathrm{Wlp}_{\mathfrak{A}}[P, d]\}\ P\ \{d\}.$$

By the definition of weakest liberal preconditions, for every graph $G \in \mathcal{G}(\mathcal{L})$, $G \models_{\mathfrak{A}} c$ implies that $G \models_{\mathfrak{A}} \mathrm{Wlp}_{\mathfrak{A}}[P, d]$. Applying [cons] then gives us the result:

$$\vdash_{\text{par}} \{c\}\ P\ \{d\}.$$

□

Completeness for Termination

In the proof rule handling as-long-possible-iteration for (weak) total correctness, the user is required to devise a termination function outside of the calculus to show that a program will eventually terminate. Previously we have shown this to be a sound method for proving (weak) total correctness. Now, we go a step further and prove that regardless of the program being iterated, there is always a suitable termination function that can be defined for that program. We refer to this property as *completeness for termination*. First however, we prove a lemma about finiteness that we require in the proof.

Lemma 3.28 (wtot implies finiteness)**.** Let P be a program and c, d be arbitrary assertions from some language $\mathfrak{A} = \langle A, \models_{\mathfrak{A}} \rangle$. Then for all graphs $G \in \mathcal{G}(\mathcal{L})$,

$$\models_{\text{wtot}} \{c\}\ P\ \{d\} \text{ implies } [\![P]\!]G \text{ is finite up to isomorphism.}$$

□

Proof. Assume that $\models_{\text{wtot}} \{c\}\ P\ \{d\}$, i.e. P does not diverge or get stuck on graphs satisfying c. We show by induction on the structure of P that the set of possible graphs (and possibly fail) resulting from executing P on such a graph is always finite (up to isomorphism).

Induction basis. Suppose that $P = \mathcal{R}$ for some set of rule schemata $\mathcal{R}$. Then by the semantics of GP, $[\![P]\!]G = \{H \in \mathcal{G}(\mathcal{L}) \mid G \Rightarrow_{\mathcal{R}} H\}$. There are finitely many rule schemata r in $\mathcal{R}$, and finitely many matches g. Each application $G \Rightarrow_{r,g} H$ is unique up to isomorphism. Together, $[\![P]\!]G$ is finite up to isomorphism.

Induction step. Suppose that $P = Q; S$ where Q, S are programs. Applying the induction hypothesis for Q, S, $[\![Q]\!]G$ is finite, and for each graph G' in that set $[\![S]\!]G'$ is finite. By the semantics of sequential composition, clearly $[\![P]\!]G$ must also be finite up to isomorphism.

Suppose that $P = \texttt{if}\ C\ \texttt{then}\ Q\ \texttt{else}\ S$ where C, Q, S are programs. By assumption, C must terminate on G. The execution of the program is then that of either Q or S, and $[\![Q]\!]G$ (resp. $[\![S]\!]G$) is finite by induction hypothesis. By the semantics of if-then-else, $[\![P]\!]G$ must also be finite up to isomorphism. Similar argument for case $P = \texttt{try}\ C\ \texttt{then}\ Q\ \texttt{else}\ S$.

Finally, suppose that $P = Q!$ where Q is some graph program. By assumption, there cannot be an infinite sequence of iterations of Q. But also, by induction hypothesis, $[\![Q]\!]G'$ for any graph $G' \in \mathcal{G}(\mathcal{L})$ is finite. Figure 3.4 summarises sequences of executions via instances of $[\text{alap}_1]_{\text{OS}}$. The depth is finite by the assumption, and the degrees of all nodes are finite by the

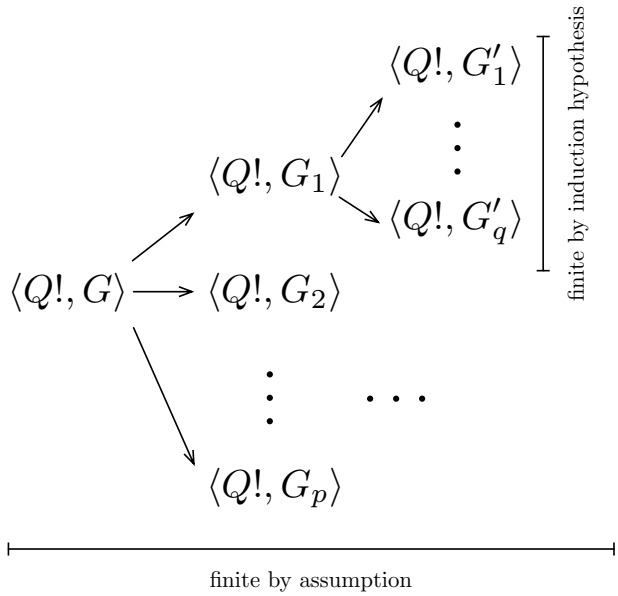


Figure 3.4: Execution tree for as-long-as-possible iteration

induction hypothesis. At the end of sequences, $[\text{alap}_2]_{\text{OS}}$ is applied terminating the iteration. Hence, $[\![P]\!]G$ is finite up to isomorphism, the result. □

With this lemma proven we are now equipped to prove completeness for termination[3]. Intuitively, we show that one can always define a termination function that at a point of the execution, returns the length of the longest sequence of configurations (that do not exit the iteration) from that point of the execution.

Theorem 3.29 (Completeness of $[!]_{\text{tot}}$ for termination)**.** Let P be a program inv an assertion from some assertion language $\mathfrak{A} = \langle A, \models_{\mathfrak{A}} \rangle$. If $\models_{\text{wtot}} \{inv\}\ P\ \{inv\}$ and $\models_{\text{wtot}} \{inv\}\ P!\ \{inv\}$, then there exists a termination function $\#$ such that P is $\#$-decreasing under inv. □

Proof. Let $G \in \mathcal{G}(\mathcal{L})$ be a graph such that $G \models_{\mathfrak{A}} inv$. By assumption that $\models_{\text{wtot}} \{inv\}\ P!\ \{inv\}$, $P!$ will not diverge or get stuck on G. Hence there must be a finite sequence of derivations:

$$\langle P!, G_0 \rangle \to \langle P!, G_1 \rangle \to \langle P!, G_2 \rangle \to \cdots \to G_n$$

via applications of $[\text{alap}_1]_{\text{OS}}$, $[\text{alap}_2]_{\text{OS}}$ where $G_0 = G$ and G_i denotes a graph resulting from the ith iteration of P. To define the termination

[3]Detlef Plump wrote the original version of this proof in [PP13], restricted to sets of conditional rule schemata. We generalise it to arbitrary programs.

function $\#$, we show that the length of such sequences starting from G is bounded. (Note that, in general, a terminating relation need not be bounded.)

For $M, N \in \mathcal{G}(\mathcal{L})$, let $M \rightsquigarrow_P N$ denote a transition $\langle P!, M\rangle \to \langle P!, N\rangle$ via $[\text{alap}_1]_{\text{OS}}$. We exploit that $\rightsquigarrow_P$ is closed under isomorphism in the following sense: given graphs M, M', N and N' such that $M \cong M'$ and $N \cong N'$, then $M \rightsquigarrow_P N$ implies $M' \rightsquigarrow_P N'$. Hence we can lift $\rightsquigarrow_P$ to a relation on isomorphism classes of graphs by defining: $[M] \rightsquigarrow_P [N]$ if $M \rightsquigarrow_P N$. For every isomorphism class $[M]$ the set $\{[N] \mid [M] \rightsquigarrow_P [N]\}$ is finite. To see this, observe that $\models_{\text{wtot}} \{inv\}\ P\ \{inv\}$ implies the finiteness (up to isomorphism) of $[\![P]\!]M$ (Lemma 3.28).

Now, since there is no infinite sequence of $\rightsquigarrow_P$-steps starting from $[G]$, it follows from König's lemma [Kön36] that the length of $\rightsquigarrow_P$-transitions starting from $[G]$ is bounded. (In the tree of all derivations starting from $[G]$, all nodes have a finite degree. Hence the tree cannot be infinite, as otherwise it would contain an infinite derivation.) Hence the length of $\rightsquigarrow_P$-derivations starting from G is bounded as well. In general, given any graph M in $\mathcal{G}(\mathcal{L})$, let $\#M$ be the length of a longest $\rightsquigarrow_P$-sequence starting from M if $M \models_{\mathfrak{A}} inv$, and $\#M = 0$ otherwise. Then if $M, N \models_{\mathfrak{A}} inv$ and $M \rightsquigarrow_P N$, we have $\#M > \#N$. Thus P is $\#$-decreasing under inv. □

Since a termination function can be defined for every iterated program that terminates, we can also use the non-existence of a termination function to prove that a program diverges.

Example 3.30. Suppose that we want to prove $\models_{\text{wtot}} \{inv\}\ (P!)!\ \{inv\}$ for some program P and assertion inv in $\mathfrak{A}$. Suppose also that we have $\models_{\text{wtot}} \{inv\}\ P!\ \{inv\}$, i.e. the "inner" program is known to terminate on graphs satisfying inv. To get the same result for $(P!)!$, we can use $[!]_{\text{wtot}}$ and define a termination function for which the program is decreasing.

Suppose that $\#$ is a termination function for which $P!$ is $\#$-decreasing, i.e. for all graphs $G, H \in \mathcal{G}(\mathcal{L})$ with $H \in [\![P!]\!]G$, $\#G > \#H$. By assumption, $P!$ is terminating under inv, so there is a sequence of configurations

$$\langle P!, G\rangle \to \cdots \to \langle P!, H\rangle \to H$$

with the final transition being an instance of $[\text{alap}_2]_{\text{OS}}$. The premise of this rule must have been satisfied, i.e. there is a sequence of configurations $\langle P, H\rangle \to^+ \text{fail}$. Hence the same graph H is in the set $[\![P!]\!]H$ since the transition

$$\langle P!, H\rangle \to H$$

is clearly possible again by $[\text{alap}_2]_{\text{OS}}$. A contradiction. A termination function $\#$ cannot exist since there are executions of $P!$ returning graphs with the same $\#$-number as the input graph. By Theorem 3.29 the triple $\models_{\text{wtot}} \{inv\}\ (P!)!\ \{inv\}$ does not hold. □

3.7 Related Work

In this section we provide some pointers to related work. First, we provide some historical references for Hoare logic and relative completeness. Following this we shall point to some recent – and quite contrasting – verification approaches applied to graph transformation.

3.7.1 Hoare Logic

The original ideas behind Hoare logic were first published by Floyd [Flo67] who considered programs as flowcharts. The approach was then adapted for program texts in Hoare's seminal paper [Hoa69], and has been developed by numerous researchers in the decades since. Developments in Hoare logic during the first decade were surveyed by Apt [Apt81, Apt84], and several more recent textbooks cover the subject quite thoroughly (e.g. [AdO09, Win93, NN07]). In the last decade an extension of Hoare logic for local reasoning – known as *separation logic* [Rey02] – has seen much success in the verification of programs that manipulate shared mutable state (e.g. a heap). Many authors have also considered extensions of Hoare logic for reasoning about object-oriented programs; of these, a particularly relevant and recent paper [ZWL13] proposed a graph-based extension of Hoare logic for this purpose. Here, program states are visualised as directed labelled graphs which characterise properties of interest in the object-oriented paradigm, e.g. aliasing and reachability. The model is tailored however to reasoning about such properties, whereas GP is much more general.

The separation of weak total correctness (termination) and total correctness (termination and absence of failing executions) follows the approach of Apt in the second part of his survey [Apt84] (although Apt does not describe weak total correctness extensively). The thesis of Pennemann [Pen09] describes slightly different notions of correctness: weak partial (as in our partial), strong partial (existence of results guaranteed but not termination), and total (as in our total). We prefer to leave guarantees of results to the strongest notion of correctness because it leaves the option open for programmers to use failure as a valid and intended result.

The idea of relative completeness was first described by Cook [Coo78]. He devised a Hoare logic for an ALGOL-like language, and proved its completeness relative to being able decide the validity of arithmetic assertions. A later paper by Clarke [Cla85] again considered completeness, showing some negative results: that there are (implemented) constructs (e.g. procedures as parameters of procedure calls) that make it an impossibility to devise a sound and relatively complete Hoare logic for the language. Several textbooks (e.g. [Win93]) cover relative completeness quite thoroughly (indeed our relative completeness proof is given in a similar style).

Weakest preconditions, as introduced by Dijkstra [Dij75, Dij76] for his Guarded Command Language, are a key notion in proving relative completeness. They, along with other predicate transformers such as strongest postconditions, can be used for verification in their own right, e.g. for a given specification compute the weakest precondition relative to a program and postcondition, and then try to deduce (ideally automatically) whether the original precondition implies the weakest precondition.

3.7.2 Verifying Graph Transformations

We (to the best of our knowledge) were the first to use Hoare logic in the verification of a language based on graph transformation [PP10a, PP12], but our original papers only considered partial correctness, did not consider relative completeness, and did not present the calculi in an extensional style. We extended this work in [PP13] to allow (weak) total correctness proofs, requiring users to devise termination functions for programs under as-long-as-possible iteration. Termination of graph grammars in general is known to be undecidable [Plu98], but in special cases can be shown to hold. For example, [EEdL$^+$05] describes termination criteria for model transformations expressed as graph transformations, and [VVGE$^+$06] – again for model transformations expressed as graph transformations – gives a termination analysis technique using Petri net abstractions that either proves termination, or returns a "does not know" answer.

The closest related work to our own is that of Habel, Pennemann, and Rensink, who laid the foundations for assertional reasoning about programs based on graph transformation. They contributed weakest precondition calculi for proving the correctness of programs with respect to first-order structural properties. We give special attention to this work, and compare it with our own approach in Section 4.6.

Other than the aforementioned work, most verification approaches in graph transformation have focused on graph grammars, i.e. sets of rules iteratively executed on some start graph, rather than programming languages like the one considered in this thesis. In the rest of this section we will give an overview of these approaches with some pointers into the literature.

One approach is to logically infer the correctness of graph transformation systems, translating away from graphical rules and formalisms, working instead with a logical characterisation such as that of [Cou90]. Along these lines, Strecker [Str08] models graph transformation rules with (a fragment of) first-order logic over graph structure in the interactive proof assistant Isabelle. While Isabelle can assist with some automation, this is otherwise a manual task, which may be more challenging for users given the translation away from graphical abstractions. Da Costa and Ribeiro [dCR12] propose to define graph grammars using relational structures, and

to use first-order logic to model rule applications. Their approach is equivalent to the single-pushout framework (although they claim the work can be adapted for others), and is intended to provide a way for theorem provers to be used with work that relies on such algebraic frameworks. Tran and Percebois [TP12] are using this approach to verify invariants of graph grammars, using Isabelle for the implementation. In a very recent paper from these authors and Strecker [PST13], it is shown how one can verify global invariants involving paths, directly at the rule level; again with Isabelle modelling the transformations. The approach is sound, but incomplete; rather focusing on the preservation of reachability and separation properties.

An approach quite in contrast to theorem proving is model checking; this involving the exhaustive and automatic search of a state space in order to verify some specification. In the context of graph transformation, such states are (encodings of) graphs, and transitions between them must correspond to the application of rules. Two early and distinct approaches to model checking graph transformation systems – CheckVML and GROOVE – are compared in [RSV04].

The idea of CheckVML [SV03] is to exploit off-the-shelf model checking technology. More specifically, a graph transformation system is encoded in Promela, which is then submitted to the SPIN model checking tool for formal analysis. Whilst benefiting from an existing tool that has been the subject of several years of research, by translating away from the core concepts of graphs and graph transformation systems, it becomes difficult to suitably handle issues like symmetries in graphs, and counterexamples are reported at the abstraction level of SPIN and not translated back into graphs.

GROOVE [Ren03, GdMR+12] instead works with graphs and graph transformation throughout the entire model checking process. Safety, reachability, and danger properties are specified by a graph-based logic, and model checking algorithms are tailored to graphs. While the underlying model of GROOVE allows for existing graph transformation theory to be directly applied, a consequence is that little of the theory and tool development from traditional model checking can be used without significant reworking.

Baresi and Spoletini [BS06] explored the use of Alloy [Jac12] in analysing graph transformation systems. Alloy is a structural modelling language, based on first-order logic and the notion of relation. The Alloy Analyzer tool takes a system expressed in this language and performs finite-scope checks for verification and example generation. Baresi and Spoletini described how to render a graph transformation system (in AGG) as an Alloy model, and showed how the tool could check e.g. reachability properties, and applicability of sequences of graph transformation rules. The approach however only analyses the systems for a finite scope, relying on Jackson's so-called "small scope hypothesis" (that flaws in models are often

revealed in relatively small scopes [Jac12]). The authors (with others) have also looked at transforming AGG specifications into models for the Bogor model checker, with support for both attributed typed graphs and layered graph transformation systems [BRRS08].

The model checking problem becomes undecidable when we want to verify graph transformation systems that have infinite states spaces. In order to facilitate automatic verification, approximation techniques are required to allow us to work with a more abstract view of the system (there is the possibility however of introducing "spurious" counterexamples in that they do not appear in the original system). A well known approach by König et al., introduced in [BCK01, BK02] and further elaborated in [BCK08], approximates the behaviour of graph transformation systems using finite Petri net-like structures obtained from an approximated unfolding construction. If the abstraction introduces a counterexample that does not exist in the system, the abstraction can be incrementally refined to remove it [KK06]. The techniques have been implemented in the tool Augur 2 for (attributed) graph transformation systems [KK08]. Another approach is the framework of neighbourhood abstraction [BBKR08] (unifying independent work by Rensink, Distefano, and Bauer [RD06, Bau06]), which is based on neighbourhood similarity. An abstract graph is obtained by folding "neighbourhood equivalent" nodes (roughly, nodes with the same labels and degrees) into one, keeping count of the original number up to a given bound of precision. This framework has been implemented in the model checker GROOVE [RZ10, ZR12]. A more recent approach, pattern abstraction [RZ12], allows for a property-driven graph abstraction based on given collections of "patterns" describing structures of interest. This method is more fine-grained than neighbourhood abstraction, although implementing it remains as future work.

3.8 Summary

In this chapter we have:

- described how to reason with Hoare logic, and the notions of soundness and (relative) completeness;
- defined an abstract notion of assertion language, emphasising that for graph programs they need not be logical languages;
- defined three notions of correctness – partial, weak total, and total – based on the operational semantics of graph programs;
- defined three extensional Hoare calculi (i.e. calculi not tied to particular assertion languages) for these three notions of correctness;

- shown that the model checking problem becomes undecidable when there are effective constructions for expressing the existence of successful or failing executions of arbitrary programs;
- proven the soundness of our calculi with respect to the three notions of correctness;
- proven the relative completeness of our partial correctness rules, assuming the expressiveness of the assertion language;
- proven the existence of termination functions for iterated programs that terminate;
- briefly reviewed alternative approaches to verification in graph transformation.

Chapter 4

Verification with E-Conditions

In Chapter 3 we introduced Hoare calculi for verifying graph programs, though we did so in an extensional style, i.e. without tying them to a particular assertion language. In doing so we allowed for a separate and clean treatment of both the proof rules themselves and the assertion languages we "plug in" to them. We also discussed tradeoffs between the power of the proof rules and the power of the assertion languages, such as the decidability of the model checking problem when particular properties can be expressed.

The next step we take in this thesis is to study "instances" of our calculi that are fixed to a particular assertion language: one that can express interesting properties of graphs, but for which the model checking property is decidable, and for which there are constructions for weakest liberal preconditions, the existence of successful executions, and the existence of failing executions. This approach, in contrast to the previous chapter, is referred to as an *intensional* one.

This chapter begins with a discussion on our proposed assertion language – nested conditions with expressions (short E-conditions) – which adds expressions and relations over labels to the nested conditions developed by Habel, Pennemann, and Rensink [HP09, Ren04]. Then, we discuss Hoare calculi with E-conditions that, in particular, define transformations for Wlp, SE, and FE for arbitrary E-conditions and a certain class of programs. We consider technical properties of the calculi and language, such as the soundness of the rules, correctness of the transformations, and the problem of (relative) completeness. Finally, we explore some related and historical work (with a particular focus on the work by Pennemann et al.).

4.1 Nested Conditions with Expressions

When studying the verification of simple, imperative programming languages such as those described in [AdO09, Win93], pre- and postconditions

are often expressed using formulae of first-order logic. Typically, a formula will express properties about "locations", which under an interpretation correspond to the values of variables in a particular program state.

For graph programs, a different approach for assertional reasoning is needed. Program states not mappings from locations to values, but rather are graphs in $\mathcal{G}(\mathcal{L})$. The idea of reasoning about locations is not abstract enough for our purposes; instead, we need a way to intuitively (but formally) reason about properties of graphs. In the case of GP, we might want to reason about structural properties of graphs, such as:

P1 "the graph is loop free";

P2 "every node is incident to exactly one outgoing edge";

P3 "the graph is undirected";

but also it is important to be able to reason about properties of the labels, the manipulation of which is an important practical feature of GP:

P4 "no two adjacent nodes have the same label";

P5 "every node labelled by an integer i has an outgoing edge to a node labelled 0 or j where $j > i$";

P6 "all nodes are labelled by integers, and every pair of nodes has an edge between them if the source label is smaller than the target label".

The formalism we propose in this section – *nested conditions with expressions* (short *E-conditions*) – allows for the expression of such properties about both graph structure and relations between labels. E-conditions combine ideas from (conditional) rule schemata, logic, and nested conditions (due to [HP09, Ren04]), to be a formalism situated at the same level of abstraction as graph programs, whilst remaining intuitive for users with only a basic understanding of logic. The level of abstraction is also key for the constructions of the assertions Wlp, SE, and FE characterised in Section 3.4, since the "interaction" between rule schemata and pre- and postconditions can be studied using familiar techniques from graph transformation, e.g. morphisms, pushouts.

4.1.1 E-Conditions by Example

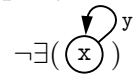
Before considering the formal definitions of E-conditions and their satisfaction relation, we want to convey an intuition as to how they can express graph properties. We will do so by example, using the properties **P1**–**P6** listed earlier in this section to introduce the features of the formalism. (For

simplicity, we take that **P1**–**P6** refer only to the parts of graphs that are unmarked.)

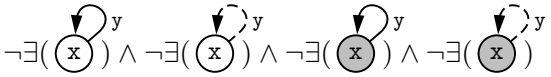
At their simplest, E-conditions have the form:

$$\exists(C)$$

where C is a graph labelled with expressions. A graph G in $\mathcal{G}(\mathcal{L})$ satisfies such an E-condition if there is an assignment α from variables in C to values in $\mathbb{L}$ such that there is an injective morphism from C^α to G. We can use Boolean negation $\neg$ to express the non-existence of such a morphism, and so straight away express the property **P1** with the E-condition:



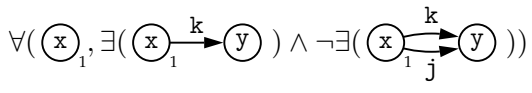
Here, `x` and `y` in the labels are variables that can be instantiated to any value in $\mathbb{L}$. A graph G will satisfy this if it does not contain a node incident to a loop with any combination of labels. We can strengthen this to forbid loops when either the node or edge are marked (or both) using Boolean connectives:



It is important to note here that the `x`s and `y`s are not bound to each other: the E-condition forbids the existence of any loop regardless of marking, and regardless of label.

Remark 4.1 (Marked nodes and edges)**.** In this thesis, unless a program is explicitly utilising marked nodes or edges, for simplicity we will usually write E-conditions about only the unmarked parts of the graph. When we write that some E-condition expresses that "the graph has property X", this is really short for the more correct: "the graph – ignoring all marked edges, marked nodes, and edges incident to them – has property X". If no rule schema of a program matches or creates marked items, then any marked items in the input graph will be present and unchanged in the resulting graph. Of course, if marked items are operated on, it will usually be necessary to write E-conditions about those items when constructing interesting proofs. □

Property **P2**, "every node is incident to exactly one outgoing edge", requires a less simple E-condition. When descriptions of properties have the pattern "every instance of some structure X should be within some particular context Y" we require a combination of universal quantification and nesting. Informally, the universally quantified part describes X, and then a nested E-condition describes the required context Y. The following E-condition describes property **P2**:



This E-condition is satisfied by any graph in which every node is incident to an outgoing edge, but not incident to more than one. Numbers are used to indicate which nodes are the same down the levels of nesting; here, the node labelled x (which is universally quantified) appears in the nested E-condition where information about its required context is given. Again, the ks and ys in the nested E-condition are not bound together, but this time, the xs are bound. Once a variable is used within an E-condition, every other occurence of it down the levels of nesting evaluates to the same value under an assignment. It can help to think of this nesting in E-conditions as a tree of (injective) graph morphisms[1] equipped with Boolean symbols, as in Figure 4.1.

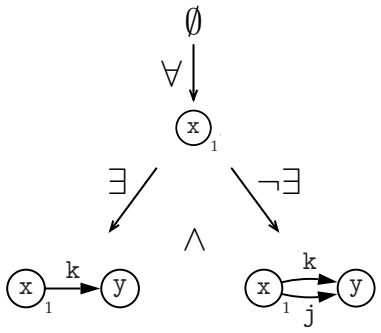
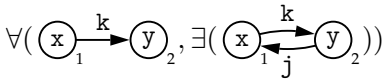


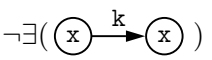
Figure 4.1: Tree representation of the E-condition expressing **P2**

Property **P3** can be expressed by an E-condition with nesting, too. Here, we define undirected to mean that if there is an edge from some node v_1 to v_2, then there is also an edge from v_2 to v_1:



Here, the xs (resp. ys, ks) are bounded.

Thus far, the graphs of our E-conditions have been labelled with variables distinct from each other and representing any combination of labels from $\mathbb{L}$. An important and powerful feature of E-conditions that we have not yet discussed, is the ability to constrain what values in $\mathbb{L}$ these variables represent. There are two ways of achieving this. One way is through using expressions as labels, e.g. one node might have label x with another having label x⋆x. Writing properties about nodes and edges with the same label is even simpler, as this E-condition for **P4** demonstrates:



[1] Actually, E-conditions *are* morphisms equipped with Boolean symbols (and assignment constraints). We will properly reveal and discuss this technical detail shortly.

Under any assignment both xs must be mapped to the same value in $\mathbb{L}$, hence this E-condition is only satisfied by graphs that have no two adjacent nodes with the same value.

The other way of constraining the instantiation of variables is to add so-called assignment constraints. Informally, assignment constraints are simple Boolean expressions establishing relations between variables, and restricting the types of values that variables can (or cannot) be instantiated to. Given an assignment α for a graph, an assignment constraint must evaluate to true when interpreted with regards to α. Assignment constraints can be written for each graph in each level of nesting of an E-condition; they are displayed after a vertical bar (which can be read aloud as "where" or "such that"). For example, property **P5** is expressed by:

$$\forall(\textcircled{\texttt{x}}_1 \mid \texttt{int(x)}, \exists(\textcircled{\texttt{x}}_1 \xrightarrow{\texttt{k}} \textcircled{\texttt{y}} \mid \texttt{y = 0 or y > x}))$$

A graph will satisfy this E-condition, if every node with an integer label x has an outgoing edge to a node labelled with 0 or some y that is larger than x. The innermost assignment constraint is hopefully clear. The outermost one, `int(x)`, will evaluate to true under an assignment if `x` is mapped to an integer; false if not. In rule schemata, such a predicate is unnecessary because all variables are typed. In E-conditions however, it is important to note that for notational simplicity, all variables are treated as untyped.

Remark 4.2 (Untyped variables)**.** For notational simplicity, all of the variables appearing in the graphs and assignment constraints of E-conditions are untyped. Rather, predicate symbols in the assignment constraints are used to restrict the permissible values in $\mathbb{L}$ that a variable can be mapped to. Type safety is imposed not at the syntactic level, but rather at the semantic level: assignments are required to be "well-typed". For example, an assignment is not well-typed if it maps a variable to a string when that same variable appears within an arithmetic expression. □

Property **P6** can be expressed by an E-condition exploiting several of the features we have discussed:

$$\neg\exists(\textcircled{\texttt{x}} \mid \texttt{not int(x)}) \wedge \forall(\textcircled{\texttt{x}}_1\ \textcircled{\texttt{y}}_2 \mid \texttt{x < y}, \exists(\textcircled{\texttt{x}}_1 \xrightarrow{\texttt{k}} \textcircled{\texttt{y}}_2))$$

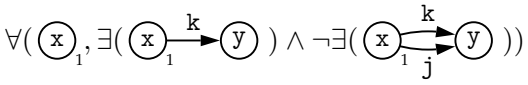
The first conjunct is satisfied by graphs in which every node has an integer label[2]. The second conjunct is satisfied if every pair of nodes has an edge between them if the source label is smaller than the target label.

Up until now we have ignored an important technical detail: our E-conditions for properties **P1**–**P6** have all expressed properties about graphs,

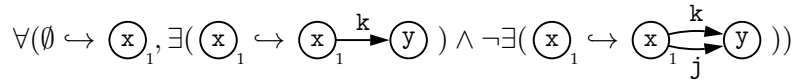
[2]It might appear that universal quantification could remove the need for double negation. But such an E-condition without nesting, i.e. $\forall(C \mid \gamma)$ is vacuously true – this will become clear from the formal semantics of satisfaction.

but actually, the formalism is more general than this, in that it expresses properties about graph morphisms. As for nested conditions [HP09], E-conditions can be seen as generalisations of graph consistency constraints [HW95] and (negative) application conditions [HHT96].

As hinted towards in Figure 4.1, E-conditions are trees of (injective) morphisms equipped with Boolean symbols and assignment constraints. We have simply been following a convention (that we will continue) of drawing only the codomain of morphisms. So whereas we wrote:



for property **P2**, the same E-condition including the morphisms would be:



When expressing properties of graphs, the root in the tree of morphisms is always the empty graph. E-conditions of this form are sometimes referred to as *E-constraints*, a nod towards so-called graph constraints. When expressing properties about, e.g. matches for a rule schema with left graph L, the root in the tree of morphisms would be L[3].

4.1.2 Definition of E-Conditions

Having hopefully given an intuition for reading and writing E-conditions, we now proceed to give in this subsection a formal definition. This definition relies however on two others: (1) the label alphabet for the graphs of E-conditions; and (2) the abstract syntax of assignment constraints. These are considered in order, before we formally give a definition of E-conditions and discuss some notational conventions.

The label alphabet we fix for the graphs of E-conditions is very similar to that for the graphs of rule schemata. This is intentional, to allow for E-conditions to express properties about particular matches for rule schemata. The only difference, as hinted towards in Section 4.1.1, is that variables all belong to VarId, a class of *untyped* variable identifiers. This allows for notational simplicity, avoiding the need to repeatedly declare types of variables, instead only restricting possible assignments when necessary through the assignment constraints. This approach simplifies the presentation syntactically; but the need to, for example, avoid the `xs` in `x*x+5` being mapped to any value in $\mathbb{L}$ other than an integer, is now dealt with in the semantics of satisfaction by requiring assignments to be *well-typed* for expressions.

[3]Ignore the mismatch between variables in rule schemata and E-conditions for the moment. This will be addressed shortly.

Definition 4.3 (Graph labels for E-conditions)**.** Let EC (for E-Conditions) denote the label alphabet containing all expressions that can be derived from the syntactic class ECLabel of Figure 4.2. Here, VarId is a class of untyped variable identifiers. By $\mathcal{G}(\text{EC})$ we denote the class of all graphs labelled over EC. □

Integer	::=	Digit {Digit} | VarId | '−' Integer
		| Integer ArithOp Integer
ArithOp	::=	'+' | '−' | '*' | '/'
String	::=	'"' {Char} '"' | VarId | String '.' String
Atom	::=	Integer | String | VarId
List	::=	`empty` | Atom | VarId | List ':' List
ECLabel	::=	List Mark
Mark	::=	`true` | `false`

Figure 4.2: Abstract syntax of graph labels for E-conditions

For technical reasons, we treat VarId as a strict superset of the classes of typed variables, as shown in the Venn diagram of Figure 4.3. This allows us to, for example, use the left- and right-hand graphs of rule schemata in E-conditions since $\mathcal{G}(\text{RS})$ can now be viewed as a strict subset of $\mathcal{G}(\text{EC})$. We simply treat variables from LVar, AVar, SVar, and IVar as untyped when they are used within the context of E-conditions (using assignment constraints when it is necessary to restrict assignments of variables to particular types).

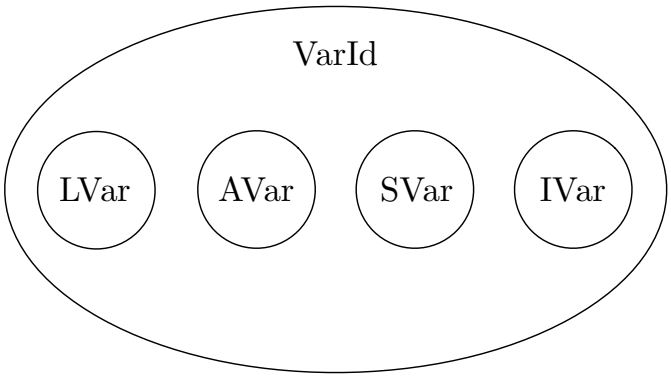


Figure 4.3: Venn diagram of the variable classes

Remark 4.4 (Naming of syntactic categories)**.** Note that we use the same names (List, Integer, etc.) for the syntactic categories of Figure 4.2 as we do

for the grammar of Figure 2.11. This is for simplicity, since both grammars generate identical expressions – modulo variable classes – and the correct grammar can usually be inferred from the context (if not, we explicitly state it). □

Assignment constraints are simple Boolean expressions that form a component of E-conditions. They allow for the restriction of possible values in $\mathbb{L}$ that variables can be mapped to. This is achieved through simple relations – such as equality, greater than, less than – predicates about type, and Boolean operators.

Definition 4.5 (Assignment constraint)**.** An *assignment constraint* γ is simply a Boolean expression generated by the syntactic category AssCon of Figure 4.4. All variables in expressions are untyped and belong to the class VarId, and List, Integer are as defined in Figure 4.2. □

AssCon	::=	List ('=' | '\=') List | Integer IntRel Integer | `not` AssCon
		| AssCon (`and` | `or`) AssCon | Type '`(`' List '`)`' | `true`
IntRel	::=	'>' | '<' | '>=' | '<='
Type	::=	`int` | `string` | `atom`

Figure 4.4: Abstract syntax of assignment constraints

We will give a formal semantics of assignment constraints in Section 4.1.3, but the intended meaning of most constraints should be intuitively clear from the syntax. Assignment constraints of the form $t\,(l)$ with t in Type, l in List, are to be interpreted as predicates[4] (Boolean-valued functions) and evaluate to true if the input is of the indicated type (false otherwise).

Example 4.6 (Assignment constraints)**.** Consider the assignment constraint:

`a > b and c = a+b:d and atom(d)`

where `a`, `b`, `c`, `d` are variables. The intended meaning, informally, is that `c` is mapped to a list of length two: the first element the sum of two integers (the first integer larger than the second), and the second element some atom (an integer or string). □

Notation 4.7 (Type predicate notation)**.** For brevity, we introduce some notation for when several variables are to be constrained to the same type. For lists $l_1, \ldots, l_n$ and type t in Type, we allow:

$$t\,(l_1, \ldots, l_n)$$

[4]In previous work, we were writing `type(`l`)` $= t$ instead of predicates.

to be short for:

$$t\,(l_1)\ \texttt{and}\ \ldots\ \texttt{and}\ t\,(l_n)\,.$$

□

With the label alphabet and assignment constraints defined, we now can give a definition of E-conditions. Recall from Section 4.1.1 that E-conditions are trees of injective graph morphisms equipped with logical symbols and assignment constraints. We formalise these notions now and discuss some conventions we use to write them as simply as possible.

Definition 4.8 (E-condition)**.** A *nested condition with expressions* (short *E-condition*) c over a graph P is of the form $\texttt{true}$ or $\exists(a \mid \gamma,\, c')$, where $a\colon P \hookrightarrow C$ is an injective graph morphism with $P, C \in \mathcal{G}(\mathrm{EC})$, γ is an assignment constraint, and c' is an E-condition over C.

Boolean formulae over E-conditions over P yield E-conditions over P, that is, $\neg c$, $c_1 \wedge c_2$, and $c_1 \vee c_2$ are E-conditions over P if c, c_1, c_2 are E-conditions over P. □

We remark that the morphisms of E-conditions are injective[5] because matching in GP is injective.

Several important abbreviations of E-conditions are introduced (e.g. universal quantification, writing only codomains) in the following note, that will be used extensively throughout the remainder of this thesis.

Notation 4.9 (Abbreviating E-conditions)**.** For brevity, we write $\texttt{false}$ for $\neg\texttt{true}$, $c \Rightarrow d$ for $\neg c \vee d$, $c \Leftrightarrow d$ for $c \Rightarrow d \wedge d \Rightarrow c$, $\exists(a \mid \gamma)$ for $\exists(a \mid \gamma, \texttt{true})$, $\exists(a, c')$ for $\exists(a \mid \texttt{true}, c')$, and $\forall(a \mid \gamma, c')$ for $\neg\exists(a \mid \gamma, \neg c')$. In our examples, when the domain of morphism $a\colon P \hookrightarrow C$ can unambiguously be inferred, we write only the codomain C. For instance, an E-condition $\exists(\emptyset \hookrightarrow C, \exists(C \hookrightarrow C'))$ can be written as $\exists(C, \exists(C'))$, where the domain of the outermost morphism is the empty graph, and the domain of the nested morphism is the codomain of the encapsulating E-condition's morphism. □

Since E-conditions are trees of morphisms, the domains of morphisms along the nesting are easily inferred. The domain of the "outermost" morphism, i.e. the root of the tree, can also be easily inferred depending on the context. E-conditions expressing graph properties (e.g. properties **P1**–**P6**) have as their root the empty graph, $\emptyset$ (these common types of E-conditions are often referred to as E-constraints). E-conditions expressing properties of graphs relative to some rule schema match (resp. comatch) will have the left-hand (resp. right-hand) graph of the rule schema as their root.

[5] In Section 6.3 we show how node (or edge) identification can be represented in E-conditions.

Definition 4.10 (E-constraint)**.** An E-condition c is an *E-constraint* if $c = \texttt{true}$ or if $c = \exists(a : \emptyset \hookrightarrow C \mid \gamma, c')$. Boolean formulae over E-constraints yield E-constraints, that is, $\neg c$, $c_1 \wedge c_2$, and $c_1 \vee c_2$ are E-constraints if c, c_1, c_2 are E-constraints. □

Example 4.11 (E-constraint)**.** The E-condition:

$$\forall(\ \textcircled{x}_1 \xrightarrow{k} \textcircled{y}_2 \mid \texttt{x} > \texttt{y}, \exists(\ \textcircled{x}_1 \overset{l}{\circlearrowleft}, \textcircled{x}_1 \xrightarrow{k} \textcircled{y}_2))$$

(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:

$$\neg\exists(\emptyset \hookrightarrow \textcircled{x}_1 \xrightarrow{k} \textcircled{y}_2 \mid \texttt{x} > \texttt{y},\ \neg\exists(\ \textcircled{x}_1 \xrightarrow{k} \textcircled{y}_2 \hookrightarrow \textcircled{x}_1 \overset{l}{\circlearrowleft}, \textcircled{x}_1 \xrightarrow{k} \textcircled{y}_2 \mid \texttt{true}, \texttt{true})).$$

□

Before we move on to the semantics of satisfaction, we introduce a notational convenience allowing us to express concisely facts and requirements about variables present in graphs, morphisms, and E-conditions.

Definition 4.12 (Variable set vars(x))**.** We denote by vars(x) the set of all variables appearing in x, where x can be a graph, graph morphism, rule schema condition, (conditional) rule schema, assignment constraint, or E-condition. □

4.1.3 Semantics of Satisfaction

In this section we give a formal definition of satisfaction for E-conditions, which we informally described for examples about graph properties in Section 4.1.1.

In general, we are checking whether morphisms between graphs in $\mathcal{G}(\mathcal{L})$ satisfy E-conditions (checking whether graphs satisfy E-constraints is a particular instance of this task). Satisfaction relies on assignments under which we evaluate the graphs of the E-condition and its assignment constraint. It also relies on the notion of substitution to ensure equal assignment of variables along the nesting (or down the tree of morphisms). We proceed first by defining assignments and substitution for E-conditions, as well as defining how assignments and substitutions are applied to lists and assignment constraints.

Definition 4.13 (Assignment for E-conditions; domain)**.** An *assignment* α is a partial function $\alpha : \text{VarId} \to \mathbb{L}$. The *domain of* α, denoted by dom(α), is the set of all variables for which α is defined. □

Recall that graphs in $\mathcal{G}(\text{EC})$ may be labelled with expressions containing type-specific operators, e.g. `x+y` for integers, `s.t` for strings. Because in E-conditions we do not consider variables as typed, we must handle type safety at the semantic level by requiring that assignments for satisfaction are *well-typed*. Intuitively, a well-typed assignment will not map any variables to values for which expressions and assignment constraints cannot be semantically interpreted, e.g. a well-typed assignment for `x+y` (resp. `s.t`) will map `x`, `y` to integers (resp. `s`, `t` to strings).

Definition 4.14 (Well-typed assignment)**.** Let l denote a list. An assignment α is *well-typed for* l, if (1) it is defined for all variables in l; (2) variables in integer expressions are mapped to values in $\mathbb{Z}$; and (3) variables in string expressions are mapped to values in Char^*.

Given a graph $G \in \mathcal{G}(\text{EC})$, an assignment α is *well-typed for* G if it is well-typed for all of the lists occurring in G. Similarly, if $g : G \to H$ is a morphism and $G, H \in \mathcal{G}(\text{EC})$, then an assignment α is *well-typed for* g if it is well-typed for G and H.

Let γ be an assignment constraint. An assignment α is *well-typed for* γ, if (1–3) above hold for all lists occurring in γ, and moreover, variables in integer relations are mapped to values in $\mathbb{Z}$. □

Next, we define the evaluation of labels and assignment constraints with respect to well-typed assignments. Labels, graphs, and assignment constraints respectively evaluate to values in $\mathcal{L}$, graphs in $\mathcal{G}(\mathcal{L})$, and Boolean values in $\mathbb{B}$.

Definition 4.15 (Application of assignments)**.** Let $(l\ b)$ denote some label with l a list (with untyped variables) and b a mark, and let α denote some assignment well-typed for l. The *application of* α *to* $(l\ b)$, denoted by $(l\ b)^\alpha$, is defined analogously to Definition 2.38.

Given a graph G in $\mathcal{G}(\text{EC})$ and an assignment α well-typed for G, we write G^α for the graph in $\mathcal{G}(\mathcal{L})$ that is obtained from G by replacing each label $(l\ b)$ with $(l\ b)^\alpha$ (note that G^α has the same nodes, edges, source and target functions as G). If $g\colon G \to H$ is a graph morphism with $G, H \in \mathcal{G}(\text{EC})$, then g^α denotes the morphism $\langle g_V^\alpha, g_E^\alpha \rangle\colon G^\alpha \to H^\alpha$.

Let γ denote some assignment constraint, and α denote some assignment well-typed for γ. The *application of* α *to* γ, denoted γ^α, is defined inductively as follows.

1. If γ is `true`, then $\gamma^\alpha = \text{true}$. If γ has the form $l_1 = l_2$ (resp. $l_1 \mathrel{\backslash=} l_2$) with l_1, l_2 in List, then $\gamma^\alpha = \text{true}$ if $l_1^\alpha = l_2^\alpha$ (resp. $l_1^\alpha \neq l_2^\alpha$), or false otherwise. If γ has the form $i_1 \bowtie i_2$ with i_1, i_2 in Integer and $\bowtie$ in IntRel, then $\gamma^\alpha = \text{true}$ if $i_1^\alpha \bowtie_{\mathbb{B}} i_2^\alpha$ where $\bowtie_{\mathbb{B}}$ is the relation on integers represented by $\bowtie$, and false otherwise.

2. If γ has the form `not` γ_1 with γ_1 in AssCon, then γ^α = true if γ_1 = false, and false otherwise. If γ has the form γ_1 `and` γ_2 (resp. γ_1 `or` γ_2) with γ_1, γ_2 in AssCon, then γ^α = true if γ_1^α = true = γ_2^α (resp. γ_1^α = true or γ_2^α = true).

3. Finally, if γ has the form $t\,(l)$ with l in List and t = `int` (resp. `string`, `atom`), then γ^α = true if $l^\alpha \in \mathbb{Z}$ (resp. Char*, $\mathbb{Z} \cup$ Char*), and false otherwise.

□

We define *substitutions* for replacing variables with list expressions. The notion is required (1) in the definition of satisfaction, for ensuring consistency of labels down the tree of morphisms; and (2) in the construction of weakest liberal preconditions.

Definition 4.16 (Substitution for untyped variables; domain)**.** A *substitution* is a partial function σ : VarId $\to$ List where List is as defined in Figure 4.2. The *domain of* σ, denoted by dom(σ), is the set of all variables for which σ is defined. □

Substitutions also require the notion of well-typedness; without it, it would be possible to transform lists such as `x+5` into, e.g. `"Plump"+5`, which is a syntactically invalid list (and cannot be semantically interpreted).

Definition 4.17 (Well-typed substitution)**.** Let l denote a list. A substitution σ is *well-typed for* l, if (1) variables in integer expressions are mapped to expressions in Integer; (2) variables in string expressions are mapped to expressions in String.

Given a graph $G \in \mathcal{G}(\text{EC})$, a substitution σ is *well-typed for* G if it is well-typed for all of the lists occurring in G. Similarly, if $g : G \to H$ is a morphism and $G, H \in \mathcal{G}(\text{EC})$, then a substitution σ is *well-typed for* g if it is well-typed for G and H.

Let γ be an assignment constraint. A substitution σ is *well-typed for* γ, if (1–2) above hold for all lists occurring in γ, and moreover, variables in integer relations are mapped to expressions in Integer.

Let c be an E-condition. A substitution σ is *well-typed for* c, if c = `true`, or if $c = \exists(a \mid \gamma, c')$ and σ is well-typed for a, γ, c', or if c is a Boolean formula over E-conditions and σ is well-typed for each E-condition within it. □

The application of substitutions is defined simply as the in-place replacement of variables in lists and assignment constraints.

Definition 4.18 (Application of substitutions)**.** Let $(l\ b)$ denote some label with l a list (with untyped variables) and b a mark, and let σ denote some substitution well-typed for l. The *application of* σ *to* $(l\ b)$, denoted $(l\ b)^\sigma$,

is defined $(l\ b)^\sigma = (l^\sigma\ b)$. Here, l^σ is obtained from l by replacing every variable $\mathtt{x}$ for which σ is defined with $\sigma(\mathtt{x})$.

Let G be a graph in $\mathcal{G}(\text{EC})$ and σ be a substitution well-typed for G. Then $G^\sigma \in \mathcal{G}(\text{EC})$ is the graph obtained from G by replacing each label $l\ b$ with $(l\ b)^\sigma$. If $g\colon G \to H$ is a graph morphism with $G, H \in \mathcal{G}(\text{EC})$, then g^σ denotes the morphism $\langle g_V, g_E\rangle\colon G^\sigma \to H^\sigma$.

Let γ denote an assignment constraint γ, and σ a substitution well-typed for γ. Then γ^σ is the assignment constraint obtained from γ by replacing each list l with l^σ.

Let c denote E-conditions, and σ a substitution well-typed for c. Then when c has the form $\mathtt{true}$, $c^\sigma = \mathtt{true}$, and when c has the form $\exists(a \mid \gamma, c')$, $c^\sigma = \exists(a^\sigma \mid \gamma^\sigma, (c')^\sigma)$. If c is a Boolean formula over E-conditions, then σ is distributed over the E-conditions, i.e. $(\neg c_1)^\sigma = \neg c_1^\sigma$, $(c_1 \wedge c_2)^\sigma = c_1^\sigma \wedge c_2^\sigma$, and $(c_1 \vee c_2)^\sigma = c_1^\sigma \vee c_2^\sigma$. □

We remark that in our definition, if a substitution is not defined for a variable $\mathtt{x}$, then $\mathtt{x}^\sigma = \mathtt{x}$.

Finally, before defining the satisfaction relation, we define *substitutions induced by assignments*, which we use to enforce the consistency of labels down the tree of morphisms.

Definition 4.19 (Substitutions induced by assignments)**.** Let $\alpha : \text{VarId} \to \mathbb{L}$ denote some assignment. The *substitution induced by* α, denoted $\sigma_\alpha : \text{VarId} \to \text{List}$, maps every variable $\mathtt{x}$ in the domain of definition of α to the list that is obtained from $\alpha(\mathtt{x})$ by replacing integers and strings with their syntactic counterparts. □

Example 4.20 (Induced substitution)**.** Consider the assignment $\alpha = (\mathtt{x} \mapsto 23, \mathtt{y} \mapsto 56 : \text{“York”} : \text{“cat”})$ where $23, 56$ are integers and “York”, “cat” are strings. Then the substitution induced by α would be $\sigma_\alpha = (\mathtt{x} \mapsto \mathtt{23}, \mathtt{y} \mapsto \mathtt{56} : \text{"}\mathtt{York}\text{"} : \text{"}\mathtt{cat}\text{"})$ where $\mathtt{23}$, $\mathtt{56}$ are syntactic digits, and "$\mathtt{York}$", "$\mathtt{cat}$" are syntactic strings. □

With the concepts of assignment, substitution, and induced substitutions now defined, we can proceed to define the satisfaction of E-conditions by morphisms, and the special case of satisfying E-constraints by graphs.

Definition 4.21 (Satisfaction of E-conditions)**.** We define inductively the *satisfaction* of E-conditions by injective graph morphisms. 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 E-condition $c = \exists(a\colon P \hookrightarrow C \mid \gamma, c')$, denoted $s \models c$, if there exists an assignment α that is well-typed for a, γ and is undefined for variables present only in c', such that $S = P^\alpha$, and such that there is an injective graph morphism $q\colon C^\alpha \hookrightarrow G$ with $q \circ a^\alpha = s$, $\gamma^\alpha = \text{true}$, and $q \models (c')^{\sigma_\alpha}$. Here, σ_α is the substitution induced by α, which we require to be well-typed for all morphisms and assignment constraints

in c'. If such an assignment α and morphism q exist, we say that s *satisfies* c *by* α, and write $s \models_\alpha c$. Figure 4.5 summarises $s \models_\alpha c$ (assuming that $\gamma^\alpha = \text{true}$). □

$$\begin{array}{ccc} S = P^\alpha & \overset{a^\alpha}{\hookrightarrow} & C^\alpha \\ & \underset{G}{\overset{s \searrow \quad = \quad \swarrow q}{}} & \models (c')^{\sigma_\alpha} \end{array}$$

Figure 4.5: Satisfaction of an E-condition

We write $\not\models$ and $\not\models_\alpha$ in place of $\models$ and $\models_\alpha$, respectively, when the satisfaction relation does not hold for a morphism and E-condition.

The satisfaction of Boolean formulae over E-conditions is defined inductively. We have $s \models \neg c$ if $s \not\models c$, and $s \models c \wedge d$ (resp. $s \models c \vee d$) if $s \models c$ and $s \models d$ (resp. $s \models c$ or $s \models d$). Given an assignment α, we have $s \models_\alpha \neg c$ if $s \not\models_\alpha c$, and $s \models_\alpha c \wedge d$ (resp. $s \models_\alpha c \vee d$) if $s \models_\alpha c$ and $s \models_\alpha d$ (resp. $s \models_\alpha c$ or $s \models_\alpha d$).

Definition 4.22 (Satisfaction of E-constraints)**.** A graph G in $\mathcal{G}(\mathcal{L})$ *satisfies* an E-constraint c, denoted $G \models c$, if $i_G \colon \emptyset \hookrightarrow G \models c$[6]. □

Two E-conditions (resp. E-constraints) are said to be equivalent, if they satisfy exactly the same morphisms (resp. graphs).

Definition 4.23 (Equivalent E-conditions)**.** E-conditions c, d are said to be *equivalent*, denoted $c \equiv d$, if for all injective graph morphisms $s : S \hookrightarrow G$ with $S, G \in \mathcal{G}(\mathcal{L})$, we have that:

$$s \models c \text{ if and only if } s \models d.$$

□

4.1.4 Comparison with Nested Conditions

The question of the expressiveness of E-conditions is explored as part of Chapter 6 – but more immediately one might wonder how the formalism compares to the nested conditions described in [HP09]. In one sense, E-conditions inherit the limitations present also in nested conditions, in particular with what can be expressed structurally about a graph. Both formalisms are only able to express local (in the sense of Gaifman, see [Lib04, Gai82]) structural properties about graphs, and are unable to express "global" properties such as:

[6]The i naming the morphism stands for "initial", as the empty graph is the initial object in the category of graphs and graph morphisms. There is exactly one morphism from $\emptyset$ to any graph G.

- "the graph has a path of arbitrary length";
- "the graph is connected";
- "the graph contains a Hamiltonian cycle".

E-conditions however enrich what can be expressed about structure by allowing reasoning about data in labels. Relations between (potentially infinite sets of) labels can be expressed – which cannot be done so by nested conditions – allowing for practical reasoning about programs that manipulate graph labels.

Moreover, when expressing properties that do not concern specific labels, E-conditions have an advantage. The graphs in nested conditions are labelled over the same, finite label alphabet as the graphs they express properties about. The result is that to express a structural property regardless of labels, a disjunction over all possible combinations of labels is required. If the graphs of nested conditions are labelled over an infinite label alphabet, such as $\mathcal{L}$, then infinite disjunctions are required to express even the most basic structural properties. For example, to express that there exists an unmarked integer-labelled node, we would require the infinite nested condition:

$$\exists(\,\textcircled{0}\,) \vee \exists(\,\textcircled{1}\,) \vee \exists(\,\textcircled{-1}\,) \vee \exists(\,\textcircled{2}\,) \vee \exists(\,\textcircled{-2}\,) \vee \cdots$$

whereas this can be expressed finitely using a variable in an E-condition:

$$\exists(\,\textcircled{x} \mid \texttt{int(x)}\,).$$

Another difference is in the types of morphism allowed in the two formalisms. In nested conditions, the graph morphisms are arbitrary. In E-conditions however, we require that the morphisms are injective (since the matching in GP is injective). The result is that it is not possible to merge nodes and edges along the tree of morphisms as it is in nested conditions. In Section 6.3 however, we show that this merging can be expressed indirectly.

4.2 Proof Rules with E-Conditions

In this section we begin by defining E-constraints as an assertion language in the Hoare-style verification framework of Chapter 3. We then study instances of our Hoare proof rules specifically for E-constraints, discussing and justifying some restrictions on the body of loops and conditionals that allow us to define constructions for Wlp, SE, and FE.

We use the definition of E-constraints and their satisfaction relation to formally define an assertion language – one that we fix for the rest of this

chapter. Note that for simplicity we do not add a subscripted symbol after the satisfaction relation (as was the convention in Chapter 3), since it is clear in this chapter what assertions the relation gives meaning to.

Definition 4.24 (E-constraints as assertions)**.** Let $\mathfrak{E}$ denote the assertion language $\langle E, \models \rangle$, where E is the set of all E-constraints, and $\models \subseteq \mathcal{G}(\mathcal{L}) \times E$ is the satisfaction relation for E-conditions. □

The model checking problem – the task of checking whether a graph satisfies an assertion – is decidable for E-constraints. This of course is a basic requirement for our verification calculus to be practical.

Fact 4.25 (Model checking decidable for E-constraints)**.** Let G be a graph in $G(\mathcal{L})$ and c be an E-constraint. The decision problem:

Model Checking Problem (MCP) for $\mathfrak{E}$	
Input:	graph G, E-constraint c
Question:	does $G \models c$?

is decidable. □

Recall that the decidability of the model checking problem for the assertion language of E-constraints tells us that SE and FE cannot be effectively constructed for arbitrary programs and assertions (see Proposition 3.20). To get around this problem we impose a restriction on the form of programs in proof rules that require reasoning about successful and failing executions. The intention being that a restriction allows for there to be constructions for SE and FE, whilst not being so stifling to make the verification calculus impractical.

The restriction we impose is that the bodies of loops and the "guards" of conditionals are simply sets of (conditional) rule schemata, as opposed to arbitrary programs. This does not affect the computational completeness of GP (according to [HP01]), and in our experience, having arbitrary programs is not always necessary in loop bodies and guards: see for example the programs we prove properties about in Chapter 5. Moreover, we feel that being able to algorithmically construct assertions defining SE and FE brings practical benefits that make up for convenience lost by disallowing arbitrary programs in loop bodies and guards of conditionals.

Example 4.26 (Restricted graph programs)**.** Consider for example the common pattern of graph program below (we do not specify particular rule schemata or macros, but name them in a way to convey some intention), which does not meet the restriction of our verification calculi:

`if (computeSomething; checkSomething) then` P `else` Q

Here, the guard program begins by computing something on a graph, then checking something about the result of the computation to inform which branch of execution to follow next. For example, the guard program could nondeterministically mark a node in an undirected graph[7] and then repeatedly mark nodes adjacent to it. It could then, for example, check whether any node remains unmarked, thus determining in this example whether the graph is connected or not. Such programs can often be rewritten in the form:

`computeSomething`; `if checkSomething then` P `else` Q; `undoSomething`!

Here, the first part of the guard program is moved out of the conditional. The restriction is met, but now the effects of the computation must be "undone" at a later stage since `computeSomething` is no longer executed on a copy of the graph. (Note that this is harder to do if the program is performing a destructive test on the graph.) □

4.2.1 Partial Correctness Calculus

Let r (resp. $\mathcal{R}$) range over conditional rule schemata (resp. sets of conditional rule schemata), c, c', d, d', e, inv over E-constraints, and P, Q over graph programs. The proof rules in Figure 4.6 (also presented for reference in Figure B.1 of the appendix) together define a partial correctness calculus – as introduced extensionally in Section 3.4 – but with E-constraints specifically as the assertions. Additionally, Figure B.2 in the appendix defines partial correctness proof rules for the derived commands of GP.

The intuition behind each of the proof rules was given in Section 3.4, but we shall explain informally the realisations of Wlp, SE, and FE that are present in the proof rules of Figure 4.6.

The axiom $[\text{ruleapp}]_{\text{wlp}}$ defines a weakest liberal precondition $\text{Wlp}_{\mathfrak{E}}[r, c]$ with a disjunction of two transformations: $\text{Pre}(r, c) \vee \neg\text{App}(\{r\})$. The former expresses the weakest properties that must be satisfied for successful executions of r to establish the postcondition. The latter is used negated to express the property that r is not applicable to the graph, in which case any postcondition can be inferred (since r will fail on the graph). In practice, we find it simpler to treat these two scenarios separately: we define [ruleapp] for reasoning about successful executions of (conditional) rule schemata, and [nonapp] for reasoning about failing executions (i.e. when failure is implied by the precondition). We describe Pre and App informally in what follows, but remark that formal definitions and proofs are given later in Section 4.3.

The transformation $\text{Pre}(r, c)$ of $[\text{ruleapp}]_{\text{wlp}}$ and [ruleapp] is informally described by the following steps:

[7] That is, a graph satisfying property **P3** in Section 4.1.

$$[\text{ruleapp}]_{\text{wlp}}\ \frac{}{\{\text{Pre}(r,c) \vee \neg\text{App}(\{r\})\}\ r\ \{c\}}$$

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

$$[\text{nonapp}]\ \frac{}{\{\neg\text{App}(\{r\})\}\ r\ \{\texttt{false}\}}$$

$$[\text{ruleset}]\ \frac{\{c\}\ r\ \{d\} \text{ for each } r \in \mathcal{R}}{\{c\}\ \mathcal{R}\ \{d\}}$$

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

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

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

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

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

Figure 4.6: Partial correctness rules with E-constraints for core commands

1. form a disjunction of E-conditions over the right-hand graph of r, accounting for the possible ways in which c and comatches of r might "overlap";
2. shift this E-condition over to the left-hand graph of r; and
3. nest this within an E-constraint universally quantified over all possible matches of r (accounting also for its applicability).

The transformation $\text{App}(\mathcal{R})$ of $[\text{ruleapp}]_{\text{wlp}}$, [nonapp], [if], [try], and [!] takes as input a set of conditional rule schemata $\mathcal{R}$, and transforms it into an E-constraint expressing that at least one of the rule schemata within the set is applicable. (Again, this will be defined formally in Section 4.3.) If a

graph G satisfies $\text{App}(\mathcal{R})$, then there exists a direct derivation $G \Rightarrow_{\mathcal{R}} H$ for some graph H. If a graph G satisfies $\neg\text{App}(\mathcal{R})$, then there is no such direct derivation and executing $\mathcal{R}$ on G will lead to failure.

In [if], [try], and [!], $\text{SE}_{\mathfrak{E}}[c, \mathcal{R}]$ (resp. $\text{FE}_{\mathfrak{E}}[c, \mathcal{R}]$) is defined by the conjunction of c and $\text{App}(\mathcal{R})$ (resp. $\neg\text{App}(\mathcal{R})$). We remark that unlike for arbitrary programs, there is no set of rule schemata $\mathcal{R}$ such that both $\text{SE}_{\mathfrak{E}}[\texttt{true}, \mathcal{R}]$ and $\text{FE}_{\mathfrak{E}}[\texttt{true}, \mathcal{R}]$ can be defined by assertions satisfied by the same graph. If $\mathcal{R}$ is applicable to a graph, then its execution will always be successful. If $\mathcal{R}$ is not applicable to a graph, then its execution will always result in failure. There cannot be both successful and failing execution paths when we restrict to sets of rule schemata.

Finally, in [cons], the implications are expressed directly via a Boolean formula over E-constraints, and the user is required to show outside the calculus that the implications are valid in the following sense.

Definition 4.27 (Validity)**.** Let c denote an E-constraint. We say that c is *valid*, denoted $\models c$[8], if for all graphs $G \in \mathcal{G}(\mathcal{L})$ we have that $G \models c$. □

Automatically checking whether an E-constraint $c \Rightarrow d$ is valid is not considered in this thesis, but is important future work. This task has been explored by Pennemann for nested conditions [Pen08, Pen09], for which a resolution-like deduction calculus was proposed and prototyped. As for nested conditions, the problem of deciding the validity of E-constraints is not even semi-decidable. This is due to Trakhtenbrot's theorem from finite model theory (see e.g. [Lib04]), which applies in our context because the structures we interpret E-constraints over are finite graphs.

Fact 4.28 (Validity is undecidable)**.** Let c be an E-constraint. The decision problem:

Validity Problem (VP) for $\mathfrak{E}$	
Input:	E-constraint c
Question:	is c valid over all (finite) graphs, i.e. $\models c$?

is undecidable, and not even semi-decidable. □

4.2.2 Total Correctness Calculi

Let r (resp. $\mathcal{R}$) range over conditional rule schemata (resp. sets of conditional rule schemata), c, c', d, d', e, inv over E-constraints, and P, Q over graph programs. The proof rules in Figure 4.7 (also presented for reference in Figure B.3 of the appendix) together define a weak total correctness calculus – as introduced extensionally in Section 3.5 – but with E-constraints specifically as the assertions. Additionally, Figure B.4 in the appendix defines weak total correctness proof rules for the derived commands of GP.

[8] We sometimes omit $\models$ when showing validity is clear, e.g. in [cons].

[ruleapp]$_{\text{wlp}}$, [ruleapp], [nonapp], [ruleset], [comp], [cons] of Figure 4.6, and:

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

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

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

Figure 4.7: Weak total correctness proof rules with E-constraints for core commands

The intuition behind the weak total correctness proof rules (in the extensional approach) was given in Section 3.5, but we remark here on some differences for E-constraints.

In the conditional proof rules [if]$_{\text{wtot}}$, [try]$_{\text{wtot}}$, we do not require a third premise $\vdash_{\text{wtot}} \{c\}\ \mathcal{R}\ \{\top_{\mathfrak{E}}\}$ (where $\top_{\mathfrak{E}}$ is defined by `true`). Because we are restricting guard programs to sets of conditional rule schemata, it is unnecessary to prove that $\mathcal{R}$ will terminate. By the semantics of GP, an application of $\mathcal{R}$ either fails or results in a graph – there is no possibility of divergence or getting stuck.

The iteration proof rule [!]$_{\text{wtot}}$ is defined as its partial correctness version, with the additional premise that the iterating program – a set of rule schemata – is $\#$-decreasing under the invariant for some termination function $\#$. Note that the first premise uses $\vdash_{\text{par}}$ whereas the extensional calculus uses $\vdash_{\text{wtot}}$. This is because, for sets of rule schemata (but not arbitrary programs), $\models_{\text{par}} \{inv\}\ \mathcal{R}\ \{inv\}$ implies $\models_{\text{wtot}} \{inv\}\ \mathcal{R}\ \{inv\}$.

The proof rules in Figure 4.8 (also presented for reference in Figure B.5 of the appendix) together define a total correctness calculus – as introduced extensionally in Section 3.5 – but with E-constraints specifically as the assertions. Additionally, Figure B.6 in the appendix defines total correctness proof rules for the derived commands of GP.

The implication demanded in the extensional version of [ruleset]$_{\text{tot}}$ is realised here by the premise $\models c \Rightarrow \text{App}(\mathcal{R})$. See the discussion on validity of E-constraints in Section 4.2.1.

As for the weak total correctness calculus, there is no need for a third premise in [if]$_{\text{tot}}$, [try]$_{\text{tot}}$ since the restriction to $\mathcal{R}$ means that executions of guard programs will never lead to non-termination.

[comp], [cons] of Figure 4.6, and:

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

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

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

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

Figure 4.8: Total correctness proof rules with E-constraints for core commands

The iteration proof rule $[!]_{\text{tot}}$ is the same as $[!]_{\text{wtot}}$, since $\mathcal{R}!$ will never result in failure. Note that the first premise is not $\vdash_{\text{tot}} \{inv\}\ \mathcal{R}\ \{inv\}$ because if $\models_{\text{tot}} \{inv\}\ \mathcal{R}\ \{inv\}$ then $\mathcal{R}$ would never fail on graphs satisfying inv and $\mathcal{R}!$ would never terminate.

4.3 Transformations of E-Conditions

In this section we give formal definitions of the transformations App and Pre. They extend the basic transformations of nested conditions in [HP09], adding support for expressions and assignment constraints in E-conditions, as well as the additional features of our programming language.

4.3.1 Applicability of Sets of Rule Schemata

We define here the transformation App, which takes as input a set of rule schemata, and returns an E-constraint expressing the weakest property that a graph must satisfy for at least one rule schema in the set to be applicable to it (i.e. at least one rule schema can be applied to the graph). For a rule schema to be applicable to a graph, there must be an opportunity to apply it without violating the dangling condition, and without violating any constraints the rule schema imposes over the instantiation of variables. The definition of App makes use of two intermediate transformations, Dang and τ, which respectively address these requirements.

The transformation Dang takes as input a (conditional) rule schema $r = \langle L \Rightarrow R, \Gamma \rangle$, and returns as output an E-condition that is satisfied by morphisms $q\colon L^\alpha \hookrightarrow G$ that satisfy the dangling condition. The transformation τ takes as input a (conditional) rule schema r, returning an E-condition that is satisfied by morphisms $q\colon L^\alpha \hookrightarrow G$ and assignments α that obey the rule schema condition Γ.

The idea of transformation Dang is to generate a conjunction of negated E-conditions, each one expressing some context (e.g. an edge incident to a node that would be deleted by the rule), which if present around the image of any $q\colon L^\alpha \hookrightarrow G$ would imply that the morphism is violating the dangling condition. In other words, all the possibilities for violating the dangling condition are generated, and if q does not satisfy any of them, then q must satisfy the dangling condition.

We remark that Dang is adapted from the construction of Def in Theorem 8 of [HP09].

Definition 4.29 (Transformation Dang)**.** Let $r = \langle L \hookleftarrow K \hookrightarrow R, \Gamma \rangle$ denote a conditional rule schema. We define transformation Dang as:

$$\text{Dang}(r) = \bigwedge_{a \in A} \neg\exists a$$

where the index set A ranges over all[9] injective graph morphisms $a\colon L \hookrightarrow L^\oplus$ such that the pair $\langle K \hookrightarrow L, a \rangle$ has no natural pushout complement, and each $L^\oplus$ is a graph that can be obtained from L by adding either (1) a loop labelled by $(\mathtt{x}\ m)$, (2) a single edge between distinct nodes labelled by $(\mathtt{x}\ m)$, or (3) a single node and a non-looping edge incident to that node labelled by $(\mathtt{x}\ m)$ and $(\mathtt{y}\ n)$ respectively; in all cases, $\mathtt{x}, \mathtt{y}$ are variables distinct from each other and all variables in L, and m, n are marks. If the index set A is empty, then $\text{Dang}(r) = \mathtt{true}$. □

Example 4.30 (Transformation Dang)**.** Consider the rule schema:

$$\mathtt{reduce}(\mathtt{a}, \mathtt{b}, \mathtt{c}\colon \mathtt{list}) = \langle\ \textcircled{a}_1 \xrightarrow{\mathtt{c}} \textcircled{b}\ \Rightarrow\ \textcircled{a}_1\ \rangle.$$

Applying Dang to `reduce` yields the following E-condition:

$\text{Dang}(\mathtt{reduce})$

$= \bigwedge_{a \in A} \neg\exists a$

$= \neg\exists(\ \textcircled{a}_1 \xrightarrow{\mathtt{c}} \textcircled{b}_2 \hookrightarrow \textcircled{a}_1 \xrightarrow{\mathtt{c}} \textcircled{b}_2 \overset{\mathtt{x}}{\frown}\ ) \wedge \neg\exists(\ \textcircled{a}_1 \xrightarrow{\mathtt{c}} \textcircled{b}_2 \hookrightarrow \textcircled{a}_1 \xrightarrow{\mathtt{c}} \textcircled{b}_2 \overset{\mathtt{x}}{\frown}\ )$

$\wedge \neg\exists(\ \textcircled{a}_1 \xrightarrow{\mathtt{c}} \textcircled{b}_2 \hookrightarrow \textcircled{a}_1 \xrightarrow{\mathtt{c}} \textcircled{b}_2 \xrightarrow{\mathtt{y}} \textcircled{x}\ )$

$\wedge \neg\exists(\ \textcircled{a}_1 \xrightarrow{\mathtt{c}} \textcircled{b}_2 \hookrightarrow \textcircled{a}_1 \xrightarrow{\mathtt{c}} \textcircled{b}_2 \xleftarrow{\mathtt{y}} \textcircled{x}\ )$

$\wedge \neg\exists(\ \textcircled{a}_1 \xrightarrow{\mathtt{c}} \textcircled{b}_2 \hookrightarrow \textcircled{a}_1 \xrightarrow{\mathtt{c}} \textcircled{b}_2 \overset{\mathtt{x}}{\circlearrowright}\ )$

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

(Actually, in general, there are further conjuncts where the new edge and nodes are marked and unmarked in all combinations. We omit them here for simplicity, following the convention of Remark 4.1.)

Observe that the E-condition is expressing properties not about arbitrary graphs, but about morphisms with instantiations of the left graph of `reduce` as their domain. Such morphisms will only satisfy the E-condition if node 2 – which `reduce` would delete – is not incident to any edges. That is, if the morphism satisfies the dangling condition. □

The idea of transformation τ is to encode the rule schema condition using the assignment constraints and Boolean connectives of E-conditions – the morphisms of which are simply identity morphisms on the left-hand graph of the rule schema. The exception is the `edge` predicate, which is represented in an E-condition by a new edge in the codomain of the morphism.

Definition 4.31 (Transformation τ)**.** Let $r = \langle L \hookleftarrow K \hookrightarrow R \rangle$ denote a rule schema (i.e. no condition). We define the transformation τ in this case as:

$$\tau(r) = \texttt{true}$$

Let $r = \langle L \hookleftarrow K \hookrightarrow R, \Gamma \rangle$ denote a conditional rule schema. We define the transformation τ in this case as:

$$\tau(r) = \tau'(L, \Gamma)$$

where $\tau'(L, \Gamma)$ is defined inductively as follows (see Figure 2.15 for the syntax of rule schema conditions). If Γ has the form $t\,(l_1)$, $l_1 = l_2$, $l_1 \mathrel{\backslash=} l_2$, or $i_1 \bowtie i_2$ with t in Type, l_1, l_2 in List, i_1, i_2 in Integer, and $\bowtie$ in IntRel, then $\tau'(L, \Gamma) = \exists(L \hookrightarrow L \mid \Gamma)$.

If Γ has the form $\texttt{edge}\,(n_1, n_2)$ with n_1, n_2 in Node, then $\tau'(L, \Gamma) = \exists(L \hookrightarrow L_u^+) \vee \exists(L \hookrightarrow L_m^+)$ where L_u^+ (resp. L_m^+) is a graph equal to L, except for an additional unmarked (resp. marked) edge whose source is the node with identifier n_1, whose target is the node with identifier n_2, and whose label is a variable distinct from all others in L. The case when Γ has the form $\texttt{edge}\,(n_1, n_2, l)$ with l in List is analogous, except that the list component of the label of the new edge is l.

If Γ has the form $\texttt{not}\ c$ with c in Condition, then $\tau'(L, \Gamma) = \neg\tau'(L, c)$. If Γ has the form $c_1\ \texttt{and}\ c_2$ (resp. $c_1\ \texttt{or}\ c_2$) with c_1, c_2 in Condition, then $\tau'(L, \Gamma) = \tau'(L, c_1) \wedge \tau'(L, c_2)$ (resp. $\tau'(L, c_1) \vee \tau'(L, c_2)$). □

Example 4.32 (Transformation τ)**.** Consider the conditional rule schema:

$$\texttt{reduce2(a, b, c: int)} = \langle\ \textcircled{a}_1 \xrightarrow{\texttt{c}} \textcircled{b}\ \Rightarrow\ \textcircled{a}_1\ , \Gamma \rangle$$

where Γ = `a < b and b < c`. Applying τ to `reduce2` yields the following E-condition:

$$\begin{aligned}
\tau(\texttt{reduce2}) &= \tau'(\ \textcircled{a}_1 \xrightarrow{c} \textcircled{b}_2\ , \Gamma) \\
&= \tau'(\ \textcircled{a}_1 \xrightarrow{c} \textcircled{b}_2\ , \texttt{a} < \texttt{b}) \wedge \tau'(\ \textcircled{a}_1 \xrightarrow{c} \textcircled{b}_2\ , \texttt{b} < \texttt{c}) \\
&= \exists(\ \textcircled{a}_1 \xrightarrow{c} \textcircled{b}_2 \hookrightarrow \textcircled{a}_1 \xrightarrow{c} \textcircled{b}_2\ \mid \texttt{a} < \texttt{b}) \\
&\quad \wedge \exists(\ \textcircled{a}_1 \xrightarrow{c} \textcircled{b}_2 \hookrightarrow \textcircled{a}_1 \xrightarrow{c} \textcircled{b}_2\ \mid \texttt{b} < \texttt{c})
\end{aligned}$$

□

Now, we define the transformation App for expressing applicability of sets of (conditional) rule schemata. The idea is to express through nesting – for each rule schema – that there exists a morphism from an instantiation of the left-hand graph that satisfies the dangling condition and rule schema condition. We directly use the intermediate transformations Dang and τ in the construction.

Definition 4.33 (Transformation App)**.** Let $\mathcal{R}$ denote a set of (conditional) rule schemata. If $\mathcal{R}$ is empty, define $\mathrm{App}(\mathcal{R}) = \texttt{false}$. Otherwise, if $\mathcal{R} = \{r_1, \ldots, r_n\}$, define:

$$\mathrm{App}(\mathcal{R}) = \mathrm{app}(r_1) \vee \cdots \vee \mathrm{app}(r_n)$$

where for each (conditional) rule schema $r_i = \langle L_i \hookleftarrow K_i \hookrightarrow R_i, \Gamma_i \rangle$, define:

$$\mathrm{app}(r_i) = \exists(\emptyset \hookrightarrow L_i \mid \gamma_{r_i}, \mathrm{Dang}(r_i) \wedge \tau(r_i)).$$

Here, γ_{r_i} is a conjunction of type predicates restricting integer, string, and atom variables in r_i to their declared types. □

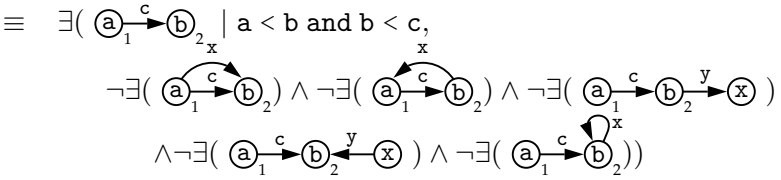
Example 4.34 (Transformation App)**.** Consider the conditional rule schema:

$$\texttt{reduce3}(\texttt{a}, \texttt{b}, \texttt{c} : \texttt{int}) = \langle\ \textcircled{a}_1 \xrightarrow{c} \textcircled{b}\ \Rightarrow\ \textcircled{a}_1\ , \Gamma \rangle$$

where $\Gamma = \texttt{a} < \texttt{b}$ and $\texttt{b} < \texttt{c}$. Applying App to reduce3 yields the following E-constraint:

$$\begin{aligned}
&\mathrm{App}(\{\texttt{reduce3}\}) \\
&\quad = \mathrm{app}(\texttt{reduce3}) \\
&\quad = \exists(\emptyset \hookrightarrow \textcircled{a}_1 \xrightarrow{c} \textcircled{b}_2\ \mid \texttt{int}(\texttt{a}, \texttt{b}, \texttt{c}), \\
&\qquad\quad \mathrm{Dang}(\texttt{reduce3}) \wedge \tau'(\ \textcircled{a}_1 \xrightarrow{c} \textcircled{b}_2\ , \Gamma)) \\
&\quad = \exists(\ \textcircled{a}_1 \xrightarrow{c} \textcircled{b}_2\ \mid \texttt{int}(\texttt{a}, \texttt{b}, \texttt{c}), \\
&\qquad\quad (\neg\exists(\ \textcircled{a}_1 \xrightarrow{c,x} \textcircled{b}_2) \wedge \neg\exists(\ \textcircled{a}_1 \xrightarrow{c,x} \textcircled{b}_2) \wedge \neg\exists(\ \textcircled{a}_1 \xrightarrow{c} \textcircled{b}_2 \xrightarrow{y} \textcircled{x}\ ) \\
&\qquad\qquad \wedge\neg\exists(\ \textcircled{a}_1 \xrightarrow{c} \textcircled{b}_2 \xleftarrow{y} \textcircled{x}\ ) \wedge \neg\exists(\ \textcircled{a}_1 \xrightarrow{c,x} \textcircled{b}_2)) \\
&\qquad\quad \wedge (\exists(\ \textcircled{a}_1 \xrightarrow{c} \textcircled{b}_2\ \mid \texttt{a} < \texttt{b}) \wedge \exists(\ \textcircled{a}_1 \xrightarrow{c} \textcircled{b}_2\ \mid \texttt{b} < \texttt{c})))
\end{aligned}$$

which can be simplified to the following equivalent E-constraint:

$\equiv$ 

(Again, as for Example 4.30, we omit for simplicity the conjuncts about marked nodes and edges in the Dang(`reduce3`) part of the E-constraint.) □

4.3.2 From Postconditions to Preconditions

In this subsection, we define the transformation Pre which takes as input a (conditional) rule schema and a postcondition, returning as output an E-constraint expressing the weakest property a graph must satisfy for successful executions of the rule schema to establish the postcondition. The transformation makes use of two intermediate transformations, A and L, which are adapted from the basic transformations of nested conditions described by Habel and Pennemann in [HP09].

The transformation A (short for "Application condition") takes as input a (conditional) rule schema and an E-constraint, and transforms the E-constraint into an equivalent E-condition over the right-hand side of the rule schema – equivalent here in the sense that a graph H satisfies the E-constraint if and only if comatches from the right-hand graph to H satisfy the E-condition generated. The transformation L (short for "Left") takes an E-condition over the right-hand graph, and shifts it along the rule schema to become an E-condition over the left-hand graph. The construction for Pre then nests this result within an E-constraint quantifying over all possible matches.

Remark 4.35 (Normalisation steps)**.** The constructions of A and L require a normalisation step for E-conditions: if an assignment constraint uses a variable not already appearing in the morphism at that level, then further down the nesting it may only appear in assignment constraints. E-conditions are easily rewritten into this form, e.g. $\exists(\emptyset \mid \texttt{atom(x)}, \exists(\,\textcircled{x}\,))$ is equivalent to $\exists(\emptyset \mid \texttt{atom(x)}, \exists(\,\textcircled{y} \mid \texttt{y = x}))$.

A further normalisation step is required for the construction of A. The graphs of E-conditions – by definition – are labelled over arbitrary expressions. We can however assume without loss of generality that nodes and edges are labelled only by distinct variables, since we can equate these variables with any other expression in the assignment constraint. For example, the E-condition:

$$\exists(\,\textcircled{\texttt{x*x}}\,)$$

can be rewritten as the equivalent E-condition $\exists(\,\textcircled{a} \mid \texttt{a = x*x})$.

□

We consider first the transformation A, adapted from Theorem 5 of [HP09]. The idea of A is to consider a disjunction of all possible "overlappings" of the right-hand graph and the graphs of the E-constraint, accounting for cases when comatches might establish (or prevent the establishment of) the postcondition, and also the cases when the images of comatches are disjoint from parts of the graph related to the postcondition. In the construction, these "overlappings" are generated by merging nodes and edges. However, since distinct labels on the syntactic level ($\mathcal{G}(\mathrm{EC})$) can be instantiated to equal labels on the semantic level ($\mathcal{G}(\mathcal{L})$), the construction in such situations applies substitutions on syntactic graphs to facilitate merging.

Definition 4.36 (Transformation A)**.** Let c denote an E-constraint normalised according to Remark 4.35, and let $r = \langle L \Rightarrow R \rangle$ denote a (conditional) rule schema sharing no variables with c[10]. We define:

$$\mathrm{A}(r, c) = \mathrm{A}'(\emptyset \hookrightarrow R, c)$$

where A' is defined inductively as follows. For injective graph morphisms $p\colon P \hookrightarrow P'$ with $P, P' \in \mathcal{G}(\mathrm{EC})$, and E-conditions over P, define:

$$\begin{aligned} \mathrm{A}'(p, \mathtt{true}) &= \mathtt{true}, \\ \mathrm{A}'(p, \exists(a\colon P \hookrightarrow C \mid \gamma, c')) &= \bigvee_{\sigma \in \Sigma} \bigvee_{e \in \varepsilon_\sigma} \exists(b \mid \gamma^\sigma, \mathrm{A}'(s, (c')^\sigma)) \end{aligned}$$

where $\Sigma, \varepsilon_\sigma, b, s$ are defined in the following. First, construct the pushout (1) of p and a (see Figure 4.9) leading to injective graph morphisms a' : $P' \hookrightarrow C'$ and $q\colon C \hookrightarrow C'$.

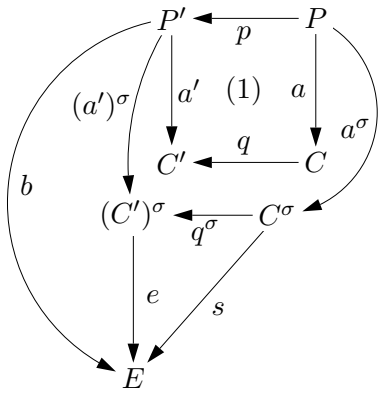


Figure 4.9: Construction of A'

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

The finite double disjunction $\bigvee_{\sigma\in\Sigma}\bigvee_{e\in\varepsilon_\sigma}$ ranges first over substitutions from the set Σ. We define this set to contain *all* possible substitutions σ such that:

$$\mathrm{dom}(\sigma) \subseteq \mathrm{vars}(C) - \mathrm{vars}(P')$$

and with images comprising lists $e \in \mathrm{List}$ such that each e is a list component in P'. (That is, all possible substitutions mapping variables in $\mathrm{vars}(C) - \mathrm{vars}(P')$ to list components of labels in P'.)

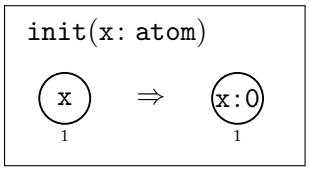
For each $\sigma \in \Sigma$, the double disjunction then ranges over every surjective graph morphism $e : (C')^\sigma \to E$ such that $b = e \circ (a')^\sigma$ and $s = e \circ q^\sigma$ are injective graph morphisms. The set ε_σ is the set of such surjective graph morphisms for a particular σ, the codomains of which we consider up to isomorphism[11], and up to the redundancy check as follows. Given a surjective graph morphism $e_1 : (C')^{\sigma_1} \to E_1$, E_1 is considered redundant and the morphism is excluded from the disjunction if there exists a surjective graph morphism $e_2 : (C')^{\sigma_2} \to E_2$, such that $E_2 \ncong E_1$, and there exists some $\sigma \in \Sigma$ such that $E_2^\sigma \cong E_1$.

Note that the definition of substitutions in Σ means that for any $\sigma \in \Sigma$, $P^\sigma = P$, and $(P')^\sigma = P'$. Note also that b and s are jointly surjective; the idea is that each E contains an image of both P' and C^σ, with the substitutions equating labels on the syntactic level and thus facilitating Es in which nodes and edges are merged.

The transformations A, A' are extended for Boolean formulae over E-conditions in the usual way, that is, $\mathrm{A}(r, \neg c) = \neg\mathrm{A}(r, c)$, $\mathrm{A}(r, c_1 \wedge c_2) = \mathrm{A}(r, c_1) \wedge \mathrm{A}(r, c_2)$, and $\mathrm{A}(r, c_1 \vee c_2) = \mathrm{A}(r, c_1) \vee \mathrm{A}(r, c_2)$ (analogous for A'). □

We demonstrate the transformation A on an example rule schema and E-constraint. This is the beginning of a running example, which will be returned to for demonstrating transformation L and then finally Pre.

Example 4.37 (Transformation A)**.** Let r denote the rule schema `init` of the program `colouring`, given in Figure 2.19 and again below:



and let c denote the E-constraint:

$$\forall(\ \textcircled{a}_1, \exists(\ \textcircled{a}_1 \mid \texttt{atom(a)}\ ) \vee \exists(\ \textcircled{a}_1 \mid \texttt{a = b:c and atom(b) and c >= 0}))$$

[11] Hence the disjunction is finite.

expressing that "every (unmarked) node is labelled by either an atom or a list comprising an atom followed by a natural number". For brevity, in what follows, we define:

$$c'_1 = \exists(\ \textcircled{\texttt{a}}_1 \mid \texttt{atom(a)}\ )$$

and:

$$c'_2 = \exists(\ \textcircled{\texttt{a}}_1 \mid \texttt{a = b:c and atom(b) and c >= 0}).$$

With the definition of $\forall$, we can take c to be the equivalent E-constraint:

$$\neg\exists(\ \textcircled{\texttt{a}}_1, \neg c'_1 \wedge \neg c'_2).$$

Now, applying transformation A to r and c, we get:

$$\begin{aligned}
&\mathrm{A}(r, c)\\
&\quad= \neg\mathrm{A}(r, \exists(\ \textcircled{\texttt{a}}_1, \neg c'_1 \wedge \neg c'_2))\\
&\quad= \neg\mathrm{A}'(\emptyset \hookrightarrow \textcircled{\texttt{x:0}}_1, \exists(\ \textcircled{\texttt{a}}_1, \neg c'_1 \wedge \neg c'_2))\\
&\quad= \neg(\textstyle\bigvee_{\sigma\in\Sigma}\bigvee_{e\in\varepsilon_\sigma} \exists(b \mid \gamma^\sigma, \mathrm{A}'(s, (\neg c'_1 \wedge \neg c'_2)^\sigma)))\\
&\quad= \neg(\exists(\textcircled{\texttt{x:0}}_1 \hookrightarrow \textcircled{\texttt{x:0}}_1\textcircled{\texttt{a}}_2, \mathrm{A}'(s_1, \neg c'_1 \wedge \neg c'_2))\\
&\qquad\vee \exists(\textcircled{\texttt{x:0}}_1 \hookrightarrow \textcircled{\texttt{x:0}}_1, \mathrm{A}'(s_2, (\neg c'_1 \wedge \neg c'_2)^{(\texttt{a}\mapsto\texttt{x:0})})))\\
&\quad= \neg(\exists(\textcircled{\texttt{x:0}}_1 \hookrightarrow \textcircled{\texttt{x:0}}_1\textcircled{\texttt{a}}_2, \neg\mathrm{A}'(s_1, c'_1) \wedge \neg\mathrm{A}'(s_1, c'_2))\\
&\qquad\vee \exists(\textcircled{\texttt{x:0}}_1 \hookrightarrow \textcircled{\texttt{x:0}}_1, \neg\mathrm{A}'(s_2, (c'_1)^{(\texttt{a}\mapsto\texttt{x:0})}) \wedge \neg\mathrm{A}'(s_2, (c'_2)^{(\texttt{a}\mapsto\texttt{x:0})})))\\
&\quad= \neg(\exists(\textcircled{\texttt{x:0}}_1 \hookrightarrow \textcircled{\texttt{x:0}}_1\textcircled{\texttt{a}}_2,\\
&\qquad\quad \neg\exists(\textcircled{\texttt{x:0}}_1\textcircled{\texttt{a}}_2 \mid \texttt{atom(a)}, \mathrm{A}'(s_{11}, \texttt{true}))\\
&\qquad\quad \wedge \neg\exists(\textcircled{\texttt{x:0}}_1\textcircled{\texttt{a}}_2 \mid \texttt{a = b:c and atom(b) and c >= 0}, \mathrm{A}'(s_{11}, \texttt{true})))\\
&\qquad\vee \exists(\textcircled{\texttt{x:0}}_1 \hookrightarrow \textcircled{\texttt{x:0}}_1,\\
&\qquad\quad \neg\exists(\textcircled{\texttt{x:0}}_1 \mid \texttt{atom(x:0)}, \mathrm{A}'(s_{21}, \texttt{true}))\\
&\qquad\quad \wedge \neg\exists(\textcircled{\texttt{x:0}}_1 \mid \texttt{x:0 = b:c and atom(b) and c >= 0}, \mathrm{A}'(s_{21}, \texttt{true}))))\\
&\quad= \forall(\textcircled{\texttt{x:0}}_1 \hookrightarrow \textcircled{\texttt{x:0}}_1\textcircled{\texttt{a}}_2,\\
&\qquad\quad \exists(\textcircled{\texttt{x:0}}_1\textcircled{\texttt{a}}_2 \mid \texttt{atom(a)}\ )\\
&\qquad\quad \vee \exists(\textcircled{\texttt{x:0}}_1\textcircled{\texttt{a}}_2 \mid \texttt{a = b:c and atom(b) and c >= 0}))\\
&\qquad\wedge \forall(\textcircled{\texttt{x:0}}_1 \hookrightarrow \textcircled{\texttt{x:0}}_1,\\
&\qquad\quad \exists(\textcircled{\texttt{x:0}}_1 \mid \texttt{atom(x:0)}\ )\\
&\qquad\quad \vee \exists(\textcircled{\texttt{x:0}}_1 \mid \texttt{x:0 = b:c and atom(b) and c >= 0}))
\end{aligned}$$

where $\Sigma = \{(), (\texttt{a} \mapsto \texttt{x:0})\}$ (here, $()$ denotes the empty substitution that replaces no variables) and the particular instances of diagrams from the

construction of A′ are as in Figure 4.10. Note that because c'_1, c'_2 contain only identity morphisms (and hence their codomains do not introduce new variables), each instance of A′(s_i, c'_j) for $i, j \in \{1, 2\}$ ranges over only one substitution: the empty substitution.

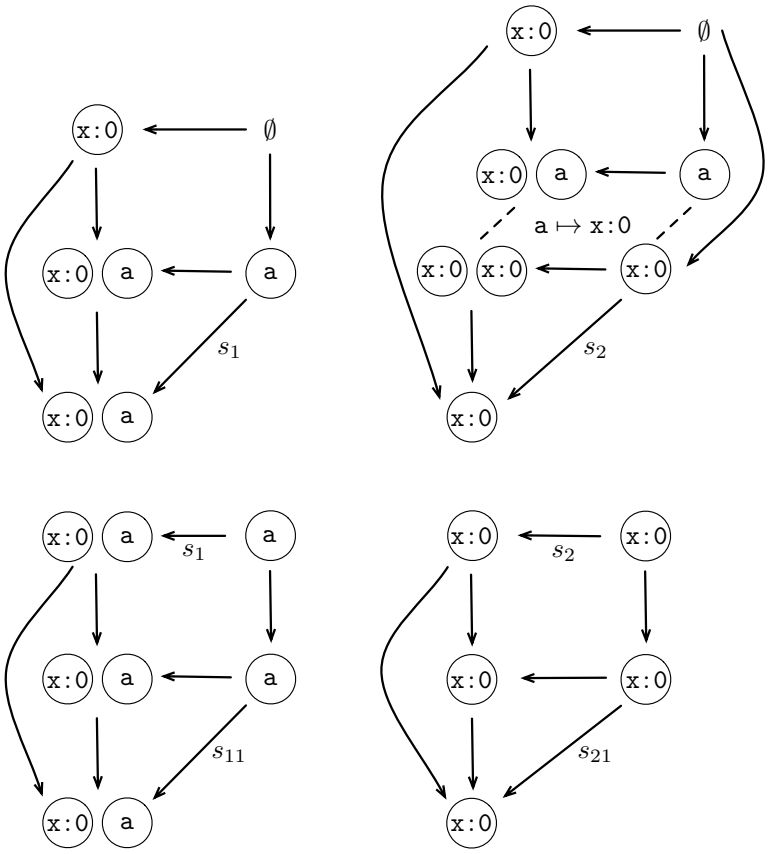


Figure 4.10: Instances of diagrams from the construction of A′

The E-condition arising from A(r, c) can be read as follows: "(1) every node that is not in the image of the right-hand graph under the the morphism is either labelled by an atom or an atom followed by a natural, and (2) the node in the image of the right-hand graph under the morphism is either labelled by an atom or an atom followed by a natural". Note that we could already apply simplifications at this stage (e.g. the disjunct $\exists(\text{x:0}_1 \mid \texttt{atom(x:0)})$ can safely be discarded since it is unsatisfiable). However, we will wait until the end of this running example (i.e. once Pre(r, c) is given) before applying any, so that the effects of the transformations can be followed more easily. □

We remark that in the worst case, transformation A can result in a factorial blow-up of the size of an E-condition. One can construct an example where graphs P' and C in Figure 4.9 both have n nodes and n edges, and

there are more than $n!$ pairwise non-isomorphic graphs E that satisfy the conditions of the construction.

Now, we define the second of the intermediate transformations, L, which we adapt from the construction of Theorem 6 in [HP09]. This takes as input an E-condition over the right-hand graph of some (conditional) rule schema, and transforms it into an E-condition over the left-hand graph. A match satisfies this new E-condition if and only if comatches satisfy the original one.

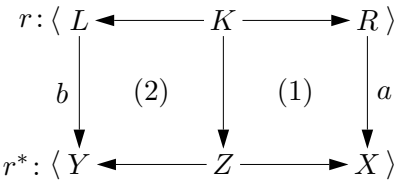
Definition 4.38 (Transformation L)**.** Let $r = \langle L \Rightarrow R \rangle$ denote a (conditional) rule schema, and let c denote an E-condition over R normalised according to Remark 4.35. The transformation $\mathrm{L}(r, c)$ is defined inductively as follows (all graphs in the construction belong to the class $\mathcal{G}(\mathrm{EC})$). We define:

$$\mathrm{L}(r, \mathtt{true}) = \mathtt{true}$$

and:

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

if $\langle K \hookrightarrow R, a \rangle$ has a natural pushout complement (1) with $r^* = \langle Y \hookleftarrow Z \hookrightarrow X \rangle$ denoting the "derived" rule by constructing natural pushout (2). If $\langle K \hookrightarrow R, a \rangle$ has no natural pushout complement, then $\mathrm{L}(r, \exists(a \mid \gamma, c')) = \mathtt{false}$.



The transformation L is extended for Boolean formulae in the usual way, that is, $\mathrm{L}(r, \neg c) = \neg\mathrm{L}(r, c)$, $\mathrm{L}(r, c_1 \wedge c_2) = \mathrm{L}(r, c_1) \wedge \mathrm{L}(r, c_2)$, and $\mathrm{L}(r, c_1 \vee c_2) = \mathrm{L}(r, c_1) \vee \mathrm{L}(r, c_2)$.

□

Note that in the construction of L, whilst the morphisms are shifted to the left-hand graph, no shift or change takes place in the assignment constraints. The idea is somewhat analogous to the axiom of assignment in classical Hoare logic: if the assignment constraint holds for the assignment determined by the match, then it will still hold after the application of the rule schema for that match.

Example 4.39 (Transformation L)**.** Recall the rule schema `init` from Example 4.37, and the E-condition over the right-hand graph of `init` given below:

$$
\begin{aligned}
&\mathrm{A}(r, c) \\
&\quad = \quad \forall(\,\textcircled{\texttt{x:0}}_1 \hookrightarrow \textcircled{\texttt{x:0}}_1\,\textcircled{\texttt{a}}_2, \\
&\qquad\quad \exists(\,\textcircled{\texttt{x:0}}_1\,\textcircled{\texttt{a}}_2 \mid \texttt{atom(a)}\,) \\
&\qquad\quad \vee\, \exists(\,\textcircled{\texttt{x:0}}_1\,\textcircled{\texttt{a}}_2 \mid \texttt{a = b:c and atom(b) and c >= 0})) \\
&\qquad \wedge\, \forall(\,\textcircled{\texttt{x:0}}_1 \hookrightarrow \textcircled{\texttt{x:0}}_1, \\
&\qquad\quad \exists(\,\textcircled{\texttt{x:0}}_1 \mid \texttt{atom(x:0)}\,) \\
&\qquad\quad \vee\, \exists(\,\textcircled{\texttt{x:0}}_1 \mid \texttt{x:0 = b:c and atom(b) and c >= 0}))
\end{aligned}
$$

Applying transformation L to the rule schema and E-condition, we get:

$$
\begin{aligned}
\mathrm{L}(r, \mathrm{A}(r, c)) \quad = \quad & \forall(\,\textcircled{\texttt{x}}_1 \hookrightarrow \textcircled{\texttt{x}}_1\,\textcircled{\texttt{a}}_2, \\
& \quad \exists(\,\textcircled{\texttt{x}}_1\,\textcircled{\texttt{a}}_2 \mid \texttt{atom(a)}\,) \\
& \quad \vee\, \exists(\,\textcircled{\texttt{x}}_1\,\textcircled{\texttt{a}}_2 \mid \texttt{a = b:c and atom(b) and c >= 0})) \\
& \wedge\, \forall(\,\textcircled{\texttt{x}}_1 \hookrightarrow \textcircled{\texttt{x}}_1, \\
& \quad \exists(\,\textcircled{\texttt{x}}_1 \mid \texttt{atom(x:0)}\,) \\
& \quad \vee\, \exists(\,\textcircled{\texttt{x}}_1 \mid \texttt{x:0 = b:c and atom(b) and c >= 0}))
\end{aligned}
$$

where the diagrams arising from applications of the construction are as given in Figure 4.11.

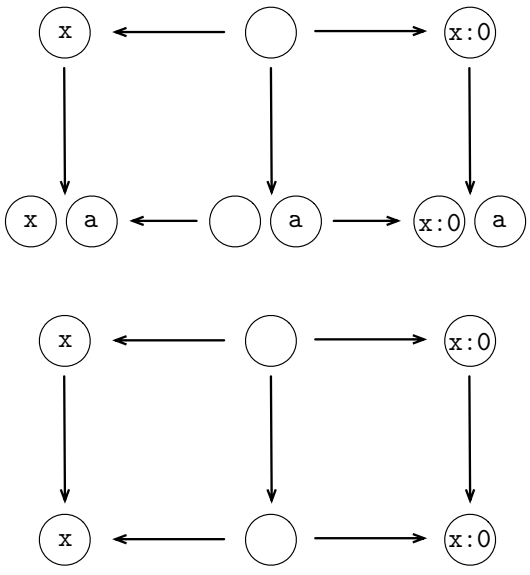


Figure 4.11: Instances of diagrams from the construction of L

□

The transformation L gives us an E-condition against which we can check particular matches. What we require – finally – for transformation Pre, is an E-constraint reasoning about all possible matches of a (conditional) rule schema in a particular graph. We want to assert that all possible matches of a rule schema in some graph have a certain property with which we can guarantee that the postcondition will hold after the rule schema is applied. Note that we assert this only for images of the left-hand graph that satisfy the dangling condition, rule schema condition, and variable types; since otherwise, the rule cannot be applied there and it does not matter whether it satisfies the properties or not.

Definition 4.40 (Transformation Pre)**.** Let c denote an E-constraint, and $r = \langle L \Rightarrow R, \Gamma \rangle$ denote a (conditional) rule schema. Define:

$$\mathrm{Pre}(r, c) = \forall(\emptyset \hookrightarrow L \mid \gamma_r, \mathrm{Dang}(r) \wedge \tau(r) \Rightarrow \mathrm{L}(r, \mathrm{A}(r, c)))$$

where γ_r is a conjunction of type predicates restricting integer, string, and atom variables in r to their declared types (as in Definition 4.33). □

Example 4.41 (Transformation Pre)**.** Continuing from Examples 4.37 and 4.39, we get:

$$\begin{aligned}
\mathrm{Pre}(r, c) &= \forall(\emptyset \hookrightarrow \textcircled{x}_1 \mid \texttt{atom(x)}, \mathrm{Dang}(r) \wedge \tau(r) \Rightarrow \mathrm{L}(r, \mathrm{A}(r, c))) \\
&= \forall(\textcircled{x}_1 \mid \texttt{atom(x)}, \mathrm{L}(r, \mathrm{A}(r, c))) \\
&= \forall(\textcircled{x}_1 \mid \texttt{atom(x)}, \\
&\quad \forall(\textcircled{x}_1\ \textcircled{a}_2, \exists(\textcircled{x}_1\ \textcircled{a}_2 \mid \texttt{atom(a)}) \\
&\quad\quad \vee \exists(\textcircled{x}_1\ \textcircled{a}_2 \mid \texttt{a = b:c and atom(b) and c >= 0})) \\
&\quad \wedge \forall(\textcircled{x}_1, \exists(\textcircled{x}_1 \mid \texttt{atom(x:0)}) \\
&\quad\quad \vee \exists(\textcircled{x}_1 \mid \texttt{x:0 = b:c and atom(b) and c >= 0})))
\end{aligned}$$

Since r does not delete any nodes, and does not have a rule schema condition, $\mathrm{Dang}(r) \wedge \tau(r) = \texttt{true}$, simplifying the nested E-constraint generated by Pre. We can simplify $\mathrm{Pre}(r, c)$ further by hand to get:

$$\begin{aligned}
\mathrm{Pre}(r, c) \quad \equiv \quad & \forall(\textcircled{x}_1\ \textcircled{a}_2 \mid \texttt{atom(x)}, \exists(\textcircled{x}_1\ \textcircled{a}_2 \mid \texttt{atom(a)}) \\
& \quad \vee \exists(\textcircled{x}_1\ \textcircled{a}_2 \mid \texttt{a = b:c and atom(b) and c >= 0}))
\end{aligned}$$

□

4.4 Soundness

In this section we prove the soundness of our calculi with E-constraints, i.e. proving a triple in a calculus for a notion of correctness implies its truth with regards to that notion of correctness.

We have already proved soundness for the extensional calculi in Theorems 3.22, 3.23, and 3.24, and so the main task that remains to prove soundness for our calculi with E-constraints is to show that:

1. $c \wedge \text{App}(\mathcal{R})$ defines $\text{SE}_{\mathfrak{C}}[c, \mathcal{R}]$;
2. $c \wedge \neg\text{App}(\mathcal{R})$ defines $\text{FE}_{\mathfrak{C}}[c, \mathcal{R}]$; and
3. $\text{Pre}(r, c) \vee \neg\text{App}(\{r\})$ defines $\text{Wlp}_{\mathfrak{C}}[r, c]$.

The soundness proof will be presented as follows. First, we will state and prove a lemma about induced substitutions and satisfaction that is useful in later proofs. Then, we will prove the correctness of App and Pre in the sense that they define the assertions in (1–3) above. Finally, the soundness of our extensional calculi and the correctness of the transformations will be used to prove the soundness of our verification calculi with E-constraints.

We remark that some of the proofs utilise basic definitions and facts about pushout and pullback constructions. The essentials are presented together in Appendix C.

4.4.1 Induced Substitutions and Satisfaction

In this subsection, we state and prove a lemma about the relationship between induced substitutions and satisfaction that is useful in later proofs. Intuitively, if a substitution induced by an assignment is defined only for variables in the graph that the E-condition is over, then the substitution can be applied (or removed if already there) without affecting satisfiability.

Lemma 4.42 (Induced substitutions and satisfaction)**.** Given an injective morphism $p : P^{\alpha} \hookrightarrow G$ with $P \in \mathcal{G}(\text{EC})$, $G \in \mathcal{G}(\mathcal{L})$, and α a well-typed assignment with $\text{dom}(\alpha) = \text{vars}(P)$, and given an E-condition c over P, we have that:

$$p \models c \text{ if and only if } p \models c^{\sigma_\alpha}.$$

□

Proof. The definition of E-conditions gives rise to three cases.

Case One. Let $c = \texttt{true}$. We have $\texttt{true}^{\sigma_\alpha} = \texttt{true}$. All morphisms satisfy `true`.

Case Two. Let $c = \exists(a\colon P \hookrightarrow C \mid \gamma, c')$.

Only if. Suppose that $p \models c$. Then there is a well-typed assignment α' such that $P^{\alpha} = P^{\alpha'}$ and $\gamma^{\alpha'} = \text{true}$, and a morphism $q\colon C^{\alpha'} \hookrightarrow G$ such that $q \circ a^{\alpha'} = p$ and $q \models (c')^{\sigma_{\alpha'}}$. Since $P^{\alpha} = P^{\alpha'}$, α' must contain at least the

mappings of α. Let σ_α be the substitution induced by α. By its definition, for every variable $\mathtt{x}$ in its domain we have that $\sigma_\alpha(\mathtt{x})^{\alpha'} = \alpha(\mathtt{x})$. Clearly, $p \models_{\alpha'} \exists(a^{\sigma_\alpha} \mid \gamma^{\sigma_\alpha})$. The assumption that $q : C^{\alpha'} \hookrightarrow G \models (c')^{\sigma_{\alpha'}}$ and the definition of induced substitutions gives us:

$$q \colon (C^{\sigma_\alpha})^{\alpha'} \hookrightarrow G \models ((c')^{\sigma_\alpha})^{\sigma_{\alpha'}}.$$

Together, and with the definition of satisfaction, we get the result that $p \models \exists(a^{\sigma_\alpha} \mid \gamma^{\sigma_\alpha}, (c')^{\sigma_\alpha}) = c^{\sigma_\alpha}$.

If. Suppose that $p \models c^{\sigma_\alpha}$. Then there is a well-typed assignment α' such that $P^\alpha = (P^{\sigma_\alpha})^{\alpha'}$ and $(\gamma^{\sigma_\alpha})^{\alpha'} = \text{true}$, and a morphism $q : (C^{\sigma_\alpha})^{\alpha'} \hookrightarrow G$ such that $q \circ (a^{\sigma_\alpha})^{\alpha'} = p$ and $q \models ((c')^{\sigma_\alpha})^{\sigma_{\alpha'}}$. By the definition of induced substitutions, for every variable $\mathtt{x}$ on which σ_α is defined, $\sigma_\alpha(\mathtt{x})^{\alpha'} = \alpha(\mathtt{x})$. If the mappings of α are added to α', we then have that $P^\alpha = P^{\alpha'}$, $\gamma^{\alpha'} = \text{true}$, and $q : C^{\alpha'} \hookrightarrow G$ such that $q \circ a^{\alpha'} = p$ and $q \models (c')^{\sigma_{\alpha'}}$. Together, and with the definition of satisfaction, we get the result that $p \models \exists(a \mid \gamma, c') = c$.

Case three. Let c denote a Boolean formula over E-conditions over P. The statement follows from the definition of satisfaction and the hypothesis. □

4.4.2 Applicability of Sets of Rule Schemata

In this subsection we will prove that App is "correct", in the sense that with negation and conjunction it defines the assertions SE and FE.

To prove the correctness of App we first prove the correctness of the two intermediate transformations it uses: Dang and τ. Recall that the former is satisfied by (potential) matches if they satisfy the dangling condition (i.e. the application of a rule schema will only delete nodes if it will also delete all edges incident to it), and that the latter is satisfied by (potential) matches if the rule schema condition evaluates to true under the morphism and assignment. We state these properties of the transformations formally and prove them.

We remark that the proof is adapted from that of Theorem 8 in [HP09].

Lemma 4.43 (Dangling condition)**.** For all (conditional) rule schemata $r = \langle L \Rightarrow R, \Gamma \rangle$, and all injective graph morphisms $q : L^\alpha \hookrightarrow G$ with α a well-typed assignment and $G \in \mathcal{G}(\mathcal{L})$,

$$q \models \text{Dang}(r) \text{ if and only if } q \text{ satisfies the dangling condition.}$$

□

Proof. Only if. Assume that $q \models \text{Dang}(r)$. By definition of $\models$ and the construction of Dang, we have $q \models \text{Dang}(r) = \bigwedge_{a \in A} \neg\exists a$ where A ranges over

morphisms $a : L \hookrightarrow L^{\oplus}$ such that $\langle K \hookrightarrow L, a\rangle$ has no (natural) pushout complement. Each $L^{\oplus}$ is obtained from L by adding either (1) a loop, (2) an edge between distinct nodes, or (3) a new node incident to a non-looping edge (i.e. the three possible ways a single edge can be added to L). It follows that there is no assignment α' and morphism $q' : (L^{\oplus})^{\alpha'} \hookrightarrow G$ with $q' \circ a^{\alpha'} = q$. The morphism q satisfies the dangling condition, because no node in the image of q that would be deleted by r, is incident to an edge in G outside of the image, i.e. the image of some edge from $L^{\oplus} - L$ in q'.

If. Assume that $q : L^{\alpha} \hookrightarrow G$ satisfies the dangling condition for r. Then the pair $\langle K^{\alpha} \hookrightarrow L^{\alpha}, q\rangle$ has a pushout complement $D \in \mathcal{G}(\mathcal{L})$. We assume that there is an $a \in A$ from the construction such that $\langle K \hookrightarrow L, a\rangle$ has no pushout complement, and some assignment α' such that $q \models_{\alpha'} \exists a$, then derive a contradiction. This assumption gives us a morphism $q' : (L^{\oplus})^{\alpha'} \hookrightarrow G$ with $q' \circ a^{\alpha'} = q$. The assignment α' is the same as α other than for having mappings for the additional variables in $L^{\oplus}$ (i.e. a variable for the extra edge to those in L, and possibly a variable for an extra node). Construct (2) (see Figure 4.12) as a pullback of $(L^{\oplus})^{\alpha'} \hookrightarrow G \hookleftarrow D$. By the universal property of pullbacks, there is a morphism $K^{\alpha} \hookrightarrow (K')^{\alpha'}$ such that the resulting diagrams commute. By the pushout-pullback decomposition, $(1) + (2)$ has a decomposition into pushouts (1) and (2), and $\langle K^{\alpha} \hookrightarrow L^{\alpha}, a^{\alpha'}\rangle$ has a pushout complement. Clearly, before the application of assignments α and α', the pair of morphisms $\langle K \hookrightarrow L, a\rangle$ has a pushout complement in $\mathcal{G}(\mathrm{EC})$. A contradiction. There is no assignment α' such that $q \models_{\alpha'} \exists a$. Hence we derive $q \models \bigwedge_{a \in A} \neg\exists a = \mathrm{Dang}(r)$, i.e. the result that $q \models \mathrm{Dang}(r)$.

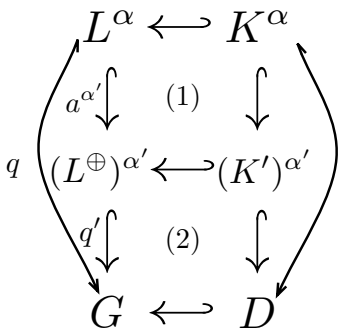


Figure 4.12: Diagram chasing for a contradiction

□

Lemma 4.44 (Rule schema condition)**.** Let $r = \langle L \Rightarrow R, \Gamma\rangle$ denote a conditional rule schema, and let α_t denote an assignment for typed variables defined for exactly the (typed) variables in L[12]. Let $q : L^{\alpha} \hookrightarrow G$ denote

[12]Note that vars(Γ) $\subseteq$ vars(L) by definition of conditional rule schemata.

an injective morphism with $G \in \mathcal{G}(\mathcal{L})$ and α a well-typed assignment such that $\alpha(\mathtt{x}) = \alpha_t(\mathtt{x})$ for every variable $\mathtt{x}$ in L. Then,

$$q \models \tau'(L, \Gamma) \text{ if and only if } \Gamma^{q,\alpha_t} = \text{true}.$$

□

Proof. Induction basis. Suppose that Γ has the form $t\,(l_1)$, $l_1 = l_2$, $l_1 \mathrel{\backslash=} l_2$, or $i_1 \bowtie i_2$ with l_1, l_2 in List, i_1, i_2 in Integer, and $\bowtie$ in IntRel. Then $\tau'(L, \Gamma) = \exists(L \hookrightarrow L \mid \Gamma)$. The assignment constraint is identical to the rule schema condition, and since $\alpha(\mathtt{x}) = \alpha_t(\mathtt{x})$ for all variables $\mathtt{x}$ in Γ, we have $\Gamma^{\alpha} = \Gamma^{q,\alpha_t}$. With the fact that $q \circ (L \hookrightarrow L)^{\alpha} = q$ the satisfaction relation $\models_{\alpha}$ holds always for q and $\exists(L \hookrightarrow L)$, and thus the result that $q \models \tau'(L, \Gamma)$ if and only if $\Gamma^{q,\alpha_t} = \Gamma^{\alpha} = \text{true}$.

Suppose that Γ has the form $\mathtt{edge}\,(n_1, n_2)$ with n_1, n_2 in Node. Then $\tau'(L, \Gamma) = \exists(a_u : L \hookrightarrow L_u^+) \vee \exists(a_m : L \hookrightarrow L_m^+)$ where L_u^+ (resp. L_m^+) is a graph equal to L, except for an additional unmarked (resp. marked) edge whose source is the node with identifier n_1, whose target is the node with identifier n_2, and whose label is a variable distinct from all others in L. Then $q \models \tau'(L, \Gamma)$ if and only if there is an assignment α' and morphism $q'_u \colon (L_u^+)^{\alpha'} \hookrightarrow G$ or $q'_m \colon (L_m^+)^{\alpha'} \hookrightarrow G$ with $q' \circ a_u^{\alpha'} = q$ or $q' \circ a_m^{\alpha'} = q$ where α' is equal to α but with an additional mapping for the new variable to the list component of an edge in G from $q(n_1)$ to $q(n_2)$. Such an edge in G exists if and only if $\Gamma^{q,\alpha_t} = \text{true}$. The case when Γ has the form $\mathtt{edge}\,(n_1, n_2, l)$ with l in List is analogous, but with the additional requirement that $q \models \tau'(L, \Gamma)$ if and only if the new edge in G has as its list component $l^{\alpha} = l^{\alpha_t}$ which is the case if and only if $\Gamma^{q,\alpha_t} = \text{true}$.

Induction step. Only if. Suppose that Γ has the form $\mathtt{not}\ c$ with c in Condition. By construction and assumption we have that $q \models \neg\tau'(L, c)$, and by definition of satisfaction, $q \not\models \tau'(L, c)$. By induction hypothesis we get that $c^{q,\alpha_t} = \text{false}$. Together with the semantics of rule schema conditions, we get the result that $\Gamma^{q,\alpha_t} = \text{true}$.

Suppose that Γ has the form $c_1\ \mathtt{and}\ c_2$ (resp. $c_1\ \mathtt{or}\ c_2$) with c_1, c_2 in Condition. By construction and assumption we have that $q \models \tau'(L, c_1) \wedge \tau'(L, c_2)$ (resp. $\tau'(L, c_1) \vee \tau'(L, c_2)$). In both cases we can get the result by applying the definition of satisfaction for Boolean formulae over E-conditions and by applying the induction hypothesis.

Induction step. If. Assuming that $\Gamma^{q,\alpha_t} = \text{true}$, one can construct a similar argument in the other direction yielding $q \models \tau'(L, \Gamma)$. □

Using the lemmata for Dang and τ, we prove that App correctly expresses what must be satisfied for a set of conditional rule schemata to be applicable to a graph.

Proposition 4.45 (Applicability of a set of rule schemata)**.** For any set of (conditional) rule schemata $\mathcal{R}$, and any graph $G \in \mathcal{G}(\mathcal{L})$,

$$G \models \mathrm{App}(\mathcal{R}) \text{ if and only if there exists a direct derivation } G \Rightarrow_{\mathcal{R}} H$$

for some graph $H \in \mathcal{G}(\mathcal{L})$. □

Proof. Define $i_G \colon \emptyset \hookrightarrow G$.

Only if. Assume that $G \models \mathrm{App}(\mathcal{R})$. By the definitions of $\models$ and App, we have that $i_G \models \mathrm{App}(\mathcal{R}) = \mathrm{app}(r_1) \vee \ldots \vee \mathrm{app}(r_n)$ with each $r_i \in \mathcal{R}$. By assumption, there is a (conditional) rule schema $r = \langle L \Rightarrow R, \Gamma \rangle$ in $\mathcal{R}$, and a well-typed assignment α such that $i_G \models_\alpha \mathrm{app}(r) = \exists(a : \emptyset \hookrightarrow L \mid \gamma_r, \mathrm{Dang}(r) \wedge \tau(r))$. There exists an injective graph morphism $q \colon L^\alpha \hookrightarrow G$ with $q \circ a^\alpha = i_G$, $q \models \mathrm{Dang}(r)^{\sigma_\alpha}$ and $q \models \tau(r)^{\sigma_\alpha}$. By Lemma 4.42, we have $q \models \mathrm{Dang}(r)$ and $q \models \tau(r)$. By Lemma 4.43, the dangling condition is satisfied by q. By assumption we have $\gamma_r^\alpha = \text{true}$, so α adheres to the declared types of variables, and so there exists an assignment α_t for typed variables such that $\alpha_t(\mathtt{x}) = \alpha(\mathtt{x})$ for all variables $\mathtt{x}$ in L. With this and Lemma 4.44, $\Gamma^{q,\alpha_t} = \text{true}$ if $\tau(r) = \tau'(L, \Gamma)$ (i.e. if there is a rule schema condition). Putting everything together, and by the semantics of rule schema application, q is a match for r. Hence there is a direct derivation $G \Rightarrow_{r,q} H$ for some graph $H \in \mathcal{G}(\mathcal{L})$. As r is in $\mathcal{R}$, we get the result that there exists a graph H such that $G \Rightarrow_{\mathcal{R}} H$.

If. Assume that there exists a graph H such that $G \Rightarrow_{\mathcal{R}} H$. Then there is a rule schema $r \in \mathcal{R}$ such that $G \Rightarrow_r H$. Hence there is some instantiation of the typed variables in L by a typed assignment α_t that gives a match $q \colon L^{\alpha_t} \hookrightarrow G$ for r, and $\Gamma^{q,\alpha_t} = \text{true}$ if there is a rule schema condition Γ. In this case, by Lemma 4.44, we have $q \models \tau'(L, \Gamma)$, and then with Lemma 4.42 get $q \models \tau'(L, \Gamma)^{\sigma_{\alpha_t}} = \tau(r)^{\sigma_{\alpha_t}}$. In the case when r has no condition, this satisfaction relation still holds since $\tau(r) = \mathtt{true}$. The morphism q is guaranteed to satisfy the dangling condition since direct derivations are constructed with two natural pushouts. With Lemma 4.43 this gives us that $q \models \mathrm{Dang}(r)$. Since α_t is defined only for variables in L (variables appearing in Γ must also appear in L, by the definition of rule schema conditions), we get $q \models \mathrm{Dang}(r)^{\sigma_{\alpha_t}}$. By the definition of $\models$, we get $q \models \mathrm{Dang}(r)^{\sigma_{\alpha_t}} \wedge \tau(r)^{\sigma_{\alpha_t}}$. From the construction we have that γ_r restricts the instantiations of variables to the types that were declared in r, so clearly we have that $\gamma_r^{\alpha_t} = \text{true}$. Bringing this all together, we have that $i_G \models_{\alpha_t} \mathrm{app}(r) = \exists(\emptyset \hookrightarrow L \mid \gamma_r, \mathrm{Dang}(r) \wedge \tau(r))$ since $q \circ (\emptyset \hookrightarrow L^{\alpha_t}) = i_G$ and by definition of $\models$, we get $G \models \mathrm{app}(r)$. As $\mathrm{app}(r)$ is a disjunct of $\mathrm{App}(\mathcal{R})$, by the definition of disjunction we get the result that $G \models \mathrm{App}(\mathcal{R})$. □

With this proposition about the transformation App, it is now easy to

show how – with conjunction and negation – it defines exactly SE and FE for the assertion language of E-constraints.

Theorem 4.46 (App and conjunction defines SE)**.** For any set of (conditional) rule schemata $\mathcal{R}$, and any E-constraint c,

$$c \wedge \text{App}(\mathcal{R}) \text{ defines } \text{SE}_{\mathfrak{E}}[c, \mathcal{R}].$$

□

Proof. Suppose that some G satisfies $c \wedge \text{App}(\mathcal{R})$. By definition of conjunction, this is the case if and only if $G \models c$ and $G \models \text{App}(\mathcal{R})$. By Proposition 4.45, $G \models \text{App}(\mathcal{R})$ if and only if $G \Rightarrow_{\mathcal{R}} H$ for some $H \in \mathcal{G}(\mathcal{L})$, or by the semantics of rule schema sets, there is some $H \in [\![\mathcal{R}]\!]G$. Together we get that $G \models c \wedge \text{App}(\mathcal{R})$ if and only if ($G \models c$ and $H \in [\![\mathcal{R}]\!]G$ for some graph H). The E-constraint $c \wedge \text{App}(\mathcal{R})$ defines $\text{SE}_{\mathfrak{E}}[c, \mathcal{R}]$ as characterised by Definition 3.11. □

Theorem 4.47 (Negated App and conjunction defines FE)**.** For any set of (conditional) rule schemata $\mathcal{R}$, and any E-constraint c,

$$c \wedge \neg\text{App}(\mathcal{R}) \text{ defines } \text{FE}_{\mathfrak{E}}[c, \mathcal{R}].$$

□

Proof. Suppose that some G satisfies $c \wedge \neg\text{App}(\mathcal{R})$. By definition of conjunction, this is the case if and only if $G \models c$ and $G \models \neg\text{App}(\mathcal{R})$. By definition of negation, G does not satisfy $\text{App}(\mathcal{R})$. By Proposition 4.45, $G \not\models \text{App}(\mathcal{R})$ if and only if $G \not\Rightarrow_{\mathcal{R}}$, and by the semantics of rule schema sets, fail $\in [\![\mathcal{R}]\!]G$. Together we get that $G \models c \wedge \neg\text{App}(\mathcal{R})$ if and only if ($G \models c$ and fail $\in [\![\mathcal{R}]\!]G$). The E-constraint $c \wedge \neg\text{App}(\mathcal{R})$ defines $\text{FE}_{\mathfrak{E}}[c, \mathcal{R}]$ as characterised by Definition 3.12. □

4.4.3 From Postconditions to Preconditions

In this subsection we will prove that $\text{Pre}(r, c)$ is correct, in the sense that with $\neg\text{App}(\{r\})$ it defines the assertion $\text{Wlp}_{\mathfrak{E}}[r, c]$.

To prove the correctness of Pre, we must first prove the correctness of the two intermediate transformations it uses: A and L. Recall that the former transforms an E-constraint into an E-condition over the right-hand graph of a rule schema, and that the latter transformation then shifts it along to be over the left-hand graph of the rule schema. We state propositions about A and L first and prove them, before proving the correctness of Pre.

Proposition 4.48 (From E-constraints to E-conditions over R)**.** Let c be an E-constraint normalised according to Remark 4.35. For all (conditional) rule schemata $r = \langle L \Rightarrow R, \Gamma \rangle$ sharing no variables with c[13], and all injective graph morphisms $h\colon R^\alpha \hookrightarrow H$ with $H \in \mathcal{G}(\mathcal{L})$ and α a well-typed assignment with $\mathrm{dom}(\alpha) = \mathrm{vars}(L)$,

$$h \models \mathrm{A}(r, c) \text{ if and only if } H \models c.$$

□

In order to prove Proposition 4.48, we first prove a lemma stating that an E-condition c over P can be shifted along an injective graph morphism $p : P \hookrightarrow P'$ with $P, P' \in \mathcal{G}(\mathrm{EC})$. The proof is adapted from the proof of Lemma 3 in [HP09]. On the one hand, it is simplified since we consider only injective graph morphisms in our E-conditions, but on the other hand, it is made more complicated by the separation of graphs over the syntactic and semantic label alphabets.

Lemma 4.49 (Shifting E-conditions over morphisms)**.** Let $p\colon P \hookrightarrow P'$ and $p''\colon (P')^\alpha \hookrightarrow H$ denote injective morphisms with $P, P' \in \mathcal{G}(\mathrm{EC})$ and $H \in \mathcal{G}(\mathcal{L})$. Let c denote an E-condition over P in which the nodes and edges of each graph (except those also in P) are labelled by list components over distinct variables, and in which assignment constraints are normalised according to Remark 4.35. Let α denote a well-typed assignment that is undefined for variables only appearing in the nesting of c, and $\mathrm{vars}(C) - \mathrm{vars}(P') \not\subseteq \mathrm{dom}(\alpha)$ for all codomains C of the outermost morphisms of c. Then, we have that:

$$p'' \models \mathrm{A}'(p, c)^{\sigma_\alpha} \text{ if and only if } p'' \circ p^\alpha \models c^{\sigma_\alpha}.$$

□

Proof of Lemma 4.49. Induction basis. Let $c = \mathtt{true}$. Then we have $p'' \models \mathrm{A}'(p, \mathtt{true})^{\sigma_\alpha} = \mathtt{true}$ and $p'' \circ p^\alpha \models \mathtt{true}$. All morphisms satisfy $\mathtt{true}$.

Induction step. Let $c = \exists(a : P \hookrightarrow C \mid \gamma, c')$. Figure 4.13 contains the diagram from the construction before and after the application of well-typed assignments α and α' in the following.

Only if. Assume that $p'' \models \mathrm{A}'(p, c)^{\sigma_\alpha}$, i.e.

$$p'' \models \mathrm{A}'(p, c)^{\sigma_\alpha} = \bigvee_{\sigma \in \Sigma} \bigvee_{e \in \varepsilon_\sigma} \exists(b\colon P' \hookrightarrow E \mid \gamma^\sigma, \mathrm{A}'(s\colon C^\sigma \hookrightarrow E, (c')^\sigma))^{\sigma_\alpha}.$$

There exists at least one $\sigma \in \Sigma$ and one $e \in \varepsilon_\sigma$ such that:

$$p''\colon (P')^\alpha \hookrightarrow H \models \exists(b\colon P' \hookrightarrow E \mid \gamma^\sigma, \mathrm{A}'(s, (c')^\sigma))^{\sigma_\alpha}.$$

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

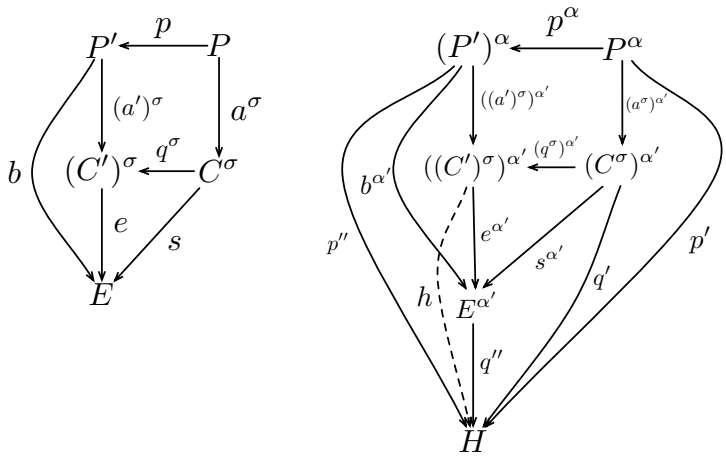


Figure 4.13: Instantiating the construction with assignments

By definition of $\models$ and induced substitutions, there exists a well-typed assignment α' with $\mathrm{dom}(\alpha) \subseteq \mathrm{dom}(\alpha')$, such that:

$$p'' \models_{\alpha'} \exists(b \mid \gamma^\sigma, \mathrm{A}'(s, (c')^\sigma)).$$

By definition of $\models_{\alpha'}$ and the fact that $\mathrm{vars}(P') \subseteq \mathrm{dom}(\alpha) \subseteq \mathrm{dom}(\alpha')$, there exists an injective graph morphism $q'' \colon E^{\alpha'} \hookrightarrow H$ with $p'' = q'' \circ b^{\alpha'}$. Define $q' = q'' \circ s^{\alpha'}$ and $p' = p'' \circ p^\alpha$, both of which are injective since injectivity is closed under composition. By construction, $a' \circ p = q \circ a$ is a pushout. Since σ only replaces variables introduced in C, and thus also present in C' but not P or P', we have that $P^\sigma = P$, $(P')^\sigma = P'$, and $(a')^\sigma \circ p = q^\sigma \circ a^\sigma$ is a pushout. Clearly, applying α' to the morphisms of this pushout results in a pushout of graphs from $\mathcal{G}(\mathcal{L})$. By construction and application of assignments, we have $b^{\alpha'} = e^{\alpha'} \circ ((a')^\sigma)^{\alpha'}$ and $s^{\alpha'} = e^{\alpha'} \circ (q^\sigma)^{\alpha'}$. With everything together,

$$\begin{aligned} p'' \circ p^\alpha &= q'' \circ b^{\alpha'} \circ p^\alpha \\ &= q'' \circ e^{\alpha'} \circ ((a')^\sigma)^{\alpha'} \circ p^\alpha \\ &= q'' \circ e^{\alpha'} \circ (q^\sigma)^{\alpha'} \circ (a^\sigma)^{\alpha'} \\ &= q'' \circ s^{\alpha'} \circ (a^\sigma)^{\alpha'} \\ &= q' \circ (a^\sigma)^{\alpha'} \\ &= p' \end{aligned}$$

and thus $p' \colon P^\alpha \hookrightarrow H \models_{\alpha'} \exists(a^\sigma \colon P \hookrightarrow C^\sigma \mid \gamma^\sigma)$.

By assumption we have:

$$q'' \colon E^{\alpha'} \hookrightarrow H \models \mathrm{A}'(s \colon C^\sigma \hookrightarrow E, (c')^\sigma)^{\sigma_{\alpha'}}$$

and by induction hypothesis:

$$q' \colon (C^{\sigma})^{\alpha'} \hookrightarrow H \models ((c')^{\sigma})^{\sigma_{\alpha'}}.$$

With the definition of $\models$ we get that:

$$p' \models_{\alpha'} \exists(a^{\sigma} \colon P \hookrightarrow C^{\sigma} \mid \gamma^{\sigma}, (c')^{\sigma}).$$

Since α' has at least the mappings of α, α is undefined for variables only occurring within $(c')^{\sigma}$, and σ does not introduce variables not already in P', we have that:

$$p' \models_{\alpha'} \exists(a^{\sigma} \colon P \hookrightarrow C^{\sigma} \mid \gamma^{\sigma}, (c')^{\sigma})^{\sigma_{\alpha}}.$$

Finally, note first that $\mathrm{dom}(\sigma) \subseteq \mathrm{vars}(C) - \mathrm{vars}(P') \not\subseteq \mathrm{dom}(\alpha)$, and secondly, that one can always define an assignment that has the net effect of first applying substitution σ and then α'. Then, we get the result that:

$$p' = p'' \circ p^{\alpha} \models \exists(a \mid \gamma, c')^{\sigma_{\alpha}}.$$

If. Assume that $p'' \circ p^{\alpha} \models c^{\sigma_{\alpha}}$, i.e.

$$p'' \circ p^{\alpha} \models \exists(a \colon P \hookrightarrow C \mid \gamma, c')^{\sigma_{\alpha}}.$$

By definition of $\models$ and induced substitutions, there exists a well-typed assignment α' with $\mathrm{dom}(\alpha) \subseteq \mathrm{dom}(\alpha')$, such that:

$$p'' \circ p^{\alpha} \models_{\alpha'} \exists(a \colon P \hookrightarrow C \mid \gamma, c').$$

Define $p' = p'' \circ p^{\alpha}$, which is injective since injectivity is closed under composition. By the definition of $\models_{\alpha'}$, there exists an injective graph morphism $C^{\alpha'} \hookrightarrow H$ with $(C^{\alpha'} \hookrightarrow H) \circ a^{\alpha'} = p'$. Consider substitutions $\sigma \in \Sigma$ where $(\gamma^{\sigma})^{\alpha'} = \text{true}$, and injective graph morphisms $q' : (C^{\sigma})^{\alpha'} \hookrightarrow H$ with $q' \circ (a^{\sigma})^{\alpha'} = p'$ and $q' \models ((c')^{\sigma})^{\sigma_{\alpha'}}$ (α' has mappings for additional variables introduced by σ as by construction they must be in the domain of p''). At least one such morphism is guaranteed to exist (i.e. if σ is the empty substitution, by assumption).

From the construction yield pushouts $q^{\sigma} \circ a^{\sigma} = (a')^{\sigma} \circ p$ with pushout objects $(C')^{\sigma}$. Clearly, the morphisms under the assignment α' yield pushouts $(q^{\sigma})^{\alpha'} \circ (a^{\sigma})^{\alpha'} = ((a')^{\sigma})^{\alpha'} \circ p^{\alpha}$. By the universal property of pushouts, for each of the pushouts there is a unique morphism $h : ((C')^{\sigma})^{\alpha'} \to H$ with $p'' = h \circ ((a')^{\sigma})^{\alpha'}$ and $q' = h \circ (q^{\sigma})^{\alpha'}$. Consider $y \circ x = h$, a surjective-injective factorisation of h with $x : ((C')^{\sigma})^{\alpha'} \to X$ surjective, $y : X \hookrightarrow H$ injective, $X \in \mathcal{G}(\mathcal{L})$, and injective morphisms $t = x \circ (q^{\sigma})^{\alpha'}$ and $u = x \circ ((a')^{\sigma})^{\alpha'}$. Now, we argue that for whichever $X \in \mathcal{G}(\mathcal{L})$ is obtained from the factorisation, the construction yields a graph $E \in \mathcal{G}(\mathrm{EC})$ such that $E^{\alpha'} \cong X$.

Suppose that x is an injective morphism, and hence an isomorphism since it is also surjective, i.e. $((C')^{\sigma})^{\alpha'} \cong X$. The construction yields an isomorphism, i.e. $E \cong C^{\sigma}$. It follows that $E^{\alpha'} \cong X$. Suppose now that x is non-injective, i.e. some nodes (edges) in $(C^{\sigma})^{\alpha'}$ are merged. Since t, u are injective, the images of $(P')^{\alpha}$ and $(C^{\sigma})^{\alpha'}$ in X must overlap. Hence, at least one variable in $C^{\alpha'}$ must be instantiated to some list in $(P')^{\alpha}$ such that node (or edge) labels in the two graphs under α' are equal. These nodes (or edges) may not be possible to merge at the syntactic level (e.g. `x*x` and `a` could evaluate to the same integer but are distinct expressions). However, for non-empty $\sigma \in \Sigma$, a variable `a` in C can be replaced by a list in P' such that $\alpha'(\mathtt{a}) = \sigma(\mathtt{a})^{\alpha'}$. (Note in particular that the labels in $C - P$, to which σ is applied, are simply distinct variables and not composite expressions.) Now, with P' and C sharing at least one label, the construction yields surjective morphisms $e : C^{\sigma} \to E$ where $E \ncong C^{\sigma}$ and $E^{\alpha'} \cong X$. The construction gives us $s = e \circ q^{\sigma}$ and $b = e \circ (a')^{\sigma}$. It follows that $e^{\alpha'}, q'', s^{\alpha'}$, and $b^{\alpha'}$ are equal to x, y, t, and u up to isomorphism.

In all cases,

$$\begin{aligned} p'' &= h \circ ((a')^{\sigma})^{\alpha'} \\ &= q'' \circ e^{\alpha'} \circ ((a')^{\sigma})^{\alpha'} \\ &= q'' \circ b^{\alpha'} \end{aligned}$$

giving us:

$$p'' \models_{\alpha'} \exists(b \mid \gamma^{\sigma}).$$

In each case the assumption give us:

$$q' : (C^{\sigma})^{\alpha'} \hookrightarrow H \models ((c')^{\sigma})^{\sigma_{\alpha'}}$$

and by induction hypothesis:

$$q'' : E^{\sigma} \hookrightarrow H \models \mathrm{A}'(s, (c')^{\sigma})^{\sigma_{\alpha'}}.$$

With the definition of satisfaction, we get:

$$p'' \models_{\alpha'} \exists(b \mid \gamma^{\sigma}, \mathrm{A}'(s, (c')^{\sigma})).$$

With the definition of $\models$ and induced substitutions, and since α' has at least the mappings of α, we get:

$$p'' \models \exists(b \mid \gamma^{\sigma}, \mathrm{A}'(s, (c')^{\sigma}))^{\sigma_{\alpha}}$$

and finally, with the definition of $\models$ for Boolean formulae over E-conditions, we get the result that:

$$p'' \models \bigvee_{\sigma\in\Sigma} \bigvee_{e\in\varepsilon_\sigma} \exists(b \mid \gamma^\sigma, \mathrm{A}'(s,(c')^\sigma))^{\sigma_\alpha} = \mathrm{A}'(p,c)^{\sigma_\alpha}.$$

When considering Boolean formulae over E-conditions, the statement follows from the definition and induction hypothesis. □

Now, we can easily prove Proposition 4.48 since it is simply an instance of Lemma 4.49.

Proof of Proposition 4.48. From the construction of A in Definition 4.36, the statement of Lemma 4.49, Lemma 4.42[14], and that $\mathrm{dom}(\alpha) \cap \mathrm{vars}(c) = \emptyset$, we get:

$$\begin{aligned} h \models \mathrm{A}(r,c) &\quad \text{if and only if} \\ h \models \mathrm{A}(r,c)^{\sigma_\alpha} &\quad \text{iff} \\ h \models \mathrm{A}'(i_R,c)^{\sigma_\alpha} &\quad \text{iff} \\ h \circ i_R^\alpha \models c^{\sigma_\alpha} &\quad \text{iff} \\ i_H\colon \emptyset^\alpha \hookrightarrow H \models c^{\sigma_\alpha} &\quad \text{iff} \\ i_H\colon \emptyset \hookrightarrow H \models c &\quad \text{iff} \\ H \models c. & \end{aligned}$$

□

Proposition 4.50 (E-conditions over R to over E-conditions over L)**.** For every (conditional) rule schema $r = \langle L \hookleftarrow K \hookrightarrow R, \Gamma\rangle$, every E-condition c over R normalised according to Remark 4.35, 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 with $\mathrm{dom}(\alpha) \subseteq \mathrm{vars}(L) \cup \mathrm{vars}(\gamma)$ for outermost assignment constraints γ in c,

$$g \models \mathrm{L}(r,c)^{\sigma_\alpha} \text{ if and only if } h \models c^{\sigma_\alpha}.$$

□

[14] If $\mathrm{vars}(R) \subset \mathrm{vars}(L)$, we make the assumption that $\mathrm{dom}(\alpha) = \mathrm{vars}(R)$ without loss of generality, since no additional variables in L appear in c, nor can they be introduced by the construction.

Our proof of the proposition adapted from the proof of Theorem 6 in [HP09]. As before, the proof is simplified by the restriction to injective graph morphisms in E-conditions, but is made more complicated by the separation of graphs over the syntactic and semantic label alphabets, the use of partially labelled graphs, and the use of natural pushouts.

Proof. We prove the proposition using structural induction.
Induction basis. Let $c = \texttt{true}$. By construction, we get

$$\mathrm{L}(r, c)^{\sigma_\alpha} = \mathrm{L}(r, \texttt{true})^{\sigma_\alpha} = \texttt{true}.$$

We have $g \models \texttt{true}$ and $h \models \texttt{true}$. All morphisms satisfy `true`.

Induction step. For an E-condition $c = \exists(a \mid \gamma, c')$ over R, the construction distinguishes two cases. On both sides of the statement, we will use α' to denote the satisfying assignment, which we assume to be minimal (in that it is undefined for variables not appearing in the E-conditions), and also that $\mathrm{dom}(\alpha) \subseteq \mathrm{dom}(\alpha')$. With the definition of induced substitutions, and for E-conditions of this form, it is sufficient to show:

$$g \models_{\alpha'} \mathrm{L}(r, \exists(a \mid \gamma, c')) \text{ if and only if } h \models_{\alpha'} \exists(a \mid \gamma, c').$$

Let l and s denote the inclusions $K \hookrightarrow L$ and $K \hookrightarrow R$, respectively. Let (1) and (2) denote the natural pushouts of the construction, and $(1)^{\alpha'}$ and $(2)^{\alpha'}$ denote the same diagrams but after the application of the well-typed assignment α' to the morphisms (as in Figure 4.14).

$$\begin{array}{ccccc} L & \xhookleftarrow{l} & K & \xhookrightarrow{s} & R \\ {\scriptstyle b}\downarrow & (2) & \downarrow & (1) & \downarrow{\scriptstyle a} \\ Y & \hookleftarrow & Z & \hookrightarrow & X \end{array} \qquad \begin{array}{ccccc} L^\alpha & \xhookleftarrow{l^\alpha} & K^\alpha & \xhookrightarrow{s^\alpha} & R^\alpha \\ {\scriptstyle b^{\alpha'}}\downarrow & (2)^{\alpha'} & \downarrow & (1)^{\alpha'} & \downarrow{\scriptstyle a^{\alpha'}} \\ Y^{\alpha'} & \hookleftarrow & Z^{\alpha'} & \hookrightarrow & X^{\alpha'} \end{array}$$

Figure 4.14: Instantiating the construction with assignments

Case one. The morphisms $\langle s, a \rangle$ have a natural pushout complement. By construction, we have $\mathrm{L}(r, \exists(a \mid \gamma, c')) = \exists(b \mid \gamma, \mathrm{L}(r^*, c'))$ where $b \colon L \hookrightarrow Y$ and $r^* = \langle Y \hookleftarrow Z \hookrightarrow X \rangle$.

A. First, we show that given an injective graph morphism $q' \colon Y^{\alpha'} \hookrightarrow G$ with $q' \circ b^{\alpha'} = g$, there is a decomposition of the pushouts (see Figure 4.15) which yields the injective graph morphism $q \colon X^{\alpha'} \hookrightarrow H$ with $q \circ a^{\alpha'} = h$ (required to show that $h \models_{\alpha'} c$ holds). By Lemma C.5 there is a pullback of q' and $D \hookrightarrow G$, obtaining the pullback object $F \in \mathcal{G}(\mathcal{L}_\perp)$. By the universal property of (injective) pullbacks, there is a unique (injective) graph morphism $K^\alpha \hookrightarrow F$ such that the arising diagrams commute. By the pushout-pullback decomposition of

Lemma C.9, $(2')$ and $(4')$ are both natural pushouts. By construction $b^{\alpha'}$ must satisfy the dangling condition with respect to l^{α}, and so by Lemma C.7 a natural pushout complement is unique up to isomorphism. Hence $F \cong Z^{\alpha'}$ and $(2')$ is equal to $(2)^{\alpha'}$ from the construction (up to isomorphism).

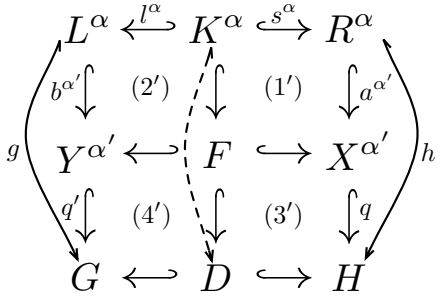


Figure 4.15: Decomposing a rule application

Now construct the natural pushout $(1')$ of $K^\alpha \hookrightarrow F$ and s^α. By the uniqueness of natural pushout complements, $(1')$ equals $(1)^{\alpha'}$ up to isomorphism. By the universal property of pushouts, there is a unique morphism $q\colon X^{\alpha'} \to H$ with $q \circ a^{\alpha'} = h$. By pushout decomposition (Lemma C.8), diagram $(3')$ is also a pushout. Since q' and hence $F \hookrightarrow D$ are injective, it follows that q is also injective.

B. Given an injective graph morphism $q\colon X^{\alpha'} \hookrightarrow H$ with $q \circ a^{\alpha'} = h$, one can yield $q'\colon Y^{\alpha'} \hookrightarrow G$ with $q' \circ b^{\alpha'} = g$ by instantiating $(1), (2)$ into $(1)^{\alpha'}, (2)^{\alpha'}$, and decomposing these into $(1')$–$(4')$ as above, i.e. start by constructing $(3')$ as a pullback of q and $D \hookrightarrow H$.

C. The induction hypothesis states that $q'\colon Y^{\alpha'} \hookrightarrow G \models \mathrm{L}(r^*, c')^{\sigma_{\alpha'}}$ if and only if $q\colon X^{\alpha'} \hookrightarrow H \models (c')^{\sigma_{\alpha'}}$. Together with the definitions of L, $\models$, and the statements A–B above, we have:

$$
\begin{aligned}
& g \models_{\alpha'} \mathrm{L}(r, \exists(a \mid \gamma, c')) = \exists(b \mid \gamma, \mathrm{L}(r^*, c')) \\
\text{iff}\quad & \gamma^{\alpha'} = \text{true and there exists } q'\colon Y^{\alpha'} \hookrightarrow G \text{ such that } q' \circ b^{\alpha'} = g \\
& \quad \text{and } q'\colon Y^{\alpha'} \hookrightarrow G \models \mathrm{L}(r^*, c')^{\sigma_{\alpha'}} \\
\text{iff}\quad & \gamma^{\alpha'} = \text{true and there exists } q\colon X^{\alpha'} \hookrightarrow H \text{ such that } q \circ a^{\alpha'} = h \\
& \quad \text{and } q \models (c')^{\sigma_{\alpha'}} \\
\text{iff}\quad & h \models_{\alpha'} \exists(a \mid \gamma, c')
\end{aligned}
$$

Case two. The morphisms $\langle s, a \rangle$ do not have a natural pushout complement. By construction, we have $\mathrm{L}(r, \exists(a \mid \gamma, c')) = \texttt{false}$. The problem reduces to showing that $g \models \texttt{false}$ if and only if $h \models \exists(a \mid \gamma, c')^{\sigma_\alpha}$. No morphism

satisfies `false`, hence it is sufficient to argue that h does not satisfy $\exists(a \mid \gamma, c')^{\sigma_\alpha}$ by any assignment.

Assume that $h \models \exists(a \mid \gamma, c')^{\sigma_\alpha}$. Then there exists a well-typed assignment α' with $\mathrm{dom}(\alpha) \subseteq \mathrm{dom}(\alpha')$ such that $h \models_{\alpha'} \exists(a \mid \gamma, c')$, and hence an injective graph morphism $q : X^{\alpha'} \hookrightarrow H$ with $q \circ a^{\alpha'} = h$. Then, as in case one, the pushout can be decomposed into pushouts $(1')$ and $(3')$. This means that the morphisms $\langle s, a \rangle$ have a pushout complement, which contradicts the assumption.

When considering Boolean formulae over E-conditions over R, the statement follows from the definition and induction hypothesis. □

We conclude this subsection by proving that $\mathrm{Pre}(r, c)$ defines a liberal precondition, and then with $\neg\mathrm{App}(\{r\})$ defines a weakest liberal precondition $\mathrm{Wlp}_{\mathfrak{E}}[r, c]$. Since the construction of Pre uses several transformations considered already, our proof will directly use the propositions we have been proving up to this point.

Theorem 4.51 (Pre defines a liberal precondition)**.** Let c be an E-constraint, the graphs of which are labelled with list components over (sequences of) distinct variables (see Remark 4.35). For any (conditional) rule schema r, $\mathrm{Pre}(r, c)$ defines a liberal precondition in the sense that for every direct derivation $G \Rightarrow_r H$,

$$G \models \mathrm{Pre}(r, c) \text{ if and only if } H \models c.$$

□

Proof. Define $i_G : \emptyset \hookrightarrow G$.

Only if. Suppose that $G \models \mathrm{Pre}(r, c)$. Then by definition of $\models$ and the construction of Pre, there is no assignment α such that:

$$i_G \models_\alpha \exists(\emptyset \hookrightarrow L \mid \gamma_r, \mathrm{Dang}(r) \wedge \tau(r) \wedge \neg\mathrm{L}(r, \mathrm{A}(r, c)))$$

Observe that removing the conjunct $\neg\mathrm{L}(r, \mathrm{A}(r, c))$ yields the E-constraint:

$$\exists(\emptyset \hookrightarrow L \mid \gamma_r, \mathrm{Dang}(r) \wedge \tau(r)) = \mathrm{app}(r) = \mathrm{App}(\{r\})$$

i.e. an E-constraint satisfied by a graph G if there is a morphism $L^\alpha \hookrightarrow G$ for some α that is a match for r, i.e. there is a direct derivation $G \Rightarrow_r H$ (by Proposition 4.45). Hence $\mathrm{Pre}(r, c)$ expresses that there is no match for r that does not also satisfy $\mathrm{L}(r, \mathrm{A}(r, c))^{\sigma_\alpha}$, i.e. by definition of $\models$, there is no match $q : L^\alpha \hookrightarrow G \not\models \mathrm{L}(r, \mathrm{A}(r, c))^{\sigma_\alpha}$. Hence for all matches q,

$$\begin{aligned} & q\colon L^\alpha \hookrightarrow G \models \mathrm{L}(r, \mathrm{A}(r,c))^{\sigma_\alpha} \\ \text{iff} \quad & h\colon R^\alpha \hookrightarrow H \models \mathrm{A}(r,c)^{\sigma_\alpha} \\ \text{iff} \quad & h\colon R^\alpha \hookrightarrow H \models \mathrm{A}(r,c) \\ \text{iff} \quad & H \models c \end{aligned}$$

by Lemma 4.42, Proposition 4.48, and Proposition 4.50.

If. Suppose that $H \models c$ where $G \Rightarrow_{r,q,h} H$ for any pair of matches $q : L^\alpha \hookrightarrow G$ and comatches $h : R^\alpha \hookrightarrow H$ with α an assignment for typed variables. The following is true of all matches q by α. Since the variables of α correspond with the types in r, $\gamma_r^\alpha = \text{true}$. If there is a rule schema condition Γ, then $\Gamma^{q,\alpha} = \text{true}$ and so by Lemma 4.44 we have that $q \models \tau'(L,\Gamma) = \tau(r)$. (If there is no rule schema condition, then trivially $q \models \tau(r)$.) By Lemma 4.43 we have that $q \models \mathrm{Dang}(r)$.

By Lemma 4.42, Proposition 4.48, and Proposition 4.50:

$$\begin{aligned} & H \models c \\ \text{iff} \quad & h \models \mathrm{A}(r,c) \\ \text{iff} \quad & h \models \mathrm{A}(r,c)^{\sigma_\alpha} \\ \text{iff} \quad & q \models \mathrm{L}(r, \mathrm{A}(r,c))^{\sigma_\alpha} \end{aligned}$$

i.e. there is no match q that does not satisfy $\mathrm{L}(r, \mathrm{A}(r,c))^{\sigma_\alpha}$. Together, and with the definition of satisfaction and the fact that $q \circ (\emptyset \hookrightarrow L)^\alpha = i_G$,

$$i_G \models \neg\exists(\emptyset \hookrightarrow L \mid \gamma_r, \mathrm{Dang}(r) \wedge \tau(r) \wedge \neg\mathrm{L}(r, \mathrm{A}(r,c)))$$

which we rearrange and abbreviate to the result that:

$$G \models \forall(L \mid \gamma_r, \mathrm{Dang}(r) \wedge \tau(r) \Rightarrow \mathrm{L}(r, \mathrm{A}(r,c))).$$

□

Corollary 4.52 (Pre and App define a liberal precondition)**.** Let r denote a (conditional) rule schema and c and E-constraint. Then we have that $\mathrm{Pre}(r,c) \vee \neg\mathrm{App}(\{r\})$ is a liberal precondition relative to r, c. □

Theorem 4.53 (Pre and App define a weakest liberal precondition)**.** Let r denote a (conditional) rule schema, c an E-constraint, and a a liberal precondition relative to r, c. Then,

$$a \Rightarrow \mathrm{Pre}(r,c) \vee \neg\mathrm{App}(\{r\})$$

i.e. $\mathrm{Pre}(r,c) \vee \neg\mathrm{App}(\{r\})$ defines a weakest liberal precondition $\mathrm{Wlp}_{\mathfrak{C}}[r,c]$. □

Proof. Let G denote any graph in $\mathcal{G}(\mathcal{L})$ and assume that $G \models a$.

Case one. Suppose that there exists a direct derivation $G \Rightarrow_r H$ for some H. By the definition of liberal preconditions, $H \models c$, and by Theorem 4.51, $G \models \text{Pre}(r, c)$. Together and with the definition of disjunction, we get that $G \models a \Rightarrow \text{Pre}(r, c) \vee \neg\text{App}(\{r\})$.

Case two. Suppose that there does not exist a direct derivation from G via r. By Proposition 4.45, $G \models \neg\text{App}(\{r\})$ and so $G \models a \Rightarrow \text{Pre}(r, c) \vee \neg\text{App}(\{r\})$.

Hence for any graph G satisfying a liberal precondition a, we can deduce that G satisfies $\text{Pre}(r, c) \vee \neg\text{App}(\{r\})$ in both the case that r is applicable and r is not, and so we get the result that $\models a \Rightarrow \text{Pre}(r, c) \vee \neg\text{App}(\{r\})$. □

4.4.4 Soundness Theorems

Finally, we state the soundness theorems for our verification calculi with E-constraints. Soundness here follows from the soundness of our extensional calculi in Chapter 3, as well as the results about defining Wlp, SE, and FE with E-constraints.

Theorem 4.54 (Soundness for partial correctness)**.** For all (restricted) graph programs P and E-constraints c, d,

$$\vdash_{\text{par}} \{c\}\ P\ \{d\} \text{ implies } \models_{\text{par}} \{c\}\ P\ \{d\}.$$

□

Proof. The soundness of [ruleapp]$_{\text{wlp}}$, [ruleset], [comp], [if], [try], [!], and [cons] follows from the soundness of the extensional calculus for partial correctness (Theorem 3.22), and from Theorems 4.53, 4.46, and 4.47 about defining Wlp, SE, and FE.

The soundness of [ruleapp] (resp. [nonapp]) follows from that of the axiom [ruleapp]$_{\text{wlp}}$ by dropping the disjunct $\neg\text{App}(\{r\})$ (resp. dropping the disjunct $\text{Pre}(r, c)$ and setting $c = \texttt{false}$). □

Theorem 4.55 (Soundness for weak total correctness)**.** For all (restricted) graph programs P and E-constraints c, d,

$$\vdash_{\text{wtot}} \{c\}\ P\ \{d\} \text{ implies } \models_{\text{wtot}} \{c\}\ P\ \{d\}.$$

□

Proof. Follows from Theorems 3.23, 4.53, 4.46, 4.47, and from the fact that sets of (conditional) rule schemata always terminate (with a graph or failure). □

Theorem 4.56 (Soundness for total correctness)**.** For all (restricted) graph programs P and E-constraints c, d,

$$\vdash_{\text{tot}} \{c\}\ P\ \{d\} \text{ implies } \models_{\text{tot}} \{c\}\ P\ \{d\}.$$

□

Proof. Follows from Theorems 3.24, 4.53, 4.46, 4.47, and from the fact that sets of (conditional) rule schemata always terminate (with a graph or failure). □

4.5 The Completeness Problem

Having shown the soundness of our calculi with E-constraints, the natural next question to ask is whether they are also relatively complete, i.e. whether all true Hoare triples can be proven in the calculi relative to an oracle for deciding the validity of assertions.

In Section 3.6.3 we proved the relative completeness of our extensional partial correctness calculus. This proof however assumed that the assertion language was expressive, i.e. able to express weakest liberal preconditions relative to arbitrary programs and postconditions. With a proof of expressiveness for E-constraints, we would inherit the relative completeness result for the partial correctness calculus of this chapter (albeit for the restricted definition of graph programs that we use.)

E-constraints however are not expressive because nested conditions are not: there are iterated programs and postconditions for which we cannot express the weakest precondition. Example 5.22 of [Pen09] demonstrates this, requiring a weakest precondition e that expresses: "the number of nodes is even". This cannot be expressed by nested conditions, nor E-constraints; it is a non-local property.

However, the fact that E-constraints are not expressive does not necessarily mean that we do not have relative completeness. We cannot, for example, even denote e in a specification as an E-constraint, so we cannot claim that some triple $\models \{e\}\ P\ \{d\}$ is some counterexample to relative completeness! Hence it remains an open question as to whether all denotable valid triples $\models \{c\}\ P\ \{d\}$ can be proven $\vdash \{c\}\ P\ \{d\}$; we just cannot inherit relative completeness from the extensional calculi since we do not have expressiveness. (Although, an interesting item of future work would be to attempt to determine a minimum extension of E-conditions that would make it expressive: path properties, counting?)

Note that while the question of relative completeness remains open, we do still however have completeness for termination, the proof of which did not require any assumptions (e.g. expressiveness) about the assertion language.

4.6 Comparison to the Oldenburg Approach

The most closely related work to our own is that which originated from Annegret Habel's group at the Carl von Ossietzky University of Oldenburg, whilst Karl-Heinz Pennemann was working there as a Ph.D. student; hence dubbed in this thesis as the "Oldenburg[15] approach". They laid the foundations for assertional reasoning about programs based on graph transformation (see e.g. [HP09, Pen09]), contributing weakest precondition calculi for proving correctness relative to first-order structural properties. Our approach of course is adapted from theirs: in the following we make a comparison and discuss where we have gone beyond.

A first comparison to be made is in the programs we considered. Although their programs are based on the same minimal and complete core as GP [HP01], they consider graphs over finite label alphabets, they do not support the relabelling DPO approach, and do not support expressions over labels (attributes) as in the rules of GP (although they do allow nodes and edges to be selected and unselected). While Pennemann does show in Fact 4.20 of [Pen09] that relabelling of nodes can be simulated by programs in his framework, such simulations are not practical in our setting of expressions and infinite label alphabets (a rule that relabels, for example, an integer label `x` of some node to `x+1` would require an infinite number of simulations as described in Fact 4.20). A contribution we thus made beyond the Oldenburg work is the ability to reason about and prove properties not just about graph structure, but also about *data*, i.e. relations and constraints between labels drawn over an infinite label alphabet. Hence we have expanded the class of problems we can model and reason about as graph programs (many graph algorithms, for example, exploit information encoded as labels, such as colours and weights). We also argue that our approach simplifies reasoning about properties that are completely independent of labels (e.g. "the graph is non-empty"): we would write an E-condition with its graphs labelled over distinct and unconstrained variables (representing any labels in $\mathbb{L}$), as opposed to a disjunction of nested conditions over all combinations of the entire label alphabet.

A second comparison is to be made in how our program specifications are proven correct. Whereas we adopted a Hoare-style approach, Habel, Pennemann, and Rensink followed Dijkstra [Dij75, Dij76] and defined weakest precondition calculi for graph programs [HPR06]. To prove a specification $\vdash_{\text{tot}} \{c\}\ P\ \{d\}$ they require one to show the validity of $c \Rightarrow \text{Wp}(P, d)$, where Wp is a transformation from programs and postconditions to weakest preconditions. The idea is to automate as much as possible, handing the task of deciding validity over to a theorem prover [Pen08, Pen09]. In the context of finite graphs, the problem is undecidable (and not

[15] Some collaborators were otherwise affiliated, e.g. Arend Rensink.

even semidecidable): should automation then fail, we might be left with a difficult implication to try and prove (or refute) by other means. With Hoare logic we instead aim from the outset for human-guided compositional reasoning in proofs: we ultimately envisage as much of this reasoning implemented as possible (e.g. in Isabelle), with human input required for isolated, difficult parts of the proof (e.g. finding invariants). Both of our approaches of course suffer from the difficulty of finding invariants for iteration constructs. In the weakest precondition calculi, the construction for iterated programs generates an infinite conjunction of nested conditions. Indeed, it is argued that this is a necessity for expressing the weakest preconditions of such programs. We however prefer to require that loop invariants remain *finite* E-conditions – even at the expense of not being able to express certain weakest preconditions (and only approximate them) – because *infinite* E-conditions are a strictly more expressive formalism. One could, for example, express the existence of an arbitrary-length path by an infinite disjunction of E-conditions (each disjunct expressing the existence of an i-length path, for all $1 \leq i$).

A final comparison to make is in the generality of our work. Whereas our work is tailored to graphs and GP, the Oldenburg approach is presented for more abstract objects than graphs, so that their results can be applied in other graph-like settings. Our work generalises in a different way, in that our calculi are extensional and allow for user-defined assertion languages to be "plugged in", and – subject to showing certain conditions – inherit properties like soundness. In particular, our framework does not even require that assertions be logical ones: it is quite fine, for example, to use graph grammars, graph reduction specifications [BPR04b], or even graph programs themselves.

4.7 Related Work

The basic ideas of E-conditions originated in the 90s with negative application conditions and graph consistency constraints [HHT96, HW95]. Negative application conditions have the form $\neg\exists(L \rightarrow C)$ or $\neg\exists(R \rightarrow C)$, and are given with rules $r : \langle L \Rightarrow R \rangle$ to restrict the permissible contexts of matches and comatches. Graph consistency constraints have the form $\forall(\emptyset \rightarrow C, \exists(C \rightarrow C'))$. Rensink in [Ren04] generalised these concepts, and showed how first-order logic (over edge-labelled graphs without parallel edges) has an equivalent representation as a recursively nested set of morphisms. Along the lines of Rensink, nested conditions (allowing parallel edges) were described in [HPR06, HP09] in which they were used to define a weakest preconditions semantics for high-level programs – see Section 4.6. We proposed E-conditions in [PP10a, PP12] as an extension of nested conditions – adding expressions – allowing us to target the verification of

a graph transformation language that manipulated labels as well as structure. A similar approach was used earlier by Orejas [Ore08] for attributed graph constraints, but lacked for example the nesting of E-conditions (but on the other hand, they were able to define a sound and complete inference system for their fragment of graph constraints [OEP10]).

Efforts have been made to lift these specification formalisms to more general objects than graphs. For example, nested conditions have been considered in the framework of weak adhesive HLR categories [HPR06, HP09], making the results applicable for other graph-like structures such as Petri-nets and hypergraphs. Some classical results from graph transformation (e.g. local Church-Rosser, local confluence, embedding) have already been generalised to high-level transformation systems with nested conditions [EGH$^+$13, EGH$^+$12]. Bruggink et al. [BCHK11] lifted nested application conditions to the categorical setting of reactive systems (as defined by Leifer and Milner), and showed that several constructions (e.g. computing weakest preconditions) can be defined more elegantly in this general setting.

Habel and Radke in [HR10] extended nested conditions with hyperedge replacement systems (as in [Hab92]), making the formalism more expressive and able to capture several non-local properties (e.g. arbitrary-length paths, connectedness). With a generalisation of the transformation Pre (efforts are underway [Rad10]), the formalism could be plugged in to our extensional calculi to allow for practical proofs more powerful than those presented in this thesis.

4.8 Summary

In this chapter we have:

- extended the graph conditions specification formalism with expressions (E-constraints) to allow for graphical reasoning about the structure and data of graphs;
- instantiated our extensional Hoare calculi with E-constraints;
- defined transformations Pre and App (for a restricted class of programs) which define the assertions Wlp, SE, and FE;
- proven the correctness of these transformations and hence the soundness of the calculi with E-constraints;
- discussed the open problem of whether or not our calculi with E-constraints are relatively complete;
- compared our approach with the weakest preconditions approach from Oldenburg.

Chapter 5

Verification Examples

In the previous two chapters we have described (1) extensional Hoare calculi – axiomatic semantics – for graph programs with three notions of correctness; and (2) an assertion language for these calculi expressive enough to define the special assertions Pre, SE, and FE (under certain restrictions). Up to this point we have demonstrated the concepts, definitions, and transformations on small and partial examples. This is fine, of course, for helping to explain particular technical details. But what we have not attempted to show just yet is that our ideas can be used together to verify interesting properties of interesting graph programs.

This chapter revisits and introduces some graph programs, and verifies a number of partial, weak total, and total correctness specifications. We focus on using our proof calculi with E-constraints, but later consider a program and specification that require an assertion language more expressive.

5.1 Computing a Graph Colouring

First, we return to the graph colouring[1] program introduced in Chapter 2, and use our proof calculi with E-constraints to prove that given a suitable input graph, any output graph is correctly coloured. Moreover, we prove that the program is totally correct, i.e. will always terminate and never fail.

We give the program `colouring` again in Figure 5.1. Given a graph in which all (unmarked) nodes are labelled with atoms, recall that the program first appends an initial colour of 0 to each node label, before repeatedly matching adjacent nodes with the same colour and incrementing the colour of the target. At termination, the graph is returned with a colouring encoded into the labels.

[1] Recall that a colouring is an assignment of natural numbers to nodes such that no two adjacent nodes have the same colour.

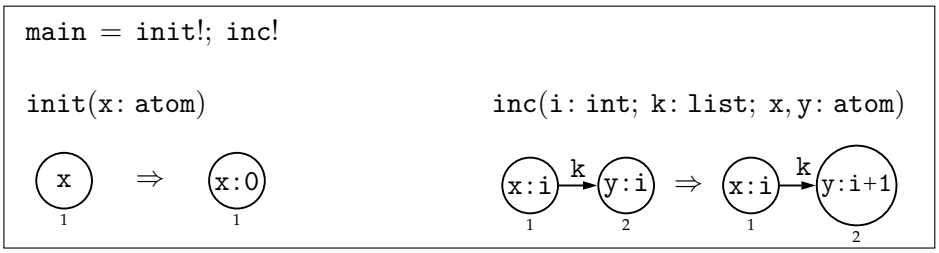


Figure 5.1: The program `colouring`

Example 5.1 (Partial correctness of `colouring`)**.** We begin by formalising a partial correctness specification of `colouring` using E-constraints:

Precondition "every (unmarked) node is labelled by an atom"

$$\forall(\ \textcircled{a}_1, \exists(\ \textcircled{a}_1 \mid \texttt{atom(a)}\ ))$$

Postcondition "every (unmarked) node is labelled by a list $b : c$ with b an atom and c a natural number, and adjacent nodes are distinctly coloured"

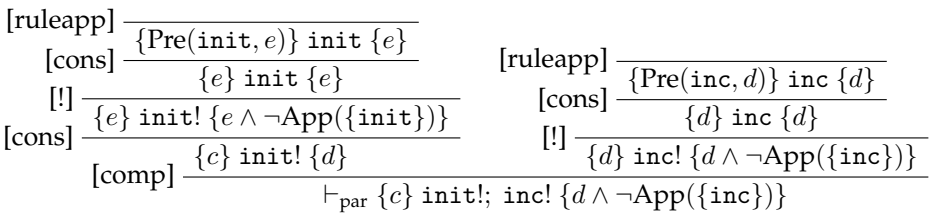
$$\forall(\ \textcircled{a}_1, \exists(\ \textcircled{a}_1 \mid \texttt{a = b:c and atom(b) and c >= 0}))$$
$$\wedge\ \neg\exists(\ \textcircled{x:i} \xrightarrow{k} \textcircled{y:i} \mid \texttt{atom(x, y) and int(i)}\ )$$

Note that the specification we are proving does not guarantee anything about marked nodes, or nodes linked by marked edges. We could write a stronger specification that requiring their absence, but we choose not to in order to keep the example simple.

We give a proof tree for this specification in Figure 5.2, with the precondition, program, and postcondition forming the triple at the root of the tree. For clarity, we split the postcondition into two parts: d and $\neg\mathrm{App}(\{\texttt{inc}\})$. This makes clear the insight that part of the postcondition is a direct result of the non-applicability of `inc` once the iteration terminates. The assertion d itself is an invariant for `inc`, and is inferred from the invariant of `init` and the non-applicability of that rule schema once its iteration terminates.

Note that we give simplified (by hand) E-constraints from Pre in Figure 5.2. The actual results of the transformation are given in full in Figure 5.3.

The three applications of [cons] have side conditions to be shown, i.e. implications that must be shown to be valid. We argue that this is the case for the non-trivial implications:



$$
\begin{aligned}
c &= \forall(\ \textcircled{a}_1, \exists(\ \textcircled{a}_1 \mid \texttt{atom(a)}\ ))\\
d &= \forall(\ \textcircled{a}_1, \exists(\ \textcircled{a}_1 \mid \texttt{a = b:c and atom(b) and c >= 0}))\\
e &= \forall(\ \textcircled{a}_1, \exists(\ \textcircled{a}_1 \mid \texttt{atom(a)}\ )\\
&\qquad \vee \exists(\ \textcircled{a}_1 \mid \texttt{a = b:c and atom(b) and c >= 0}))\\
\neg\mathrm{App}(\{\texttt{init}\}) &= \neg\exists(\ \textcircled{x}\ \mid \texttt{atom(x)}\ )\\
\neg\mathrm{App}(\{\texttt{inc}\}) &= \neg\exists(\ \textcircled{\texttt{x:i}} \xrightarrow{\texttt{k}} \textcircled{\texttt{y:i}}\ \mid \texttt{atom(x,y) and int(i)}\ )\\
\mathrm{Pre}(\texttt{init}, e) &\equiv \forall(\ \textcircled{x}_1\ \textcircled{a}_2 \mid \texttt{atom(x)}, \exists(\ \textcircled{x}_1\ \textcircled{a}_2 \mid \texttt{atom(a)}\ )\\
&\qquad \vee \exists(\ \textcircled{x}_1\ \textcircled{a}_2 \mid \texttt{a = b:c and atom(b) and c >= 0}))\\
\mathrm{Pre}(\texttt{inc}, d) &\equiv \forall(\ \textcircled{\texttt{x:i}}_1 \xrightarrow{\texttt{k}} \textcircled{\texttt{y:i}}_2 \mid \texttt{atom(x,y) and int(i)},\\
&\qquad \forall(\ \textcircled{\texttt{x:i}}_1 \xrightarrow{\texttt{k}} \textcircled{\texttt{y:i}}_2\ \textcircled{\texttt{a}}_3,\\
&\qquad\quad \exists(\ \textcircled{\texttt{x:i}}_1 \xrightarrow{\texttt{k}} \textcircled{\texttt{y:i}}_2\ \textcircled{\texttt{a}}_3 \mid \texttt{a = b:c and atom(b)}\\
&\qquad\qquad \texttt{and c >= 0}))\\
&\qquad \wedge \exists(\ \textcircled{\texttt{x:i}}_1 \xrightarrow{\texttt{k}} \textcircled{\texttt{y:i}}_2 \mid \texttt{i >= 0}))
\end{aligned}
$$

Figure 5.2: A partial correctness proof tree for the program `colouring`

Validity of $e \Rightarrow \mathrm{Pre}(\texttt{init}, e)$. If e is satisfied by a graph, then every (unmarked) node is either labelled with an atom, or a list $b : c$ with b an atom and c a natural number. Such a graph will also satisfy $\mathrm{Pre}(\texttt{init}, e)$, which expresses the same requirement of every node distinct from matches of atom-labelled nodes. Hence the implication is valid.

Validity of $d \Rightarrow \mathrm{Pre}(\texttt{inc}, d)$. If d is satisfied by a graph, then every (unmarked) node is labelled by a list $b{:}c$ with b an atom and c a natural number. Such a graph will also satisfy $\mathrm{Pre}(\texttt{inc}, d)$, which expresses that (1) the property is true of every node outside matches of `inc`; and (2) that the colours i in matches of `inc` are natural numbers (this must be the case since d expresses that all nodes are coloured, and all colours are natural numbers).

$\mathrm{Pre}(\texttt{init}, e)$

$= \quad \forall(\ \textcircled{x}_1 \mid \texttt{atom(x)}\,,$

$\forall(\ \textcircled{x}_1\ \textcircled{a}_2, \exists(\ \textcircled{x}_1\ \textcircled{a}_2 \mid \texttt{atom(a)}\ )$

$\vee\ \exists(\ \textcircled{x}_1\ \textcircled{a}_2 \mid \texttt{a = b:c and atom(b) and c >= 0}))$

$\wedge\ \forall(\ \textcircled{x}_1, \exists(\ \textcircled{x}_1 \mid \texttt{atom(x:0)}\ )$

$\vee\ \exists(\ \textcircled{x}_1 \mid \texttt{x:0 = b:c and atom(b) and c >= 0})))$

$\mathrm{Pre}(\texttt{inc}, d)$

$= \quad \forall(\ \textcircled{x:i}_1 \xrightarrow{k} \textcircled{y:i}_2 \mid \texttt{atom(x, y) and int(i)}\,,$

$\forall(\ \textcircled{x:i}_1 \xrightarrow{k} \textcircled{y:i}_2\ \textcircled{a}_3\,,$

$\exists(\ \textcircled{x:i}_1 \xrightarrow{k} \textcircled{y:i}_2\ \textcircled{a}_3 \mid \texttt{a = b:c and atom(b) and c >= 0}))$

$\wedge\ \forall(\ \textcircled{x:i}_1 \xrightarrow{k} \textcircled{y:i}_2\,,$

$\exists(\ \textcircled{x:i}_1 \xrightarrow{k} \textcircled{y:i}_2 \mid \texttt{x:i = b:c and atom(b) and c >= 0}))$

$\wedge\forall(\ \textcircled{x:i}_1 \xrightarrow{k} \textcircled{y:i}_2\,,$

$\exists(\ \textcircled{x:i}_1 \xrightarrow{k} \textcircled{y:i}_2 \mid \texttt{y:i+1 = b:c and atom(b) and c >= 0})))$

Figure 5.3: Non-simplified E-constraints arising from Pre

Hence the implication is valid.

Validity of $c \Rightarrow e$. A graph satisfying c does not contain any (unmarked) nodes labelled with non-atomic lists. Such a graph will also satisfy e because of its first nested disjunct. Hence the implication is valid.

Validity of $e \wedge \neg\mathrm{App}(\{\texttt{init}\}) \Rightarrow d$. Every (unmarked) node in a graph satisfying e is either labelled by an atom (first nested disjunct) or an atom followed by a natural number (second nested disjunct). If such a graph also satisfies $\neg\mathrm{App}(\{\texttt{init}\})$, which contradicts the first nested disjunct of e, it must then be the case that every (unmarked) node is labelled by an atom followed by a natural number – exactly what the E-constraint d expresses. Hence the implication is valid. □

Thus far we have proven that, given a graph containing only atom-labelled (unmarked) nodes, `colouring` will only ever result in graphs that are correctly coloured. However, we have not proven that `colouring` actually ever *will* return a graph. Certainly, the program will not fail: this we get immediately from the fact that as-long-as-possible iteration does not fail. However, we need some more effort to prove that the program will always terminate eventually.

Example 5.2 (Total correctness of `colouring`)**.** Our total correctness proof for `colouring` is much the same as our partial correctness one in Example 5.1, but augmented with termination functions for `init` and `inc`. Recall that in Example 3.18 we gave already a termination function for the latter. This termination function however requires that graphs satisfy the invariant for `inc` described below (in English):

> "every node label has the form $x:i$ for some atom x, and colour i (which is a natural number). Moreover, there is a sequence $0 \trianglelefteq \cdots \trianglelefteq n$ for $n \geq 0$, including each of the colours used in the graph, where $p \trianglelefteq q$ holds if $q = p$ or $q = p + 1$."

The invariant d in Example 5.1 expresses only the first sentence of the above, and hence is too weak for our total correctness proof. We strengthen it to the following for this example, to require also the existence of such a sequence of colours:

$$
\begin{aligned}
d \quad = \quad & \forall(\ \textcircled{a}_1, \exists(\ \textcircled{a}_1 \mid \texttt{a = b:c and atom(b) and c >= 0})) \\
& \wedge \forall(\ \textcircled{b:c}_1 \mid \texttt{atom(b)}, \exists(\ \textcircled{b:c}_1 \mid \texttt{c = 0}) \\
& \quad \vee \exists(\ \textcircled{b:c}_1\ \textcircled{d:c-1} \mid \texttt{atom(d)}\ ))
\end{aligned}
$$

i.e. unless the colour c of a node is 0, then some other node in the graph has colour $c - 1$.

With a stronger invariant in the part of the proof tree about `inc`, we can prove a total correctness specification for the whole program with a stronger postcondition than before:

Precondition "every (unmarked) node is labelled by an atom"

$$\forall(\ \textcircled{a}_1, \exists(\ \textcircled{a}_1 \mid \texttt{atom(a)}\ ))$$

Postcondition "every (unmarked) node is labelled by a list $b : c$ with b an atom and c a natural number, adjacent nodes are distinctly coloured, and unless the colour c of a node is 0, then there is another node with colour $c - 1$"

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

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

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

Figure 5.4: A total correctness proof tree for the program `colouring`

We give a proof tree for this total correctness specification in Figure 5.4, with the precondition, program, and postcondition forming the triple at the root of the tree. The E-constraints are given in Figure 5.5, and the termination functions are defined in the text that follows.

We define $\#_{\texttt{init}} : \mathcal{G}(\mathcal{L}) \to \mathbb{N}$ to map graphs G to the total number of (unmarked) nodes labelled by an atom in G. The rule schema `init` is clearly $\#_{\texttt{init}}$-decreasing under e since every application of `init` replaces an atomic label of a node with a (non-atomic) list. We define $\#_{\texttt{inc}} : \mathcal{G}(\mathcal{L}) \to \mathbb{N}$ for a graph $G \in \mathcal{G}(\mathcal{L})$ as:

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

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

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

The intuition behind this termination function, and a proof that `inc` is $\#_{\texttt{inc}}$-decreasing under d is given in Example 3.18.

The implications arising from applications of [cons] follow the same pattern as in Example 5.1, though we mention one case:

Validity of $c \Rightarrow e$. In the total correctness proof tree, e now has two conjuncts. The first is implied by c by the same argument of the partial correctness case. The second conjunct, expressing a property about every list $b : c$ with b an atom, is vacuously true if c holds because c asserts that every (unmarked) node label is an atom.

□

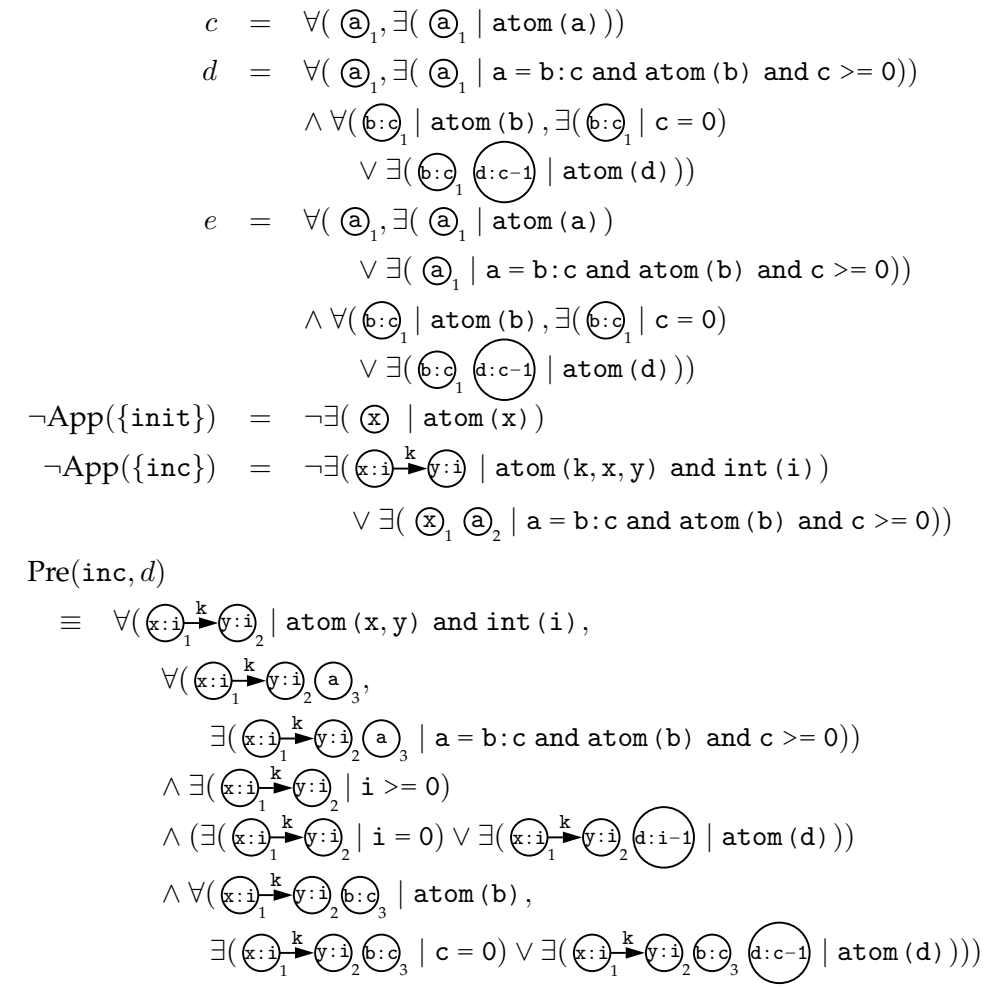


Figure 5.5: Selected E-constraints for the proof tree of Figure 5.4

5.2 Finding a 2-Colouring

For the next proofs, we continue with the theme of colouring, but consider a slightly more difficult program that introduces a (potential) failure point and a conditional. Again, we apply our verification calculi to prove some partial, weak total, and total correctness specifications.

In this section we will consider the program `2colouring`, given in Figure 5.6. The program attempts to compute a 2-colouring for a graph of unmarked nodes and edges, i.e. an assignment of colour 0 or 1 to every node such that no two adjacent nodes have the same colour. If such a colouring is computed, the graph (with colours added to lists) is returned to the user. If such a colouring is not computed, then all of the colours are removed

before the graph is returned. We remark that this particular version of the program requires input graphs to be connected.

In an execution of `2colouring`, an unmarked node is nondeterministically chosen and coloured with 0 (this is a potential failure point if no such node is in the input graph). Then, the rule schemata `colour1` and `colour2` are iteratively applied for as long as possible. These match a coloured and uncoloured node adjacent to each other, take the colour of the former, change 0 to 1 or 1 to 0, and apply that new colour to the latter node. After the iteration, the program attempts to match two adjacent nodes with the same colour. If it finds two such nodes, it iteratively removes all of the colours and returns the graph. Otherwise, it returns the graph with the 2-colouring computed displayed in the labels.

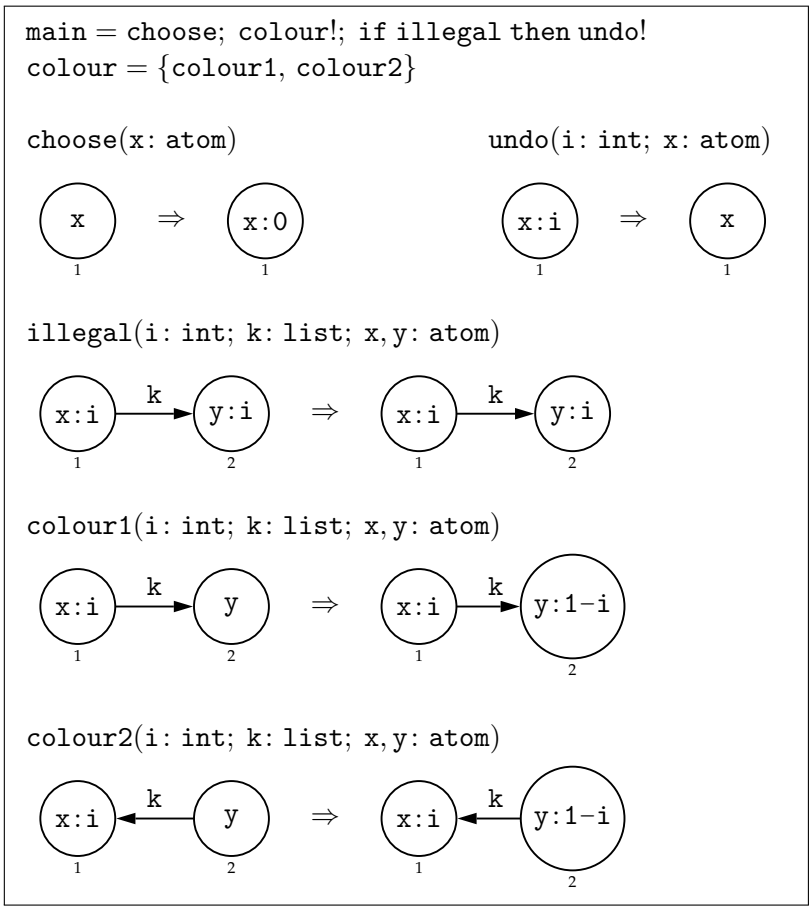
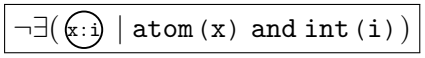


Figure 5.6: The program `2colouring`

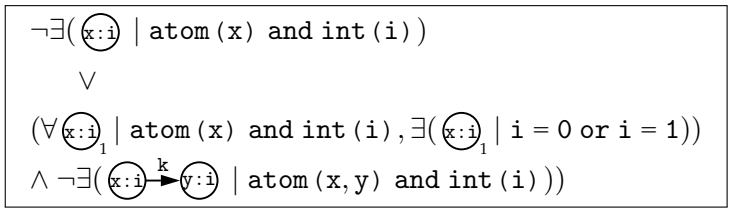
Example 5.3 (Partial correctness of `2colouring`)**.** We begin by formalising a partial correctness specification of `2colouring` using E-constraints. (Note

that for simplicity we reason only about the unmarked nodes and edges of the graphs.)

Precondition "no atom-labelled node is already coloured"



Postcondition "either the precondition holds, or every atom-labelled node with a colour has colour 0 or 1, and no two adjacent nodes have the same colour"



We give a proof tree for this partial correctness specification in Figure 5.7, and separately, all of the E-constraints used as assertions in Figures 5.8 and 5.9, the latter providing the E-constraints generated by Pre. The postcondition is split into two E-constraints and given as $c \vee d$, to make explicit that part of the postcondition – c – is the same as the precondition.

The side conditions of [cons] and [if$_2$] require us to show the validity of several E-constraints of the form $c \Rightarrow d$. We argue for the validity of all but the most trivial cases:

Validity of $c \Rightarrow$ Pre(`choose`, f). The first conjunct of the nested part of Pre(`choose`, f) is clearly satisfied by any graph. The second conjunct demands that there is not a distinct (unmarked) node from node 1 that is atom-labelled and tagged with a colour. E-constraint c expresses that no atom-labelled node is coloured, hence the whole implication is valid.

Validity of $e \Rightarrow$ Pre(`colour1`, e). For Pre(`colour1`, e) to be satisfied by a graph, for every possible match of `colour1`, node 1 must be coloured 0 or 1, and every coloured atom-labelled node outside of the match must be coloured 0 or 1. Additionally, the colour that node 2 will be assigned after the application of `colour1` must also be 0 or 1 (which it will be if `i` is assigned to 0 or 1, by `1-i` in the assignment constraint). The E-condition e is satisfied if and only if every coloured atom-labelled node has colour 0 or 1, so the whole implication must be valid.

Validity of $e \Rightarrow$ Pre(`colour2`, e). Analogous to the above.

$$[\text{comp})\ \frac{\text{Subtree } A \qquad \text{Subtree } B}{\vdash_{\text{par}} \{c\}\ \texttt{choose};\ \{\texttt{colour1}, \texttt{colour2}\}!;\ \texttt{if illegal then undo}!\ \{c \vee d\}}$$

where Subtree A is:

$$[\text{comp}]\ \frac{[\text{ruleapp}]\ \dfrac{}{\{\text{Pre}(\texttt{choose}, f)\}\ \texttt{choose}\ \{f\}}}{[\text{cons}]\ \dfrac{}{\{c\}\ \texttt{choose}\ \{f\}}} \qquad \frac{[\text{ruleset}]\ \dfrac{[\text{ruleapp}]\ \dfrac{}{\{\text{Pre}(\texttt{colour1}, e)\}\ \texttt{colour1}\ \{e\}}}{[\text{cons}]\ \dfrac{}{\{e\}\ \texttt{colour1}\ \{e\}}} \qquad [\text{ruleapp}]\ \dfrac{}{[\text{cons}]\ \dfrac{\{\text{Pre}(\texttt{colour2}, e)\}\ \texttt{colour2}\ \{e\}}{\{e\}\ \texttt{colour2}\ \{e\}}}}{\{e\}\ \{\texttt{colour1}, \texttt{colour2}\}\ \{e\}}}{[!]\ \dfrac{}{[\text{cons}]\ \dfrac{\{e\}\ \{\texttt{colour1}, \texttt{colour2}\}!\ \{e \wedge \neg\text{App}(\{\texttt{colour1}, \texttt{colour2}\})\}}{\{f\}\ \{\texttt{colour1}, \texttt{colour2}\}!\ \{e\}}}}$$

$$\{c\}\ \texttt{choose};\ \{\texttt{colour1}, \texttt{colour2}\}!\ \{e\}$$

and Subtree B is:

$$[\text{ruleapp}]\ \frac{}{[!]\ \dfrac{\{\texttt{true}\}\ \texttt{undo}\ \{\texttt{true}\}}{[\text{cons}]\ \dfrac{\{\texttt{true}\}\ \texttt{undo}!\ \{\neg\text{App}(\{\texttt{undo}\})\}}{[\text{if}_2]\ \dfrac{\{e \wedge \text{App}(\{\texttt{illegal}\})\}\ \texttt{undo}!\ \{c \vee d\}}{\{e\}\ \texttt{if illegal then undo}!\ \{c \vee d\}}}}$$

Figure 5.7: A proof tree for the program `2colouring`

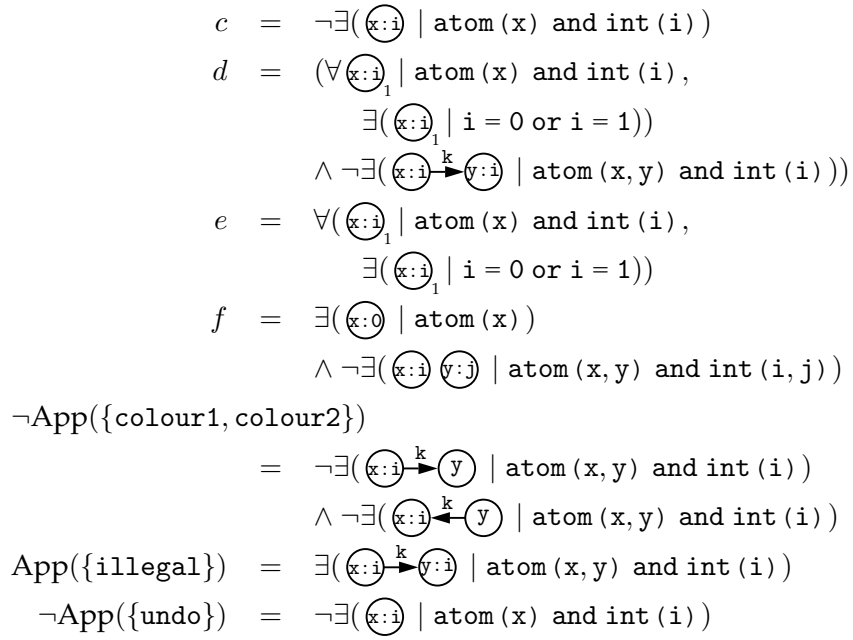


Figure 5.8: E-constraints c, d, e, f and App(_) from Figure 5.7

Validity of $f \Rightarrow e$. For e to be satisfied, every coloured atom-labelled node in the graph must be coloured with 0 or 1. If f is satisfied, then one such node is coloured 0, but there are not two coloured atom-labelled nodes, i.e. only one atom-labelled node is coloured and it has colour 0. Hence, every such node has colour 0 or 1, and the implication is valid.

Validity of $e \wedge \neg\text{App}(\{\texttt{illegal}\}) \Rightarrow c \vee d$. This E-constraint arising from [if$_2$] is valid because $e \wedge \neg\text{App}(\{\texttt{illegal}\})$ is the same E-constraint as d. □

Example 5.4 (Weak total correctness of `2colouring`)**.** The specification we considered in Example 5.3 can be shown to be weak totally correct as well as partially correct. Revising the proof tree for this is straightforward, and is achieved first by replacing [if$_2$] with [if$_2$]$_{\text{wtot}}$, which, in the verification calculi for E-constraints (though not in the extensional calculi) is the same as its partial correctness counterpart other than for the implicit $\vdash_{\text{wtot}}$ rather than $\vdash_{\text{par}}$. Then, replace the two instances of [!] with [!]$_{\text{wtot}}$, using the simple termination functions described in what follows.

For $\{\texttt{colour1}, \texttt{colour2}\}$ we can define the termination function $\#_{\texttt{colour}} : \mathcal{G}(\mathcal{L}) \to \mathbb{N}$ which for a graph $G \in \mathcal{G}(\mathcal{L})$ returns the total number of unmarked nodes with a list component that is an atom (i.e. an integer or string). Clearly, $\{\texttt{colour1}, \texttt{colour2}\}$ is decreasing for this termination function since a successful application of either rule schema reduces the number

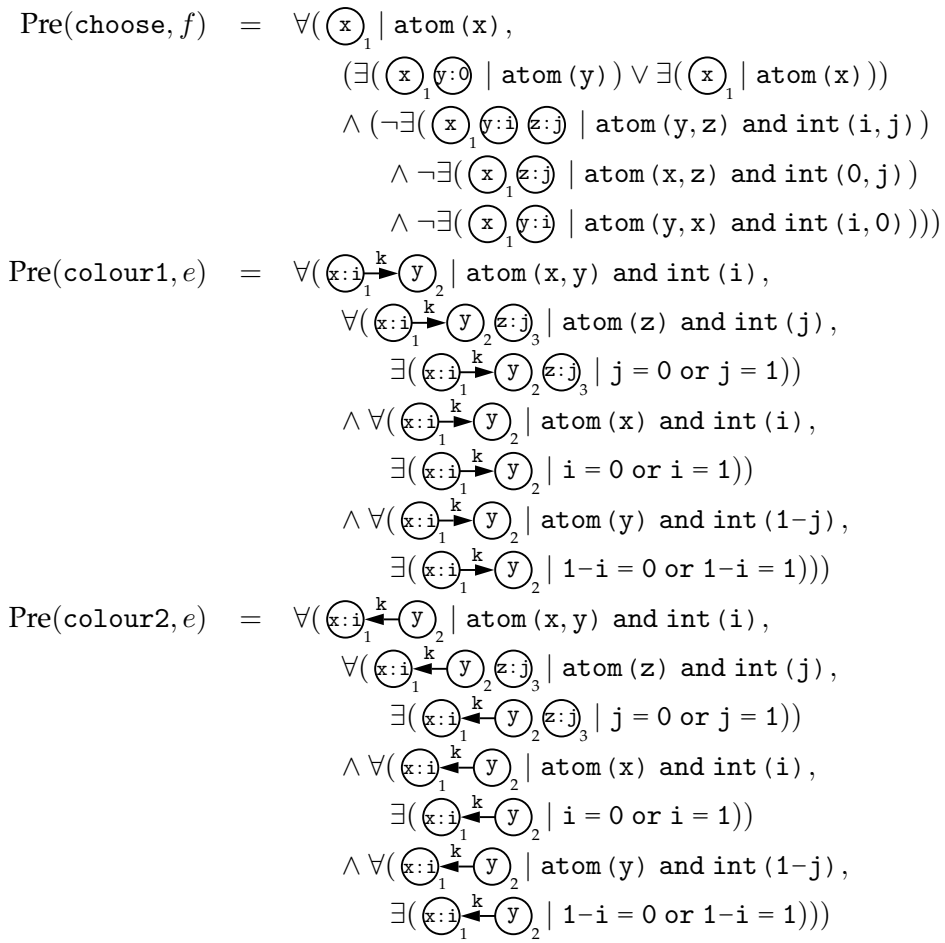


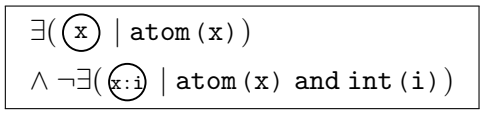
Figure 5.9: E-constraints generated by Pre from Figure 5.7

of such list components by one, replacing an atom y with a list $y:i$ of length two.

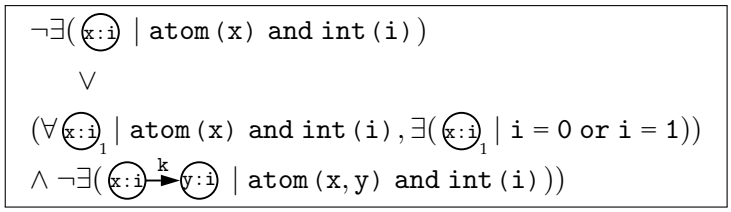
For undo define the termination function $\#_{\mathtt{undo}} : \mathcal{G}(\mathcal{L}) \to \mathbb{N}$ which for a graph $G \in \mathcal{G}(\mathcal{L})$ returns the total number of unmarked nodes with a list component of the form $x:i$, with x an atom and i an integer. Clearly, undo is decreasing for this termination function since each application of the rule schema removes the colour i of such a node, leaving it with only the atom x. □

Example 5.5 (Total correctness of 2colouring)**.** The specifications of the previous two examples are not totally correct, because the empty graph would satisfy the precondition, yet executing choose on the empty graph would cause the program to fail. Hence we strengthen the precondition to additionally assert that choose is applicable.

Precondition "there exists at least one atom-labelled node, and no atom-labelled node is already coloured"



Postcondition "either the precondition holds, or every atom-labelled node with a colour has colour 0 or 1, and no two adjacent nodes have the same colour"



We give a proof tree for this total correctness specification in Figure 5.10, where

$$b = \mathrm{App}(\{\texttt{choose}\}) = \exists(\,\textcircled{\texttt{x}}\mid \texttt{atom(x)}\,)$$

and all other E-constraints are as in Figures 5.8 and 5.9. To avoid clutter, we do not give the termination functions for $[!]_{\mathrm{tot}}$ in the proof tree, but remark that they are the same as those in Example 5.4.

The total correctness proof adds one further implication to be proven valid from the application of $[\text{ruleset}]_{\mathrm{tot}}$, but it is easily discharged.

Validity of $b \wedge c \Rightarrow \mathrm{App}(\{\texttt{choose}\})$. We get validity immediately from the fact that $b = \mathrm{App}(\{\texttt{choose}\})$. □

5.3 Existence of a Path

Now, we consider the program `reachable?` of Figure 5.11, which checks whether a path exists (along unmarked nodes and edges) between two nodes distinguished in the input graph (distinguished by the user, or perhaps by some other program). Like `2colouring`, the program `reachable?` can fail on some input graphs, in particular, input graphs omitting the pair of distinguished nodes (although the failure point is not right at the beginning of the command sequence this time).

The program is intended to operate on graphs in which nodes and edges are unmarked, and in which nodes are labelled with atoms, with the exception of two distinguished nodes. Exactly one node should have a list

$$[\text{comp})\ \frac{\text{Subtree } A \qquad \text{Subtree } B}{\vdash_{\text{tot}} \{b \wedge c\}\ \texttt{choose};\ \{\texttt{colour1}, \texttt{colour2}\}!;\ \texttt{if illegal then undo}!\ \{c \vee d\}}$$

where Subtree A is:

$$\dfrac{\dfrac{\dfrac{}{\vdash_{\text{par}} \{\text{Pre}(\texttt{choose}, f)\}\ \texttt{choose}\ \{f\}}[\text{ruleapp}]}{\vdash_{\text{par}} \{b \wedge c\}\ \texttt{choose}\ \{f\}}[\text{cons}]}{\vdash_{\text{tot}} \{b \wedge c\}\ \texttt{choose}\ \{f\}}[\text{ruleset}]_{\text{tot}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\vdash_{\text{par}} \{\text{Pre}(\texttt{colour1}, e)\}\ \texttt{colour1}\ \{e\}}[\text{ruleapp}]}{\vdash_{\text{par}} \{e\}\ \texttt{colour1}\ \{e\}}[\text{cons}] \qquad \dfrac{}{\vdash_{\text{par}} \{\text{Pre}(\texttt{colour2}, e)\}\ \texttt{colour2}\ \{e\}}[\text{ruleapp}]}{\vdash_{\text{par}} \{e\}\ \texttt{colour2}\ \{e\}}[\text{cons}]}{\vdash_{\text{par}} \{e\}\ \{\texttt{colour1}, \texttt{colour2}\}\ \{e\}}[\text{ruleset}]}{\vdash_{\text{tot}} \{e\}\ \{\texttt{colour1}, \texttt{colour2}\}!\ \{e \wedge \neg\text{App}(\{\texttt{colour1}, \texttt{colour2}\})\}}[!]_{\text{tot}}}{\vdash_{\text{tot}} \{f\}\ \{\texttt{colour1}, \texttt{colour2}\}!\ \{e\}}[\text{cons}]$$
$$\vdash_{\text{tot}} \{b \wedge c\}\ \texttt{choose};\ \{\texttt{colour1}, \texttt{colour2}\}!\ \{e\}\ [\text{comp}]$$

and Subtree B is:

$$\dfrac{\dfrac{\dfrac{}{\vdash_{\text{par}} \{\texttt{true}\}\ \texttt{undo}\ \{\texttt{true}\}}[\text{ruleapp}]}{\vdash_{\text{tot}} \{\texttt{true}\}\ \texttt{undo}!\ \{\neg\text{App}(\{\texttt{undo}\})\}}[!]_{\text{tot}}}{\vdash_{\text{tot}} \{e \wedge \text{App}(\{\texttt{illegal}\})\}\ \texttt{undo}!\ \{c \vee d\}}[\text{cons}]}{\vdash_{\text{tot}} \{e\}\ \texttt{if illegal then undo}!\ \{c \vee d\}}[\text{if}_2]$$

Figure 5.10: A total correctness proof tree for the program `2colouring`

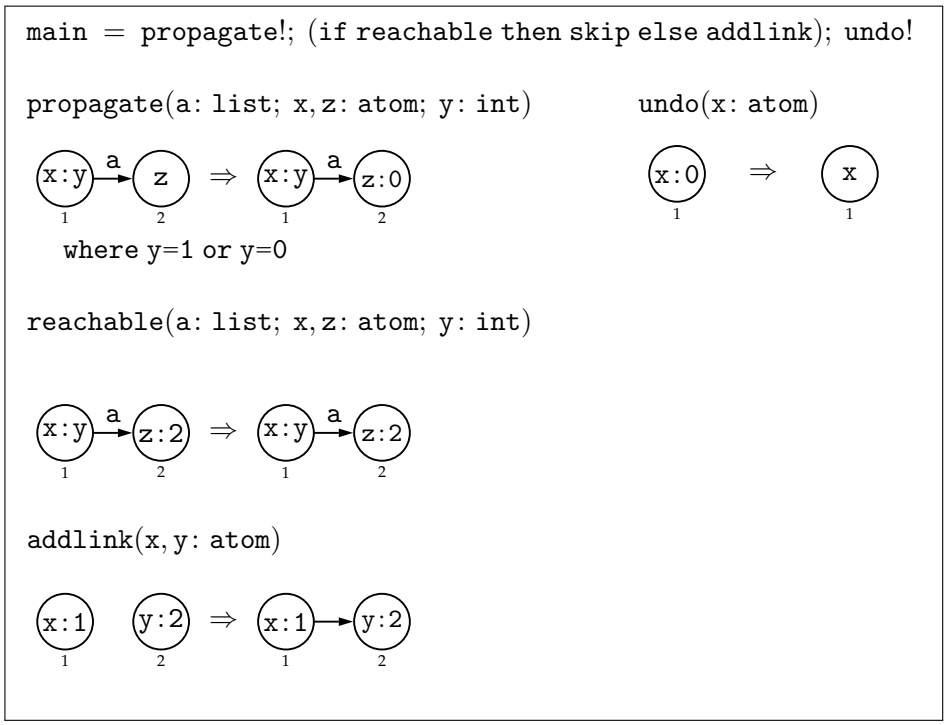


Figure 5.11: The program `reachable`?

component of the form $x:1$, and exactly one node should have a list component of the form $y:2$ (where x, y are atoms). The program `reachable`? then checks whether or not there is a path from the former node to the latter node, and if not, adds an edge from the former to the latter. It does so first by iterating `propagate`, which "tags" nodes adjacent to the first distinguished node with 0, and then iteratively tags all nodes adjacent to those. Then, it checks whether or not there is an edge directly from such a node to the second distinguished node. If there is, then there must be a path between the distinguished nodes already. If there is not, then `addlink` adds an edge directly between them. Finally, all of the 0s added to list components are removed, returning the input graph if there was a path between the distinguished nodes, otherwise the input graph augmented with a new edge between them.

The most interesting partial correctness specifications for `reachable`? – e.g. a postcondition expressing that there is a path between the two distinguished nodes – cannot be expressed using E-constraints, because such properties are non-local and hence beyond the expressive power of the formalism. We can however prove the total correctness of the program with respect to a precondition (that is also a program invariant) expressing that

all nodes are atom-labelled apart from the two distinguished nodes.

Example 5.6 (Total correctness of `reachable`?)**.** We start by formalising a total correctness specification for `reachable`?. (Again, marked nodes and edges are not considered in the specification as the program does not operate on them.)

Precondition "all nodes are atom-labelled, except for two distinguished nodes which have list components of the form $x : 1$, $y : 2$ with x, y atoms"

$$\boxed{\exists(\textcircled{\texttt{x:1}}_1\,\textcircled{\texttt{y:2}}_2 \mid \texttt{atom(x,y)}, \neg\exists(\textcircled{\texttt{x:1}}_1\,\textcircled{\texttt{y:2}}_2\,\textcircled{\texttt{p}} \mid \texttt{not atom(p)}))}$$

Postcondition "the precondition holds" (i.e. it is a program invariant)

$$\boxed{\exists(\textcircled{\texttt{x:1}}_1\,\textcircled{\texttt{y:2}}_2 \mid \texttt{atom(x,y)}, \neg\exists(\textcircled{\texttt{x:1}}_1\,\textcircled{\texttt{y:2}}_2\,\textcircled{\texttt{p}} \mid \texttt{not atom(p)}))}$$

We give a proof tree for this specification in Figure 5.12, with a (partial) list of E-constraints in Figure 5.13. For clarity, we let `visited(p)` abbreviate:

$$\texttt{p} = \texttt{a:0 and atom(a)}$$

where `a` is a fresh variable.

From instances of [cons] we get some implications to discharge as valid. (For simplicity we omit most of the E-constraints generated by Pre in this example, but the implications involving them are easy to discharge.)

Validity of $c \Rightarrow e$. The E-constraint c expresses the existence of the distinguished nodes, and that all other nodes are atom-labelled. The E-constraint e expresses the same, except that all other nodes are either atom-labelled or "visited" (i.e. tagged with 0). Hence, e must follow from c.

Validity of $e \wedge \neg\mathrm{App}(\{\texttt{undo}\}) \Rightarrow c$. If a graph satisfies e, then there are the two distinguished nodes, with all other nodes either atom-labelled or "visited" (i.e. nodes that have list component $x : 0$ with x an atom). However, if that graph also satisfies $\neg\mathrm{App}(\{\texttt{undo}\})$, then there are no such visited nodes in the graph. Hence that graph must also satisfy c, which expresses the existence of the distinguished node but also that all other nodes are atom-labelled.

The termination functions required for the instances of $[!]_{\mathrm{tot}}$ are simple. For `propagate` (resp. `undo`), we define the termination function $\#_p : \mathcal{G}(\mathcal{L}) \to \mathbb{N}$ (resp. $\#_u$) to return the number of (unmarked) nodes in a graph that are labelled by an atom (resp. number of atom-labelled nodes tagged with a 0).

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

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{\text{Pre}(\texttt{propagate}, e)\}\ \texttt{propagate}\ \{e\}}[\text{ruleapp}]}{\vdash_{\text{par}} \{e\}\ \texttt{propagate}\ \{e\}}[\text{cons}]}{\{e\}\ \texttt{propagate!}\ \{e \wedge \neg\text{App}(\{\texttt{propagate}\})\}}[!]_{\text{tot}}[\text{cons}] \quad \text{Subtree } X}{\vdash_{\text{tot}} \{c\}\ \texttt{propagate!}; P;\ \texttt{undo!}\ \{c\}}[\text{comp}]$$

Subtree X:

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

Subtree Y:

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

Figure 5.12: Total correctness proof tree for the program `reachable`?

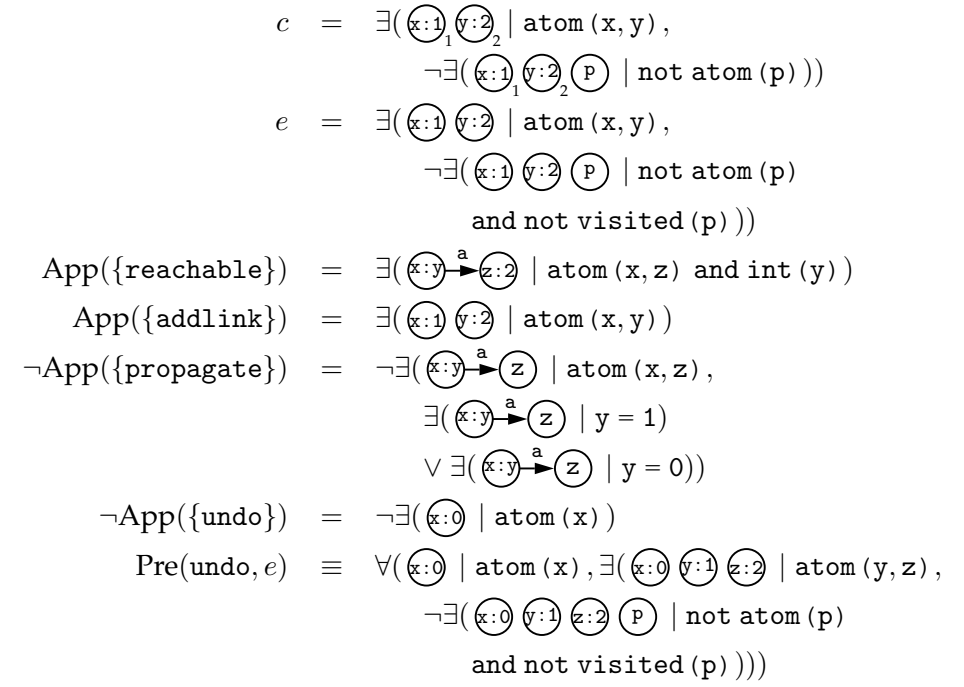


Figure 5.13: Partial list of E-conditions for Figure 5.12

Both termination functions exploit that each application of their respective rule schema explicitly reduces the number of remaining matches.

The application of rule schema `addlink` is the only potential failure point of the program, and is addressed in the proof tree with the instance of $[\text{ruleset}]_{\text{tot}}$. It must be shown that the precondition at that point implies the applicability of `addlink`, i.e. that $e \wedge \neg\text{App}(\{\texttt{reachable}\})$ implies $\text{App}(\{\texttt{addlink}\})$:

Validity of $e \wedge \neg\text{App}(\{\texttt{reachable}\}) \Rightarrow \text{App}(\{\texttt{addlink}\})$. If e is satisfied then clearly there is a pair of (unmarked) nodes with list components $x:1$, $y:2$ with x, y atoms. Hence $\text{App}(\{\texttt{addlink}\})$, which expresses the existence of such two nodes, must also be satisfied. □

5.4 Connectedness: Speculative Proof

In the previous section we verified a total correctness specification of the program `reachable`?, but noted that the most useful specifications would require a stronger assertion language – one able to express non-local properties. In this section, we consider another program for which non-local specifications are the most interesting. This time however, we provide a partial proof tree, in which the assertions are E-constraints that can also

express the existence of arbitrary length paths between nodes. We do not formally define this assertion language, and so indeed only sketch a proof tree. We focus in this section on showing how – if such a language is defined and if Pre is extended – it can be "plugged in" to our extensional calculi to allow a proof in the same spirit as those with just E-constraints.

The program we consider is `connected?`, first given in Figure 2.20, but given again here in Figure 5.14. The program takes a graph (that does not contain unmarked nodes or edges), and checks whether it is connected or not, i.e. whether there is a path between every pair of nodes in the graph (ignoring the directions of edges). If a graph is connected, the program creates a new node labelled "yes". If it is not, it indicates as such with a new node labelled "no". (See Example 2.44 for a description of how the program works.)

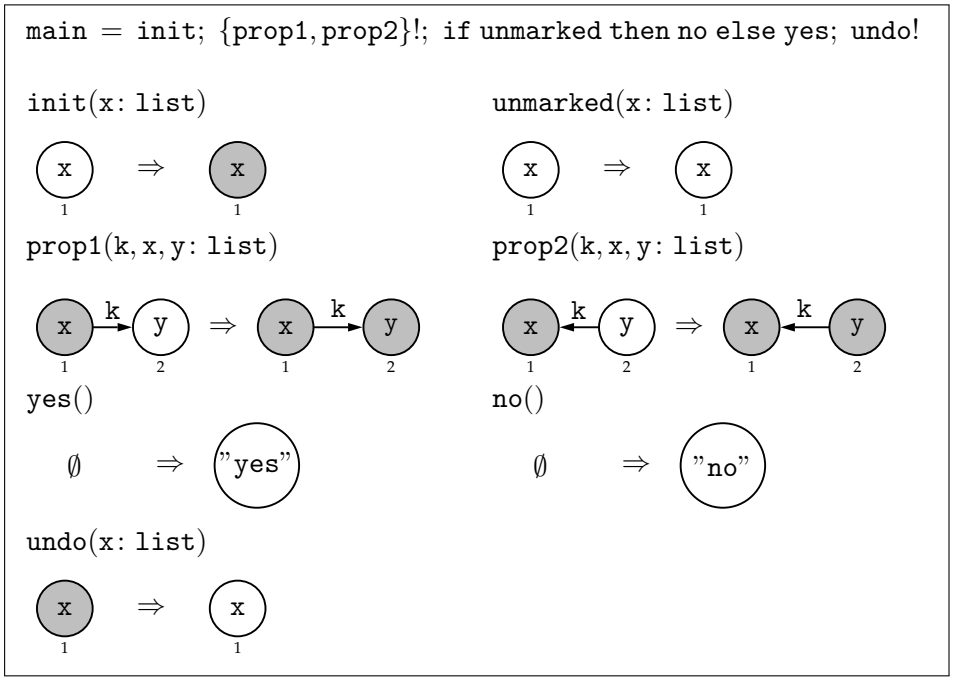


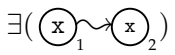
Figure 5.14: The program `connected?`

Example 5.7 (Partial correctness proof sketch for `connected?`)**.** E-constraints are not expressive enough to be able to specify the most interesting properties of `connected?`, as they require reasoning about global properties of the graph. For example, it would be desirable to prove that if a graph is connected, the program will terminate and contain a "yes" node; in the other case a "no" node. We could express that there is a path between every pair of nodes of up to length k (where k is a fixed natural number), however, we cannot express that there is a path of *arbitrary* length between every

pair of nodes, and hence we cannot express that a graph of arbitrary size is connected.

Though we have focused on verification with E-constraints in this thesis, given our extensional proof rules we do not have to be limited to them: we can simply "plug in" a more powerful assertion language. Using the hyperedge-replacement conditions of [HR10], for example, we would be able to write the specifications for `connected`? that we want, but then hit the problem of extending the transformation Pre (which for this formalism in particular appears to be difficult). For this example we are a little more conservative and simply extend E-constraints to allow the expression of arbitrary-length paths between nodes, which is exactly what we need. Again, we have the problem of extending Pre. We do not address this here, leaving the proof as only a sketch to convey the idea, and leave the study of such a formalism as future work (note the discussion in Section 7.2.3).

For this example we (informally) extend E-conditions with a wavy arrow $\rightsquigarrow$, which if present between two nodes, expresses the existence of an arbitrary-length path (across both marked and unmarked items) from the first node to the second. For example, the assertion:



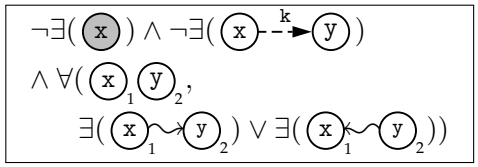
can be interpreted as: "there exists a pair of unmarked nodes with the same list component, such that there is a path from node 1 to node 2". (We continue to require injective matching, so $1 \neq 2$ in the image of the morphism.)

This extension is enough to allow us to write an interesting specification about `connected`?. For simplicity, we consider only the first part of the program:

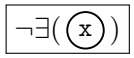
$$\texttt{init};\ \{\texttt{prop1}, \texttt{prop2}\}!$$

but choose a postcondition which clearly informs the if-then-else conditional to execute `yes`.

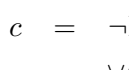
Precondition "(1) all nodes and non-looping edges are unmarked; and (2) every pair of nodes has a path between them in at least one direction"



Postcondition "the rule schema `unmarked` is not applicable"



Note that the precondition is slightly stronger than what we need, since `connected?` disregards the direction of edges. But this specification is simpler and remains true. We provide the assertions in Figure 5.15, and a proof tree separately in Figure 5.16. The soundness of the proof rules is inherited in part by the soundness of the extensional proof rules, as well as the result that App is used to define SE and FE (since these can be expressed with basic E-constraints, clearly they can also be expressed with our speculative extended assertion language).

$c = \neg\exists($ $) \wedge \neg\exists($  $)$

$p_1 = \forall($  ,
 $)$

$p_2 = \forall($  ,
 $)$
$\wedge \forall($,
 $)$

$\text{App}(\{\texttt{unmarked}\}) = \neg\exists($  $)$

$\neg\text{App}(\{\texttt{prop1}, \texttt{prop2}\})$
$= \neg\exists($ $) \wedge \neg\exists($ $)$

Figure 5.15: Partial list of assertions for Figure 5.16

The key idea of the proof is that from a connected graph of unmarked nodes, as the nodes are gradually marked, an invariant is maintained that the graph remains connected *regardless* of whether nodes are marked or not. Then once the iteration finishes, there cannot remain any unmarked nodes in the graph, because the graph is connected but `prop1` and `prop2` (which match a marked node linked to an unmarked node) cannot be applied.

What remains missing from this proof tree are assertions in place of X, Y, Z in the instances of [ruleapp]. One possibility is to manually provide some assertions and prove, one-by-one, that they define a (weakest) liberal precondition with respect to the rule schemata and postconditions. For X, for example, a suitable assertion might be $\forall($  $, c')$, where c' places $p_1 \wedge p_2$ in the context of node 1 and considers all cases in which they might "overlap".

It would clearly be more ideal however to extend the transformation Pre to handle E-constraints augmented with paths. We leave this as future work (see Section 7.2.3), but can envisage how such a construction might work, e.g. extending the disjunction in transformation A to account for the possibility of comatches being partially or totally within paths described by assertions. This could cause assertions generated by Pre to become very

$$\dfrac{\dfrac{\dfrac{}{\{X\}\ \texttt{init}\ \{p_1 \wedge p_2\}}\,[\text{ruleapp}]}{\{c \wedge p_1\}\ \texttt{init}\ \{p_1 \wedge p_2\}}\,[\text{cons}] \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\{Y\}\ \texttt{prop1}\ \{p_1 \wedge p_2\}}\,[\text{ruleapp}]}{\{p_1 \wedge p_2\}\ \texttt{prop1}\ \{p_1 \wedge p_2\}}\,[\text{cons}] \qquad \dfrac{\dfrac{}{\{Z\}\ \texttt{prop2}\ \{p_1 \wedge p_2\}}\,[\text{ruleapp}]}{\{p_1 \wedge p_2\}\ \texttt{prop2}\ \{p_1 \wedge p_2\}}\,[\text{cons}]}{\{p_1 \wedge p_2\}\ \{\texttt{prop1}, \texttt{prop2}\}\ \{p_1 \wedge p_2\}}\,[\text{ruleset}]}{\{p_1 \wedge p_2\}\ \{\texttt{prop1}, \texttt{prop2}\}!\ \{p_1 \wedge p_2 \wedge \neg\text{App}(\{\texttt{prop1}, \texttt{prop2}\})\}}\,[!] \qquad \dfrac{}{\{p_1 \wedge p_2\}\ \{\texttt{prop1}, \texttt{prop2}\}!\ \{\neg\text{App}(\{\texttt{unmarked}\})\}}\,[\text{cons}]}{\vdash_{\text{par}} \{c \wedge p_1\}\ \texttt{init};\ \{\texttt{prop1}, \texttt{prop2}\}!\ \{\neg\text{App}(\{\texttt{unmarked}\})\}}\,[\text{comp}]$$

Figure 5.16: Partial correctness proof tree for the program `connected`?

large, and would make the need for mechanical support in discharging implications a more urgent priority. □

5.5 Summary

In this chapter we have:

- considered a selection of data-manipulating graph programs;
- demonstrated the use of our calculi with E-constraints by variously proving partial, weak total, and total correctness;
- shown how our extensional calculi could be used in proving correctness relative to more powerful assertion languages; and
- motivated the need to extend E-constraints with paths, which will allow for proofs of interesting and useful non-local specifications in the same style as our proofs with basic E-constraints.

Chapter 6

A Many-Sorted Logic for Graph Programs

In Chapters 4 and 5 we discussed the verification of graph programs using our graphical E-conditions as the assertions. Some issues arise however when considering practical aspects of E-conditions, motivating the study of an assertion language more traditional and textual. In this chapter, we define and study a many-sorted first-order predicate logic as an alternative assertion language for verifying graph programs. We show its expressive equivalence to E-conditions by defining and proving correct translations between the two formalisms. With these translations, we show how E-conditions can express e.g. the identification of node (edge) variables, and formulae of arbitrary first-order arithmetic.

6.1 Motivation and Introduction

E-constraints differ from the assertion languages in conventional Hoare calculi in the sense that they are based on graph morphisms, and are not formulae in some traditional logic. While the use of graph morphisms brings a number of advantages – for example, being able to define Pre and App at the level of abstraction of rule schemata – they bring two key problems:

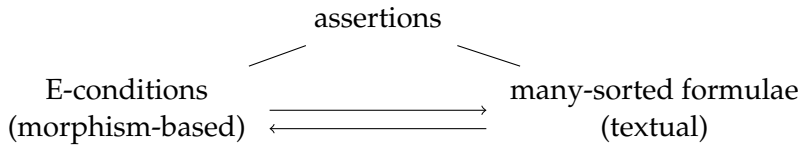
(1) Programmers may find it difficult to specify pre- and postconditions of their graph programs as E-constraints. On the one hand they are graphical, yet on the other it might be difficult to interpret their precise meaning without a basic understanding of concepts such as graph morphisms[1].

[1]The terminology alone in graph transformation – often implying its connection to category theory – might be off-putting. See "The Pushout Scare" in [Ren10].

(2) Implementing a verification system for graph programs in an interactive proof assistant such as Isabelle [NPW02] or Coq [BC04] requires a front end for editing E-conditions and translating them into a textual logic that can be processed by the proof assistant.

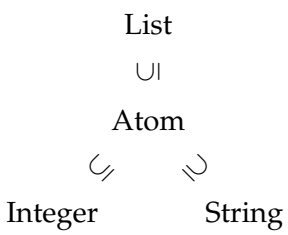
In this chapter, we address these issues by designing a many-sorted predicate logic for graphs. The formulae are textual, with a syntax and semantics familiar to anyone with some knowledge of classical first-order logic. Moreover, they are suitable to be embedded as a theory in a system like Isabelle or Coq, thus laying some groundwork for the formalisation of our verification calculi and practical assistant-guided proofs.

The main technical contributions of this chapter are the constructions of two translations: from E-conditions to many-sorted formulae, and vice versa, such that an E-condition (resp. many-sorted formula) is satisfied by a graph if and only if its translation is also satisfied by that graph. Thus, E-conditions and many-sorted formulae are equivalent in power and can both be used as assertions for graph programs:



In particular, many-sorted formulae can be used as assertions in the proof rules of Chapter 3, resulting in textual Hoare calculi that are sound with respect to the operational semantics of GP. The assertions Wlp, SE, and FE are defined by translations to E-conditions and back via the transformations Pre and App of Chapter 4.

Whereas the expressions in E-conditions are untyped, the expressions in our logic are many-sorted to distinguish clearly and easily between nodes, edges and lists. Moreover, expressions for lists are hierarchically structured into subsorts which reflect the list subtypes of GP:



The subsort hierarchy allows us to compare arbitrary labels with subsort expressions, as in the formula

$$\forall \mathtt{v{:}V.\ l(v)} = \text{"}\mathtt{abc}\text{"} \vee \exists \mathtt{x{:}I.\ l(v)} = \mathtt{x} * \mathtt{x}$$

which states that every node (vertex) `v` is labelled with the string "`abc`" or with the square of some integer `x`.

In general, many-sorted logics enjoy some well known advantages over unsorted logics, including improved readability of specifications, early error detection, and more efficient deduction procedures due to a reduced search space (see for example [Man96, BHP+92, GM92]).

The rest of this chapter is organised as follows. In Section 6.2 we define the syntax and semantics of our many-sorted logic. In Section 6.3 we define and prove correct a translation from many-sorted formulae to E-conditions. Following this we show in Section 6.4 that there is a translation from E-conditions to many-sorted formulae, establishing the expressive equivalence of the formalisms. In Section 6.6 we discuss related work, before summarising in Section 6.7.

Remark 6.1 (Contribution of syntax)**.** We remark that the abstract syntax of the many-sorted logic – described in the following section – was contributed by Detlef Plump. □

6.2 Syntax, Semantics, and Sort Hierarchy

We begin in this section by defining the syntax of our many-sorted logic, before defining how formulae are to be interpreted over graphs.

6.2.1 Abstract Syntax

First we define the expressions (or terms) of our logic. We do so by a grammar, which – similarly to those defining the syntax of graph programs – aims to guarantee that all expressions are well-typed. For example, the grammar enforces that the arguments to arithmetic operators are integer expressions, whose variables (if any) will only ever be interpreted as integers. This contrasts with E-conditions, for which well-typedness is only a concern at the semantic level.

Note that for simplicity we use the same variable categories for label variables as we do for the graphs of rule schemata. There are however two new categories of variables: for nodes and edges.

Definition 6.2 (Many-sorted expressions)**.** The grammar in Figure 6.1 defines six syntactic categories of *expressions*: Edge, Vertex, Integer, String, Atom and List. These respectively contain disjoint syntactic categories of

variables: EVar, VVar, IVar, SVar, AVar, LVar. Note that we have the following inclusions among expressions:

$$\text{Integer} \cup \text{String} \subseteq \text{Atom} \subset \text{List}.$$

Expression	::=	Edge | Vertex | List
Edge	::=	EVar
Vertex	::=	VVar | (`s` | `t`) '(' Edge ')'
Integer	::=	Digit {Digit} | IVar | '−' Integer
		| Integer ArithOp Integer
ArithOp	::=	'+' | '−' | '*' | '/'
String	::=	'"' {Char} '"' | SVar | String '.' String
Atom	::=	Integer | String | AVar
List	::=	`empty` | Atom | LVar | `l` '(' Vertex ')' | `m` '(' Edge ')'
		| List ':' List

Figure 6.1: Abstract syntax of many-sorted expressions

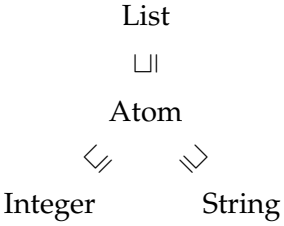
The symbol `empty` and symbols in Digit {Digit} and '"' {Char} '"' are *constant symbols*, here syntactic representations of the empty list, integers, and character strings. The symbols `s`, `t`, `l`, `m`, and −, are *function symbols* of arity one, and are syntactic representations source, target, node labelling, and edge labelling functions. The symbols `.`, :, and the symbols in ArithOp are function symbols of arity two, representing concatenation, appending lists, and the obvious arithmetic operations. (Note that the function symbol '−' is overloaded.) □

We attach different sorts (or types) to expressions generated from our grammar, depending on whether they are generated by Edge, Vertex, or List. For expressions in List, we have a hierarchy of sorts. For example, we attach different sorts to digits and character strings, while recognising that both remain lists. While this hierarchy is implicit from the grammar (for example, only expressions from Integer may be arguments to symbols in ArithOp, yet any two labels are allowed as arguments to ':'), we make the hierarchy explicit by a subsort relation, which proves to be useful in later definitions.

Definition 6.3 (Sorts, sort function, and subsort relation)**.** Every expression is associated with a *sort* (or *type*), determined by the smallest syntactic category it is contained within. Given an expression whose smallest containing syntactic category is Edge, Vertex, List, Atom, Integer, or String, we use the name of that category to denote its sort. The function:

$$\text{sort}\colon \text{Expression} \to \{\text{Edge}, \text{Vertex}, \text{List}, \text{Atom}, \text{Integer}, \text{String}\}$$

is the *sort function*, that takes an expression as input and returns its sort. Given two sorts s_1, s_2, we say that s_1 is a *subsort* of s_2, denoted $s_1 \sqsubseteq s_2$, if $s_1 = s_2$, or if $s_1 \sqsubseteq s_2$ in the following hierarchy (which is transitive):



□

Example 6.4 (Expression and sorts)**.** An example of an expression is:

$$\texttt{-1 : x."ab".y : n+2 : l(s(e))}$$

provided that $\texttt{x}, \texttt{y} \in$ SVar, $\texttt{n} \in$ IVar and $\texttt{e} \in$ EVar. The sort of the expression is List; `"ab"`, `1`, and `2` are constant symbols; and `-`, `:`, `.`, `+`, `l`, and `s` are function symbols.

On the other hand, the following:

$$\texttt{n+2 : n."a"}$$

is *not* an expression and cannot be generated by our grammar, because `n` cannot be both an integer and a string variable (the categories IVar and SVar are required to be distinct). □

The formulae of our logic are, like expressions, defined by a grammar. Again, the idea is to enforce at the syntactic level that formulae are well-typed. It is not possible, for example, to generate a formula $e_1 > e_2$ for edges e_1, e_2, since $>$ is later interpreted as a Boolean-valued function on integers.

Definition 6.5 (Many-sorted formulae)**.** Figure 6.2 defines *many-sorted formulae*, short *formulae*.

The symbol = and the symbols in IntRel are *predicate symbols* of arity two, and are syntactic representations of the obvious Boolean-valued functions. The symbols in Type and `marked` are predicate symbols of arity one, the former to be interpreted as in assignment constraints, and the latter to be interpreted as a Boolean function evaluating to true if the input item is marked. □

The operators and quantifiers are given the usual precedence, that is, $\neg, \vee, \wedge, \rightarrow, \leftrightarrow, \forall, \exists$ from high to low. Brackets are used to resolve ambiguity.

Formula	::=	`true` | `false` | Type '`(`' List '`)`' | Integer IntRel Integer
		| Edge '`=`' Edge | Vertex '`=`' Vertex | List '`=`' List
		| `marked` '`(`' (Edge | Vertex) '`)`'
		| '$\neg$' Formula | Formula BoolOp Formula
		| Quantifier (VVar '`:V`' | EVar '`:E`'
		| IVar '`:I`' | SVar '`:S`' | AVar '`:A`'
		| LVar '`:L`') '`.`' Formula
IntRel	::=	'`>`' | '`<`' | '`>=`' | '`<=`'
Type	::=	`int` | `string` | `atom`
BoolOp	::=	$\wedge$ | $\vee$ | $\Rightarrow$ | $\Leftrightarrow$
Quantifier	::=	$\forall$ | $\exists$

Figure 6.2: Abstract syntax of many-sorted formulae

Example 6.6 (Many-sorted formulae)**.** Examples of formulae are

$$\forall \texttt{v:V.}\ \exists \texttt{x:I.}\ \texttt{l(v)} = \texttt{x} * \texttt{x} \quad \text{and} \quad \exists \texttt{e:E.}\ \exists \texttt{x:I.}\ \texttt{m(e)} = \texttt{x} \wedge \texttt{x} < \texttt{0},$$

whereas $\exists$`e:E. m(e)` $<$ `0` is not a formula and cannot be generated by our grammar, because `m(e)` is not of sort Integer. □

The *free variables* of a formula are those that are not bound by a quantifier. Note that such variables still have sorts, determined by our grammar for expressions. If a formula contains no such free variables, then we call it a sentence.

Definition 6.7 (Sentence)**.** A *sentence* (or a *closed formula*) is a formula that contains no free variables. □

Expressions can be substituted for free variables. We give the definition of such substitutions, which follows standard logic textbooks other than for the requirement that the expression's sort is a subsort of the variable's sort. This requirement ensures that the resulting formula is well-typed. It prevents, for example, a character string to be substituted for an integer variable in the argument of an arithmetic operator, yet allows a variable of sort List to be replaced by an expression lower in the sort hierarchy (such as a string expression).

Note that substitutions for formulae differ from substitutions for E-conditions. In the latter, variables are simply replaced in-place regardless of context. In formulae, only free variables can be replaced – and sort matters – so context is important. To avoid confusing the two definitions, rather than using φ^σ to denote the application of a substitution, we write $\varphi[t/\mathtt{x}]$ specifying the particular replacements in square brackets.

Definition 6.8 (Substitution of free variables)**.** Let t be an expression of sort s_t, and $\mathtt{x}$ be a variable of sort $s_{\mathtt{x}}$ such that $s_t \sqsubseteq s_{\mathtt{x}}$.

Given an expression e, the result of substituting t for a variable $\mathtt{x}$ in e, denoted $e[t/\mathtt{x}]$, is defined inductively. If e is a variable $\mathtt{y}$, then $e[t/\mathtt{x}] = t$ if $\mathtt{x} = \mathtt{y}$, otherwise $e[t/\mathtt{x}] = e$. If e is a constant symbol c, then $e[t/\mathtt{x}] = c$. If e has the form $f(e_1, \ldots, e_n)$ with f a function symbol and each e_i an expression, then $e[t/\mathtt{x}] = f(e_1[t/\mathtt{x}], \ldots, e_n[t/\mathtt{x}])$.

Let φ be a formula. The result of substituting t for a variable $\mathtt{x}$ in φ, denoted $\varphi[t/\mathtt{x}]$, is defined inductively. If φ is `true` or `false`, then $\varphi[t/\mathtt{x}] = \varphi$. If φ is $p(e_1, \ldots, e_n)$ with p a predicate symbol and each e_i an expression, then $\varphi[t/\mathtt{x}] = p(e_1[t/\mathtt{x}], \ldots, e_n[t/\mathtt{x}])$. If φ is $\neg\psi$ with ψ in Formula, then $\varphi[t/\mathtt{x}] = \neg\psi[t/\mathtt{x}]$. If φ is $\psi_1 \oplus \psi_2$ with ψ_1, ψ_2 in Formula and $\oplus$ in BoolOp, then $\varphi[t/\mathtt{x}] = \psi_1[t/\mathtt{x}] \oplus \psi_2[t/\mathtt{x}]$. If φ is $Q\mathtt{y} : s\,.\ \psi$ with Q in Quantifier, $\mathtt{y}$ a variable, and ψ in Formula, then $\varphi[t/\mathtt{x}] = Q\mathtt{y} : s\,.\ \psi$ if $\mathtt{x} = \mathtt{y}$, otherwise $\varphi[t/\mathtt{x}] = Q\mathtt{y} : s\,.\ \psi[t/\mathtt{x}]$.

Let φ be a formula, $t_1, \ldots, t_n$ be expressions with each t_i having sort s_{t_i}, and $\mathtt{x}_1, \ldots, \mathtt{x}_n$ be variables with each $\mathtt{x}_i$ having sort $s_{\mathtt{x}_i}$ such that $s_{t_i} \sqsubseteq s_{\mathtt{x}_i}$. Then, the result of the substitution $\varphi[t_1/\mathtt{x}_1, \ldots, t_n/\mathtt{x}_n]$ is given by:

$$\varphi[t_1/\mathtt{x}_1][t_2/\mathtt{x}_2] \cdots [t_n/\mathtt{x}_n].$$

□

To simplify the presentation of formulae with multiple quantified variables, we let $\exists \mathtt{x}_1, \mathtt{x}_2, \ldots \mathtt{x}_n : s$ abbreviate $\exists \mathtt{x}_1 : s\,.\ \exists \mathtt{x}_2 : s\,.\ \ldots \exists \mathtt{x}_n : s$ where all variables are of the same sort. In addition, we let $\exists \mathtt{x}_1 : s_1, \mathtt{x}_2 : s_2, \ldots \mathtt{x}_n : s_n$ abbreviate the formula $\exists \mathtt{x}_1 : s_1\,.\ \exists \mathtt{x}_2 : s_2\,.\ \ldots \exists \mathtt{x}_n : s_n$ where the sorts of variables need not be the same. An analogous notation is permitted for $\forall$.

6.2.2 Semantics of Sentences

Our logic is given its semantics by interpretation functions. We begin by defining what an interpretation function is, before defining what is means for such a function to satisfy a sentence of our logic. Then, to allow sentences to be interpreted over a particular graph (i.e. to define the meaning of $G \models \varphi$), we define the notion of interpretation functions induced by graphs, which capture information about their structure and labelling.

Definition 6.9 (Satisfaction of sentences)**.** An *interpretation function* I is a mapping from (1) sorts to semantic domains, (2) constants of sort s to elements in $I(s)$, (3) expressions $f(e_1, \ldots, e_n)$, with f a function symbol and each e_i an expression, to functions of arity:

$$I(\text{sort}(e_1)) \times \cdots \times I(\text{sort}(e_n)) \to I(\text{sort}(f(e_1, \ldots, e_n))),$$

and (4) formulae $p(e_1, \ldots, e_n)$, with p a predicate symbol and each e_i an expression, to Boolean-valued functions of arity:

$$I(\text{sort}(e_1)) \times \cdots \times I(\text{sort}(e_n)) \to \mathbb{B}$$

where $\mathbb{B} = \{\text{true}, \text{false}\}$.

Let I be an interpretation function, and φ be a sentence. The *satisfaction* of φ by I, denoted $I \models \varphi$, is defined inductively as follows.

If φ is `true` (resp. `false`), then $I \models \varphi$ (resp. $I \models \varphi$ does not hold). If φ is $p(e_1, \ldots, e_n)$ with p a predicate symbol and each e_i an expression, then $I \models \varphi$ if $I(p)(I(e_1), \ldots, I(e_n)) = \text{true}$.

Let φ_1, φ_2 be sentences. If φ is $\neg\varphi_1$, then $I \models \varphi$ if $I \models \varphi_1$ does not hold. If φ is $\varphi_1 \wedge \varphi_2$ (resp. $\varphi_1 \vee \varphi_2$), then $I \models \varphi$ if $I \models \varphi_1$ and (resp. or) $I \models \varphi_2$. If φ is $\varphi_1 \to \varphi_2$, then $I \models \varphi$ if $I \models \neg\varphi_1$ or $I \models \varphi_2$. If φ is $\varphi_1 \leftrightarrow \varphi_2$, then $I \models \varphi$ if $I \models \varphi_1 \to \varphi_2$ and $I \models \varphi_2 \to \varphi_1$.

Let `x` be a variable of sort s, and φ_1 be a formula with `x` as its only free variable. Let also S denote the symbol that corresponds with sort s. If φ has the form $\exists \texttt{x} : S \texttt{.}\ \varphi_1$, then $I \models \varphi$ if there is some $a \in I(s)$ such that $I_{\texttt{x} \mapsto a} \models \varphi_1$ where $I_{\texttt{x} \mapsto a}$ is equal to I but with the addition that $I(\texttt{x}) = a$. If φ is $\forall \texttt{x} : S \texttt{.}\ \varphi_1$, then $I \models \varphi$ if for every $a \in I(s)$, $I_{\texttt{x} \mapsto a} \models \varphi_1$.

□

Definition 6.10 (Satisfaction of sentences by graphs)**.** Let G be a graph and φ be a sentence. We say that G *satisfies* φ, denoted by $G \models \varphi$, if $I_G \models \varphi$, where I_G is the *interpretation function induced by* G, defined as follows:

Sorts. We define $I_G(\text{Edge}) = E_G$, $I_G(\text{Vertex}) = V_G$, $I_G(\text{List}) = \mathbb{L}$, $I_G(\text{Atom}) = \mathbb{Z} \cup \text{Char}^*$, $I_G(\text{Integer}) = \mathbb{Z}$, and $I_G(\text{String}) = \text{Char}^*$.

Constant symbols. For a constant symbol c, $I_G(c)$ is the value in $\mathbb{L}$ corresponding to c[2].

Function symbols. We define $I_G(\texttt{s}) = s_G$ and $I_G(\texttt{t}) = t_G$. We define $I_G(\texttt{l})$ and $I_G(\texttt{m})$ to be the functions l_G and m_G respectively but returning the list component only. For $\oplus$ in ArithOp, $I_G(\oplus) = \oplus_{\mathbb{Z}}$ where $\oplus_{\mathbb{Z}}$ denotes the obvious operation on integers corresponding to $\oplus$. For the arity one function symbol $-$, we define $I_G(-)$ to be the function that returns the value of its integer input multiplied by -1. We define $I_G(\texttt{.})$ to be the function that returns the string resulting from the concatenation of its first argument followed by its second argument. Finally, we define $I_G(\texttt{:}) = \text{append}$, where

$$\text{append} : \mathbb{L} \times \mathbb{L} \to \bigcup_{n>1} (\mathbb{Z} \cup \text{Char}^*)^n$$

returns the list resulting from appending its second argument to the end of its first argument (i.e. a so-called tagged label).

[2]That is, $I_G(c)$ is the value c^α for any assignment α.

Predicate symbols. We define $I_G(=)$ to be equality in the standard sense. For $\bowtie$ in IntRel, $I_G(\bowtie) = \bowtie_{\mathbb{B}}$ where $\bowtie_{\mathbb{B}}$ denotes the obvious Boolean-valued function on integers corresponding to $\bowtie$. We define $I_G(\texttt{marked})$ to be the function that returns true if the mark component of its input is true, and false otherwise. We define $I_G(\texttt{int}) = \text{int}$, $I_G(\texttt{string}) = \text{string}$, and finally $I_G(\texttt{atom}) = \text{atom}$ where $\text{int}, \text{string}, \text{atom} : \mathbb{L} \to \mathbb{B}$ returns true if its input is in $I_G(\text{Integer})$ (resp. $I_G(\text{String})$, $I_G(\text{Atom})$), otherwise it returns false.

□

Validity of sentences is defined analogously to validity of E-constraints, i.e. a sentence is valid if every graphs in $\mathcal{G}(\mathcal{L})$ satisfies that sentence.

Definition 6.11 (Valid sentences)**.** Let φ be a sentence. We say that φ is *valid*, denoted $\models \varphi$, if for all graphs $G \in \mathcal{G}(\mathcal{L})$, $G \models \varphi$.

□

6.3 From Many-Sorted Formulae to E-Conditions

We prove in this section that every sentence of our logic has an equivalent E-constraint, by defining and proving correct the transformation Cond from formulae to E-conditions. We also demonstrate the transformation by applying it to two example formulae and showing the steps.

The transformation is defined inductively for all the possible forms that a formula may take, according to the grammar of Figure 6.2. It is more general than the corresponding transformation in [HP09]: this translated a single-sorted logic with a finite alphabet into nested conditions; here, we must handle an infinite label alphabet, the expressions of E-conditions, and the fact that the morphisms of E-conditions are injective. As a result, our transformation and technique differs in a number of ways. We can exploit the sorts of variables to simplify the E-conditions we generate (avoiding situations like in the single-sorted logic when a variable may be interpreted as either a node or an edge). However, because the morphisms are injective, we cannot use the technique of identifying items in the codomain when `m = n` appears in the logic. Instead, when a node (or edge) is declared in a formula, the corresponding E-condition is a disjunction anticipating all possible identifications of nodes (or edges). If later in the formula there is an identification such as `m = n`, the corresponding E-condition is then simply true or false depending on the context (i.e. disjunct) it is considered within.

A corollary resulting from our transformation is that first-order arithmetic can be expressed by E-conditions (the same is not true for nested conditions), which may be helpful in investigating the expressiveness of E-conditions as an assertion language (an expressiveness proof in [Win93] for a different assertion language relies on using arithmetic to encode assertions).

Before we define the construction of E-conditions from sentences, we remark upon an important correspondence it uses between variables in formulae, node and edge identifiers, and the variables in the list components of those items. Then, we define two helper functions that the construction uses.

Remark 6.12 (Corresponding variables, identifiers, and labels)**.** A key idea in the transformation from many-sorted formulae to E-conditions is to construct and exploit a correspondence between node and edge variables in formulae, node and edge identifiers in the graphs of the morphisms, and the labels of those nodes and edges (which are just variables in VarId). The typeface and whether or not the symbol is primed, in the context of this transformation, implies which part of the correspondence it refers to. The table below, for example, summarises the symbols used in the transformation when a node or edge variable $\mathtt{x}$ is declared in a formula:

Symbol	*Correspondence*
$\mathtt{x}$	node or edge sorted variable in formula
x	node or edge identifier in graph
$\mathtt{x}'$	variable in VarId labelling x

Figure 6.3 shows this correspondence for nodes. For edge variables $\mathtt{x}$, we additionally use the symbols s_x, t_x as the identifiers of the edge's source and target nodes, and $\mathtt{s}'_x, \mathtt{t}'_x$ to denote respectively their list components in VarId. This correspondence is shown by Figure 6.4.

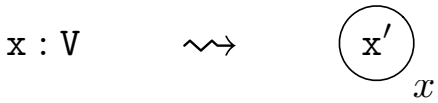


Figure 6.3: Correspondence between node variables and nodes

Figure 6.4: Correspondence between edge variables and edges

□

The first helper function VertexID takes a Vertex expression as input, returning the node identifier that would be associated with it in a graph exhibiting the correspondence described in Remark 6.12.

Definition 6.13 (Helper function VertexID)**.** Let t denote an expression in Vertex. We define:

$$\text{VertexID}(t) = \begin{cases} v & \text{if } t = \texttt{v} \text{ with } \texttt{v} \in \text{VVar} \\ s_e & \text{if } t = \texttt{s(e)} \text{ with } \texttt{e} \in \text{EVar} \\ t_e & \text{if } t = \texttt{t(e)} \text{ with } \texttt{e} \in \text{EVar} \end{cases}$$

□

The second helper function AC (for Assignment Constraint) takes as input a many-sorted expression from List, returning an equivalent assignment constraint but with labelling functions (unavailable in assignment constraints) replaced by the variables labelling items according to the correspondence of Remark 6.12.

Definition 6.14 (Helper function AC)**.** Let t denote a many-sorted expression in List. Then we define:

$$\text{AC}(t) = \begin{cases} \texttt{v}' & \text{if } t = \texttt{l(v)} \text{ with } \texttt{v} \in \text{VVar} \\ \texttt{s}'_e & \text{if } t = \texttt{l(s(e))} \text{ with } \texttt{e} \in \text{EVar} \\ \texttt{t}'_e & \text{if } t = \texttt{l(t(e))} \text{ with } \texttt{e} \in \text{EVar} \\ \texttt{e}' & \text{if } t = \texttt{m(e)} \text{ with } \texttt{e} \in \text{EVar} \\ \text{AC}(t_1) : \text{AC}(t_2) & \text{if } t = t_1 : t_2 \text{ with } t_1, t_2 \in \text{List} \\ t & \text{otherwise} \end{cases}$$

□

Now, we are ready to define the transformation Cond from many-sorted formulae to E-constraints. We require that the formulae do not contain any variables that are primed (since these have special meaning in the transformation), and require also that the same variable is not quantified more than once. It is easy to rename variables to get arbitrary formulae into this form.

Theorem 6.15 (Sentences can be expressed as E-constraints)**.** Let φ denote a sentence without primed variables and without variables quantified more than once. There is a transformation Cond such that for all graphs $G \in \mathcal{G}(\mathcal{L})$,

$$G \models \varphi \text{ if and only if } G \models \text{Cond}(\varphi).$$

Construction. For all sentences φ, let $\text{Cond}(\varphi) = \text{Cond}'(\varphi, \emptyset)$. The transformation Cond' takes the formula that remains to be translated as its first input, and the domain of the next morphism in the generated E-condition as its second input. We define it inductively over the abstract syntax of formulae (Figure 6.2) and expressions (Figure 6.1).

Let X denote a graph in $G \in \mathcal{G}(\text{EC})$. Let $\varphi', \varphi_1, \varphi_2$ denote formulae (not necessarily sentences).

If $\varphi = \texttt{true}$ (resp. $\texttt{false}$), then $\mathrm{Cond}'(\varphi, X) = \texttt{true}$ (resp. $\texttt{false}$). If $\varphi = \neg\varphi'$, then $\mathrm{Cond}'(\varphi, X) = \neg\mathrm{Cond}'(\varphi', X)$. If $\varphi = \varphi_1 \oplus \varphi_2$ with $\oplus \in \mathrm{BoolOp}$, then $\mathrm{Cond}'(\varphi, X) = \mathrm{Cond}'(\varphi_1, X) \oplus \mathrm{Cond}'(\varphi_2, X)$.

If $\varphi = \exists \texttt{x} : \texttt{L}.\ \varphi'$, then

$$\mathrm{Cond}'(\varphi, X) = \exists(X \hookrightarrow X \mid \texttt{bound(x)}, \mathrm{Cond}'(\varphi', X))$$

where $\texttt{bound(x)}$ is an alias for $\texttt{x} = \texttt{x}$. (This assignment constraint is trivially satisfied by any assignment to $\texttt{x}$, but more importantly, forces satisfying assignments to have a mapping for $\texttt{x}$, which is then equally assigned down the nesting. This is necessary if the nested part is a Boolean formula over E-conditions.)

If $\varphi = \exists \texttt{x} : S.\ \varphi'$ with $S \in \{\texttt{I}, \texttt{S}, \texttt{A}\}$, then:

$$\mathrm{Cond}'(\varphi, X) = \exists(X \hookrightarrow X \mid t\texttt{(x)}, \mathrm{Cond}'(\varphi', X))$$

where t is respectively $\texttt{int}$, $\texttt{string}$, or $\texttt{atom}$ for the three sorts.

If $\varphi = \exists \texttt{v} : \texttt{V}.\ \varphi'$, then:

$$\mathrm{Cond}'(\varphi, X) = \bigvee_{X' \in \mathrm{VMerge}(X, \texttt{v})} \exists(X \hookrightarrow X' \mid \texttt{v}' = l_{X'}(v), \mathrm{Cond}'(\varphi', X'))$$

where $l_{X'}$ returns the list component only. Here, $\mathrm{VMerge}(X, \texttt{v})$ is the (finite) set of graphs in $\mathcal{G}(\mathrm{EC})$ containing exactly the graph obtained from X by disjointly adding an unmarked node v, the graph obtained from X by disjointly adding a marked node v, and every graph obtainable from these by identifying a node with v; in the two cases with no merging, label v with variable $\texttt{v}'$. (Observe that the assignment constraint – in cases when v is identified with another node – enforces that $\texttt{v}'$ is always the same as the list component of that node.)

If $\varphi = \exists \texttt{e} : \texttt{E}.\ \varphi'$, then:

$$\mathrm{Cond}'(\varphi, X) = \bigvee_{X' \in \mathrm{EMerge}(X, \texttt{e})} \exists(X \hookrightarrow X' \mid \texttt{e}' = m_{X'}(e), \mathrm{Cond}'(\varphi', X'))$$

where $m_{X'}$ returns the list component only, and where $\mathrm{EMerge}(X, \texttt{e})$ is the (finite) set of graphs in $\mathcal{G}(\mathrm{EC})$ defined as follows. Let X^* denote a graph obtained from X by disjointly adding nodes with identifiers s_e, t_e and an edge with identifier e such that $s_{X^*}(e) = s_e$, $t_{X^*}(e) = t_e$, $l_{X^*}(s_e) = \texttt{s}'_e$, $l_{X^*}(t_e) = \texttt{t}'_e$, and $m_{X^*}(e) = \texttt{e}'$ (these labels being distinct variables in VarId). The set $\mathrm{EMerge}(X, \texttt{e})$ contains all such graphs X^* for all combinations of mark components, and all other graphs obtainable from each X^* by identifying e, s_e, t_e with all possible nodes and edges – with their list

components replacing $\mathtt{e}'$, $\mathtt{s}'_e$, $\mathtt{t}'_e$. (Note that s_e and t_e can also be identified, if their mark components are the same, to create a loop.)

If $\varphi = \forall \mathtt{x} : S \,.\, \varphi'$, then:

$$\text{Cond}'(\varphi, X) = \text{Cond}'(\neg\exists \mathtt{x} : S \,.\, \neg\varphi', X).$$

If $\varphi = \mathtt{e} = \mathtt{f}$ with $\mathtt{e}, \mathtt{f} \in \text{EVar}$, then:

$$\text{Cond}'(\varphi, X) = \mathtt{true}$$

if edges e, f are identified in X, otherwise false.

If $\varphi = v_1 = v_2$ with v_1, v_2 in Vertex, then:

$$\text{Cond}'(\varphi, X) = \mathtt{true}$$

if VertexID(v_1), VertexID(v_2) are identified in X, otherwise $\text{Cond}'(\varphi, X) = \mathtt{false}$.

If $\varphi = \mathtt{marked(e)}$ with $\mathtt{e}$ in EVar (resp. $\mathtt{marked}(v)$ with v in Vertex), then $\text{Cond}'(\varphi, X) = \mathtt{true}$ if e in E_X is marked (resp. VertexID(v) in V_X is marked), and $\mathtt{false}$ otherwise.

If $\varphi = t\,(l)$ where $t \in \text{Type}$ and $l \in \text{List}$. Then,

$$\text{Cond}'(\varphi, X) = \exists(X \hookrightarrow X \mid t\,(\text{AC}(l))\,).$$

If $\varphi = i_1 \bowtie i_2$ with $i_1, i_2 \in \text{Integer}$ and $\bowtie \in \text{IntRel}$, then:

$$\text{Cond}'(\varphi, X) = \exists(X \hookrightarrow X \mid i_1 \bowtie i_2).$$

Finally, if $\varphi = l_1 = l_2$ with $l_1, l_2 \in \text{List}$, then:

$$\text{Cond}'(\varphi, X) = \exists(X \hookrightarrow X \mid \text{AC}(l_1) = \text{AC}(l_2)).$$

□

We demonstrate the construction of Cond step-by-step on two examples of sentences. The second of the examples is more complicated, quantifying more than one node variable and using node identification.

Example 6.16. Consider the sentence:

$$\varphi = \neg\exists \mathtt{v} : \mathtt{V} \,.\, \exists \mathtt{i} : \mathtt{I} \,.\, \mathtt{l(v)} = \mathtt{i} \wedge \mathtt{i} < \mathtt{0}.$$

For a graph to satisfy φ, it must not contain a node that has a negative integer as its list component.

Applying Cond to φ results in the following E-constraint:

$$
\begin{array}{rcl}
\mathrm{Cond}(\varphi) & = & \mathrm{Cond}'(\varphi, \emptyset) \\
& = & \neg\mathrm{Cond}'(\exists \mathtt{v} : \mathtt{V}.\ \exists \mathtt{i} : \mathtt{I}.\ \mathtt{l(v)} = \mathtt{i} \wedge \mathtt{i} < 0, \emptyset) \\
& = & \neg\exists(\ \textcircled{\mathtt{v}'}_v \mid \mathtt{v}' = \mathtt{v}', \mathrm{Cond}'(\exists \mathtt{i} : \mathtt{I}.\ \mathtt{l(v)} = \mathtt{i} \wedge \mathtt{i} < 0,\ \textcircled{\mathtt{v}'}_v)) \\
& & \wedge \neg\exists(\ \textcircled{\mathtt{v}'}_v \mid \mathtt{v}' = \mathtt{v}', \mathrm{Cond}'(\exists \mathtt{i} : \mathtt{I}.\ \mathtt{l(v)} = \mathtt{i} \wedge \mathtt{i} < 0,\ \textcircled{\mathtt{v}'}_v)) \\
& = & \neg\exists(\ \textcircled{\mathtt{v}'}_v \mid \mathtt{v}' = \mathtt{v}', \exists(\ \textcircled{\mathtt{v}'}_v \mid \mathtt{int(i)}, \\
& & \quad \mathrm{Cond}'(\mathtt{l(v)} = \mathtt{i} \wedge \mathtt{i} < 0,\ \textcircled{\mathtt{v}'}_v))) \\
& & \wedge \neg\exists(\ \textcircled{\mathtt{v}'}_v \mid \mathtt{v}' = \mathtt{v}', \exists(\ \textcircled{\mathtt{v}'}_v \mid \mathtt{int(i)}, \\
& & \quad \mathrm{Cond}'(\mathtt{l(v)} = \mathtt{i} \wedge \mathtt{i} < 0,\ \textcircled{\mathtt{v}'}_v))) \\
& = & \neg\exists(\ \textcircled{\mathtt{v}'}_v \mid \mathtt{v}' = \mathtt{v}', \exists(\ \textcircled{\mathtt{v}'}_v \mid \mathtt{int(i)}, \\
& & \quad \mathrm{Cond}'(\mathtt{l(v)} = \mathtt{i},\ \textcircled{\mathtt{v}'}_v) \wedge \mathrm{Cond}'(\mathtt{i} < 0,\ \textcircled{\mathtt{v}'}_v))) \\
& & \wedge \neg\exists(\ \textcircled{\mathtt{v}'}_v \mid \mathtt{v}' = \mathtt{v}', \exists(\ \textcircled{\mathtt{v}'}_v \mid \mathtt{int(i)}, \\
& & \quad \mathrm{Cond}'(\mathtt{l(v)} = \mathtt{i},\ \textcircled{\mathtt{v}'}_v) \wedge \mathrm{Cond}'(\mathtt{i} < 0,\ \textcircled{\mathtt{v}'}_v))) \\
& = & \neg\exists(\ \textcircled{\mathtt{v}'}_v \mid \mathtt{v}' = \mathtt{v}', \exists(\ \textcircled{\mathtt{v}'}_v \mid \mathtt{int(i)}, \\
& & \quad \exists(\ \textcircled{\mathtt{v}'}_v \mid \mathtt{v}' = \mathtt{i}) \wedge \exists(\ \textcircled{\mathtt{v}'}_v \mid \mathtt{i} < 0))) \\
& & \wedge \neg\exists(\ \textcircled{\mathtt{v}'}_v \mid \mathtt{v}' = \mathtt{v}', \exists(\ \textcircled{\mathtt{v}'}_v \mid \mathtt{int(i)}, \\
& & \quad \exists(\ \textcircled{\mathtt{v}'}_v \mid \mathtt{v}' = \mathtt{i}) \wedge \exists(\ \textcircled{\mathtt{v}'}_v \mid \mathtt{i} < 0)))
\end{array}
$$

Several simplifications can clearly be made to yield an equivalent, and more readable E-constraint:

$$\equiv \neg\exists(\ \textcircled{\mathtt{i}}\ \mid \mathtt{i} < 0) \wedge \neg\exists(\ \textcircled{\mathtt{i}}\ \mid \mathtt{i} < 0).$$

This can be read as: "there does not exist a node with an integer less than zero as its list component". Here, `int(i)` is not required, since only assignments of `i` to integers are well-typed for `i < 0`.

□

Example 6.17. Consider the sentence:

$$
\begin{array}{rcl}
\varphi & = & \exists \mathtt{v} : \mathtt{V}.\ \mathtt{l(v)} = 5 \wedge \forall \mathtt{w} : \mathtt{V}. \\
& & \quad (\neg \mathtt{v} = \mathtt{w} \Rightarrow \exists \mathtt{x} : \mathtt{S}.\ \mathtt{l(w)} = \mathtt{x}) \wedge (\mathtt{l(w)} = \mathtt{l(v)} \Rightarrow \mathtt{v} = \mathtt{w}).
\end{array}
$$

For a graph to satisfy φ, it must be the case that there is a node v labelled by the integer 5, and that for any other node, being distinct from v implies that it is labelled by a string, and having the same label as v implies that it is the same node as v.

Define the following abbreviations:

$$\varphi'_1 = (\neg \mathtt{v} = \mathtt{w} \Rightarrow \exists \mathtt{x} : \mathtt{S}.\ \mathtt{l(w)} = \mathtt{x})$$

$$\varphi'_2 = (\mathtt{l(w)} = \mathtt{l(v)} \Rightarrow \mathtt{v} = \mathtt{w})$$

$$\varphi' = \forall \mathtt{w} : \mathtt{V}.\ \varphi'_1 \wedge \varphi'_2.$$

Applying the transformation Cond to φ results in the E-constraint of Figure 6.5 (note that we exclude the marked nodes from the disjunctions for simplicity of presentation).

$$
\begin{array}{rcl}
\mathrm{Cond}'(\varphi, \emptyset) & = & \exists(\ \textcircled{v'}_v \mid \mathtt{v'} = \mathtt{v}, \mathrm{Cond}'(\mathtt{l\,(v)} = 5,\ \textcircled{v'}_v) \\
& & \quad \wedge \mathrm{Cond}'(\varphi',\ \textcircled{v'}_v)) \\
& = & \exists(\ \textcircled{v'}_v \mid \mathtt{v'} = \mathtt{v}, \exists(\ \textcircled{v'}_v \mid \mathtt{v'} = 5) \\
& & \quad \wedge \neg\exists(\ \textcircled{v'}_v\ \textcircled{w'}_w \mid \mathtt{w'} = \mathtt{w}, \neg\mathrm{Cond}'(\varphi'_1 \wedge \varphi'_2,\ \textcircled{v'}_v\ \textcircled{w'}_w)) \\
& & \quad \wedge \neg\exists(\ \textcircled{v'}_{v=w} \mid \mathtt{w'} = \mathtt{v}, \neg\mathrm{Cond}'(\varphi'_1 \wedge \varphi'_2,\ \textcircled{v'}_{v=w}))) \\
& = & \exists(\ \textcircled{v'}_v \mid \mathtt{v'} = \mathtt{v}, \exists(\ \textcircled{v'}_v \mid \mathtt{v'} = 5) \\
& & \quad \wedge \neg\exists(\ \textcircled{v'}_v\ \textcircled{w'}_w \mid \mathtt{w'} = \mathtt{w}, \neg(\\
& & \qquad (\neg\mathtt{false} \Rightarrow \mathrm{Cond}'(\exists \mathtt{x} : \mathtt{S.\ l\,(w)} = \mathtt{x},\ \textcircled{v'}_v\ \textcircled{w'}_w)) \\
& & \qquad \wedge(\exists(\ \textcircled{v'}_v\ \textcircled{w'}_w \mid \mathtt{w'} = \mathtt{v'}) \Rightarrow \mathtt{false}))) \\
& & \quad \wedge \neg\exists(\ \textcircled{v'}_{v=w} \mid \mathtt{w'} = \mathtt{v}, \neg\mathrm{Cond}'(\varphi'_1 \wedge \varphi'_2,\ \textcircled{v'}_{v=w}))) \\
& = & \exists(\ \textcircled{v'}_v \mid \mathtt{v'} = \mathtt{v}, \exists(\ \textcircled{v'}_v \mid \mathtt{v'} = 5) \\
& & \quad \wedge \neg\exists(\ \textcircled{v'}_v\ \textcircled{w'}_w \mid \mathtt{w'} = \mathtt{w}, \neg(\\
& & \qquad (\neg\mathtt{false} \Rightarrow \mathrm{Cond}'(\exists \mathtt{x} : \mathtt{S.\ l\,(w)} = \mathtt{x},\ \textcircled{v'}_v\ \textcircled{w'}_w)) \\
& & \qquad \wedge (\exists(\ \textcircled{v'}_v\ \textcircled{w'}_w \mid \mathtt{w'} = \mathtt{v'}) \Rightarrow \mathtt{false}))) \\
& & \quad \wedge \neg\exists(\ \textcircled{v'}_{v=w} \mid \mathtt{w'} = \mathtt{v}, \neg(\\
& & \qquad (\neg\mathtt{true} \Rightarrow \mathrm{Cond}'(\exists \mathtt{x} : \mathtt{S.\ l\,(w)} = \mathtt{x},\ \textcircled{v'}_{v=w})) \\
& & \qquad \wedge (\exists(\ \textcircled{v'}_{v=w} \mid \mathtt{w'} = \mathtt{v'}) \Rightarrow \mathtt{true})))) \\
& = & \exists(\ \textcircled{v'}_v \mid \mathtt{v'} = \mathtt{v}, \exists(\ \textcircled{v'}_v \mid \mathtt{v'} = 5) \\
& & \quad \wedge \forall(\ \textcircled{v'}_v\ \textcircled{w'}_w \mid \mathtt{w'} = \mathtt{w}, \\
& & \qquad \exists(\ \textcircled{v'}_v\ \textcircled{w'}_w \mid \mathtt{string\,(x)}, \exists(\ \textcircled{v'}_v\ \textcircled{w'}_w \mid \mathtt{w'} = \mathtt{x})) \\
& & \qquad \wedge \neg\exists(\ \textcircled{v'}_v\ \textcircled{w'}_w \mid \mathtt{w'} = \mathtt{v'})) \\
& & \quad \wedge \forall(\ \textcircled{v'}_{v=w} \mid \mathtt{w'} = \mathtt{v'}, \\
& & \qquad (\mathtt{true} \vee \exists(\ \textcircled{v'}_{v=w} \mid \mathtt{string\,(x)}, \exists(\ \textcircled{v'}_{v=w} \mid \mathtt{w'} = \mathtt{x}))) \\
& & \qquad \wedge (\neg\exists(\ \textcircled{v'}_{v=w} \mid \mathtt{w'} = \mathtt{v'}) \vee \mathtt{true}))) \\
& = & \exists(\ \textcircled{v'}_v \mid \mathtt{v'} = \mathtt{v}, \exists(\ \textcircled{v'}_v \mid \mathtt{v'} = 5) \\
& & \quad \wedge \forall(\ \textcircled{v'}_v\ \textcircled{w'}_w \mid \mathtt{w'} = \mathtt{w}, \\
& & \qquad \exists(\ \textcircled{v'}_v\ \textcircled{w'}_w \mid \mathtt{string\,(x)}, \exists(\ \textcircled{v'}_v\ \textcircled{w'}_w \mid \mathtt{w'} = \mathtt{x})) \\
& & \qquad \wedge \neg\exists(\ \textcircled{v'}_v\ \textcircled{w'}_w \mid \mathtt{w'} = \mathtt{v'})))
\end{array}
$$

Figure 6.5: Derivation of the E-condition $\mathrm{Cond}(\varphi)$ from Example 6.17

The E-constraint of Figure 6.5 can be simplified further to the equivalent E-constraint:

$$\equiv \exists(\ \textcircled{5}_v, \forall(\ \textcircled{5}_v\ \textcircled{w'}_w, \exists(\ \textcircled{5}_v\ \textcircled{w'}_w \mid \mathtt{string\,(w')}\ ))).$$

This can be read as: "exactly one node in the graph is labelled with the integer 5, and all other nodes are labelled with strings".

Including marked nodes, the final (simplified) E-constraint would be:

$$
\begin{array}{rl}
\equiv & \exists(\ \textcircled{5}_v, \forall(\ \textcircled{5}_v\ \textcircled{w'}_w, \exists(\ \textcircled{5}_v\ \textcircled{w'}_w \mid \mathtt{string\,(w')}\ )) \\
& \quad \wedge \forall(\ \textcircled{5}_v\ \textcircled{\mathbf{w'}}_w, \exists(\ \textcircled{5}_v\ \textcircled{\mathbf{w'}}_w \mid \mathtt{string\,(w')}\ ))) \\
& \exists(\ \textcircled{\mathbf{5}}_v, \forall(\ \textcircled{\mathbf{5}}_v\ \textcircled{w'}_w, \exists(\ \textcircled{\mathbf{5}}_v\ \textcircled{w'}_w \mid \mathtt{string\,(w')}\ )) \\
& \quad \wedge \forall(\ \textcircled{5}_v\ \textcircled{\mathbf{w'}}_w, \exists(\ \textcircled{5}_v\ \textcircled{\mathbf{w'}}_w \mid \mathtt{string\,(w')}\ )))
\end{array}
$$

□

In order to prove the statement of Theorem 6.15 to be correct, we must first generalise it so that we can apply structural induction. (The formulae

in the theorem are sentences, but their sub-formulae need not be.) Hence we prove that many-sorted formulae in general can be expressed as E-conditions. The difference is that the interpretation function for the former maps free variables to particular nodes, edges, and labels, and must correspond to the domain of the morphism satisfying the latter. Then, the correctness of Theorem 6.15 follows from the fact that it is just a particular instance, i.e. no free variables and the empty graph as the domain of the morphism.

Lemma 6.18 (Formulae can be expressed as E-conditions)**.** Let φ denote a formula without primed variables and variables quantified more than once. For all injective graph morphisms $z\colon X^\alpha \hookrightarrow G$ with X, α corresponding to the free node and edge variables of φ as in Remark 6.12, and α a well-typed assignment such that no variable in $\mathrm{dom}(\alpha)$ is quantified in φ, we have that:

$$I_G^{z,\alpha} \models \varphi \text{ if and only if } z\colon X^\alpha \hookrightarrow G \models \mathrm{Cond}'(\varphi, X)^{\sigma_\alpha}.$$

Here, $I_G^{z,\alpha}$ is defined as I_G but with the following mappings for free variables in φ: (1) for each variable $\mathtt{x}$ in $\mathrm{dom}(\alpha)$, $I_G^{z,\alpha}(\mathtt{x}) = \alpha(\mathtt{x})$; (2) for each node v in X, $I_G^{z,\alpha}(\mathtt{v}) = z(v)$; and (3) for each edge e in X, $I_G^{z,\alpha}(\mathtt{e}) = z(e)$. □

Proof. See Appendix D.1. □

Now, we prove the correctness of Theorem 6.15 by showing that it is simply an instance of Lemma 6.18.

Proof of Theorem 6.15. We prove the correctness of the theorem by showing that it is an instance of Lemma 6.18. Define $i\colon \emptyset \hookrightarrow G$ and α_ϵ the assignment undefined on all variables, i.e. $\mathrm{dom}(\alpha_\epsilon) = \emptyset$. Then, by the definition of $\models$, the observation that $I_G = I_G^{i,\alpha_\epsilon}$, and Lemma 6.18, we have that:

$$\begin{aligned} G \models \varphi \text{ iff } & I_G \models \varphi \\ \text{iff } & I_G^{i,\alpha_\epsilon} \models \varphi \\ \text{iff } & i\colon \emptyset^{\alpha_\epsilon} \hookrightarrow G \models \mathrm{Cond}'(\varphi, \emptyset)^{\sigma_{\alpha_\epsilon}} \\ \text{iff } & i\colon \emptyset \hookrightarrow G \models \mathrm{Cond}'(\varphi, \emptyset) \\ \text{iff } & G \models \mathrm{Cond}(\varphi). \end{aligned}$$

□

The result that formulae can be transformed into E-conditions means also that E-conditions can express arbitrary sentences of first-order arithmetic (i.e. sentences in which all variables are of sort Integer). Arithmetic expressions and relations appear in the assignment constraints of such E-conditions, and nesting is used to handle variable quantification; the morphisms are all from $\emptyset$ to $\emptyset$, meaning that if an E-condition is satisfied by one graph, it is satisfied by all graphs.

Corollary 6.19 (E-constraints can express first-order arithmetic)**.** For all sentences φ containing only variables of sort Integer, there is an E-constraint c such that:

$$\models \varphi \text{ if and only if } \models c.$$

Construction. For all sentences φ, take $\text{Cond}(\varphi)$ to be c. □

Proof. Since the variables of φ can only be interpreted as integers, and hence the sentence cannot express anything about graph structure, if one graph satisfies φ then so do all graphs (the semantics of lists are fixed across all graphs). The same is true for the satisfaction of $c = \text{Cond}(\varphi)$, since by the construction of the transformation, the resulting E-condition's morphisms are all $\emptyset \hookrightarrow \emptyset$, and there is an injective graph morphism from the empty graph to any graph (in categorical terms it is an initial object). Together with Theorem 6.15 we have the result. □

6.4 From E-Conditions to Many-Sorted Formulae

We prove in this section that every E-constraint has an equivalent sentence in our many-sorted logic, by defining and proving correct a transformation from E-conditions to logic. We demonstrate the transformation by applying it to two example E-constraints and showing the steps.

To simplify the transformation, we define for E-conditions a normal form, and show how arbitrary E-conditions can be rewritten as equivalent ones that adhere to it. The normal form isolates expressions to the assignment constraints, substitutes `false` for E-conditions that have no well-typed assignment, and also decomposes the morphisms such that each level of nesting adds only one node or edge at a time. The idea is that each level of nesting corresponds to the quantification of a new node or edge variable.

6.4.1 Normal Form for E-Conditions

We propose a *normal form* for E-conditions, where (1) nodes and edges are labelled with distinct variables (and "unbounded" variables are used at most once across Boolean connectives); (2) assignment constraints are many-sorted formulae; and (2) the codomains of morphisms at each level of nesting contain only one more item (i.e. node or edge) than their domains. We then prove that all E-conditions can be replaced by equivalent E-conditions in normal form, and then use the assumptions allowed to us by the normal form to define a transformation to many-sorted formulae.

Definition 6.20 (Normal form for E-conditions)**.** Let c denote an E-condition. If $c = \texttt{true}$, then it is in *normal form*. If $c = \exists(a\colon P \hookrightarrow C \mid \gamma, c')$, then c is in normal form if all of the following hold:

1. The morphism a is an *inclusion*, i.e. for all nodes and edges x, $a(x) = x$;
2. The list components of all labels in P, C are single variables that are distinct from each other, and distinct from other variables (unless the node or edge is the same);
3. All variables in P, C, γ are sorted (i.e. belong to LVar, AVar, IVar, or SVar) and γ is a many-sorted formula (up to a renaming of `and` for $\wedge$, `or` for $\vee$, etc.);
4. The morphism a is either an identity morphism (i.e. $P = C$), or C is the graph P but with one additional item (either a node or edge, but not both); and
5. The E-condition c' is in normal form.

If E-conditions c, d are in normal form, then so is $\neg c$ and $c \wedge d$ (similarly for other Boolean formulae over E-conditions).

If an E-condition c can be rewritten as an equivalent E-condition $\bar{c}$ in normal form, we say that c has been *normalised* to $\bar{c}$. □

We show that arbitrary E-conditions can be rewritten as equivalent ones in normal form, addressing the five requirements in order. That morphisms can be replaced by inclusions is obvious, but the latter requirements are less so.

Fact 6.21 (Replacing morphisms with inclusions)**.** E-conditions containing morphisms that are not inclusions can be replaced by equivalent ones containing only inclusions (by a consistent renaming of identifiers). □

The requirement that the list components of all labels along the morphisms are just distinct variables is achieved by replacing lists with fresh variables, and equating these variables with the previous lists in the assignment constraints. This results in a convenient separation of information: the morphisms describe graph structure and whether items are marked or not, whereas the assignment constraints describe all the information about about list components. Hence the transformation to formulae can handle them separately.

Lemma 6.22 (Replacing lists)**.** Let c denote an E-condition. There is a transformation Relabel that yields an E-condition meeting the second requirement of Definition 6.20, such that for all injective graph morphisms s over $\mathcal{G}(\mathcal{L})$,

$$s \models c \text{ if and only if } s \models \text{Relabel}(c).$$

Construction. Consider c as a tree of morphisms equipped with assignment constraints and Boolean symbols (as in Section 4.1.1). First, consistently rename variables (to fresh variables) if they appear elsewhere in the tree but do not originate from the same parent. Then, every time a new node or edge is added along the tree, replace its list component l with a fresh variable `x` (consistently replacing the same node or edge label along the morphisms), and replace the assignment constraint γ at that level of the tree with `x` $= l$ `and` γ. Take the E-condition that results to be Relabel(c). □

Proof. See Appendix D.2. □

We illustrate this relabelling transformation on an example.

Example 6.23. Consider the E-condition:

$$c = \exists(\ \textcircled{x}_1 \xrightarrow{\texttt{x+x}} \textcircled{x}_2 \mid \texttt{int(x)}, \neg\exists(\ \textcircled{x}_1 \xrightarrow{\texttt{x+x}} \textcircled{x}_2 \circlearrowright^{\texttt{z}}\ )) \vee \exists(\ \textcircled{x}_1 \mid \texttt{string(x)}\ ).$$

We apply the transformation Relabel to c. First, since `x` appears twice in the tree (but does not originate in the root), we apply a substitution to one side of the disjunction to yield:

$$\bar{c} = \exists(\ \textcircled{x}_1 \xrightarrow{\texttt{x+x}} \textcircled{x}_2 \mid \texttt{int(x)}, \neg\exists(\ \textcircled{x}_1 \xrightarrow{\texttt{x+x}} \textcircled{x}_2 \circlearrowright^{\texttt{z}}\ )) \vee \exists(\ \textcircled{y}_1 \mid \texttt{string(y)}\ ).$$

Then, we consistently relabel the nodes and edges along the morphisms with fresh variables to yield:

$$\begin{aligned}\text{Relabel}(\bar{c}) \quad = \quad & \exists(\ \textcircled{a}_1 \xrightarrow{\texttt{b}} \textcircled{c}_2 \mid \texttt{a = x and b = x+x and c = x and int(x)}, \\ & \quad \neg\exists(\ \textcircled{a}_1 \xrightarrow{\texttt{b}} \textcircled{c}_2 \circlearrowright^{\texttt{d}} \mid \texttt{d = z})) \\ & \vee \exists(\ \textcircled{e}_1 \mid \texttt{e = y and string(y)}\ )\end{aligned}$$

We could simplify this E-condition (e.g. we could remove `d = z`), but we opt instead to show the full result of the transformation which makes no assumptions about how variables and labels might be used further down the tree. Note that this E-condition is not unique: there are infinitely more equivalent E-conditions (by a consistent renaming of variables). □

Now, we address the third requirement of normalised E-conditions. Recall the Venn diagram of variable classes in Figure 4.3. We replace variables with typed variables, i.e. variables belonging to:

$$\text{LVar} \cup \text{AVar} \cup \text{IVar} \cup \text{SVar}$$

and make assignment constraints many-sorted (up to the minor syntactic differences, e.g. `and` instead of $\wedge$). The idea is to exploit that, when variables are sorted, assignment constraints are a subset of the many-sorted formulae

(again, up to minor syntactic differences). Where assignment constraints however cannot be made to be correctly sorted, we replace the E-condition with `false`, since it will not be satisfiable by any assignment.

In the transformation from E-conditions to formulae, the assumption that assignment constraints are already sorted is a very convenient one: whereas variable types only concern us at the semantic level of E-conditions, in formulae types are already important at the syntactic level.

Lemma 6.24 (Sorting assignment constraints)**.** Let c be an E-condition adhering to the first two requirements of normal form (Definition 6.20). There is a transformation Sort which yields an E-condition meeting the third requirement, such that for all injective graph morphisms s over $\mathcal{G}(\mathcal{L})$,

$$s \models c \text{ if and only if } s \models \text{Sort}(c).$$

Construction. Consider c as a tree of morphisms equipped with assignment constraints and Boolean symbols (as in Section 4.1.1).

(1) Traverse down the tree and note for every variable $\mathtt{x}$ whether it appears in an expression *only* derivable from Integer (resp. String).

(2) For each variable $\mathtt{x}$ in the graphs and assignment constraints of each level, replace $\mathtt{x}$ with a fresh variable $\mathtt{x_I} \in$ IVar (resp. $\mathtt{x_S} \in$ SVar, $\mathtt{x_L} \in$ LVar) if $\mathtt{x}$ was noted only for Integer (resp. only for String, neither Integer nor String) in (1). If $\mathtt{x}$ was noted for both Integer and String in (1), then replace the E-condition at that level of the tree with `false`.

(3) Take the E-condition that results to be Sort(c).

□

Proof. See Appendix D.2. □

We demonstrate Sort on two E-conditions: one which is satisfiable, and one which is not.

Example 6.25. Consider the E-constraint:

$$\exists(\ \textcircled{x}_1\ \ \textcircled{y}_2 \mid \mathtt{x} = \mathtt{y}, \exists(\ \textcircled{x}_1\ \ \textcircled{y}_2 \mid \mathtt{x{+}5} = 4) \vee \exists(\ \textcircled{x}_1 \xrightarrow{\mathtt{z}} \textcircled{y}_2 \mid \mathtt{x} < \mathtt{z})).$$

Traversing the tree we find $\mathtt{x}, \mathtt{z}$ in expressions only derivable from Integer, and $\mathtt{y}$ in no expressions derivable only from Integer or String. Hence applying Sort to the E-condition gives us:

$$\exists(\textcircled{x_I}_1\ \textcircled{y_L}_2 \mid \mathtt{x_I} = \mathtt{y_L}, \exists(\textcircled{x_I}_1\ \textcircled{y_L}_2 \mid \mathtt{x_I{+}5} = 4) \vee \exists(\textcircled{x_I}_1 \xrightarrow{\mathtt{z_I}} \textcircled{y_L}_2 \mid \mathtt{x_I} < \mathtt{z_I}))$$

with $\mathtt{x_I}, \mathtt{z_I} \in$ IVar and $\mathtt{y_L} \in$ LVar. □

Example 6.26. Consider the E-constraint:

$$\exists(\emptyset \mid \texttt{atom(x)}, \exists(\emptyset \mid \texttt{x} < \texttt{y}) \vee \exists(\emptyset \mid \texttt{x.}\text{"}\texttt{ch}\text{"} = \texttt{w})) \wedge \exists(\emptyset \mid \texttt{atom(z)}).$$

Traversing the tree we find x in expressions only derivable from Integer and String, y from Integer, and w, z from neither Integer nor String. Hence applying Sort to the E-condition gives us:

$$\texttt{false} \wedge \exists(\emptyset \mid \texttt{atom}(\texttt{z}_\texttt{L}))$$

with $\texttt{z}_\texttt{L} \in$ LVar. (Note that the first disjunct was unsatisfiable because x cannot represent both an integer and string.) □

Next, we show that E-conditions can be decomposed into equivalent ones with morphisms that add at most one item (node or edge) to their codomains.

Lemma 6.27 (Decomposing morphisms)**.** Let c be an E-condition. There is a transformation Decomp which returns an E-condition meeting the fourth requirement of normal form (Definition 6.20), such that for all injective graph morphisms s over $\mathcal{G}(\mathcal{L})$,

$$s \models c \text{ if and only if } s \models \text{Decomp}(c).$$

Construction. If $c = \texttt{true}$, then $\text{Decomp}(c) = \texttt{true}$. If $c = \exists(a \mid \gamma, c')$, then

$$\text{Decomp}(c) = \exists(a_1, \exists(a_2, \cdots, \exists(a_{n-1}, \exists(a_n \mid \gamma, \text{Decomp}(c'))) \cdots)),$$

where $a_1, \ldots, a_n$ is a (finite) decomposition of a such that the codomain of each a_i is equal to its domain or contains one additional node or edge.

For Boolean formulae over E-conditions, Decomp is defined in the usual way, that is, $\text{Decomp}(\neg c) = \neg\text{Decomp}(c)$, $\text{Decomp}(c \wedge d) = \text{Decomp}(c) \wedge \text{Decomp}(d)$, and $\text{Decomp}(c \vee d) = \text{Decomp}(c) \vee \text{Decomp}(d)$.

□

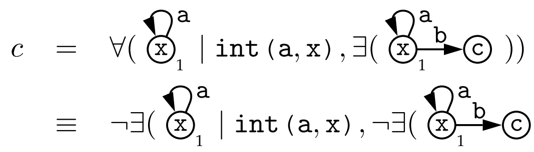
Proof. Follows from the fact that:

$$\text{Decomp}(c) \equiv \exists(a_n \circ a_{n-1} \circ \cdots \circ a_1 \mid \gamma, c').$$

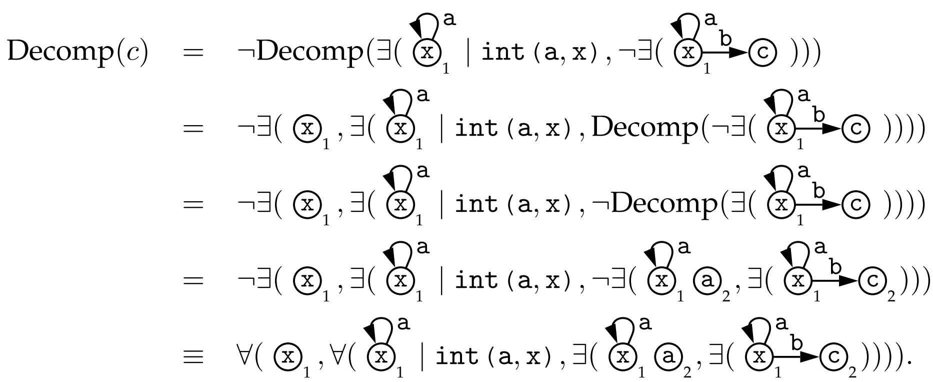
□

We demonstrate the transformation on a simple E-constraint. (Note that for a given E-condition c, $\text{Decomp}(c)$ is not unique, but it is well-defined.)

Example 6.28. Consider the E-condition:



Applying the transformation Decomp to c yields the following equivalent E-condition:



□

We have addressed the individual requirements of normal form. Finally, we state and prove that these steps can be combined to transform an arbitrary E-condition into one that is in normal form.

Proposition 6.29 (E-conditions can be normalised)**.** There is a transformation Norm, such that for all morphisms s over $\mathcal{G}(\mathcal{L})$, and all E-conditions c, Norm(c) is in normal form and:

$$s \models c \text{ if and only if } s \models \mathrm{Norm}(c).$$

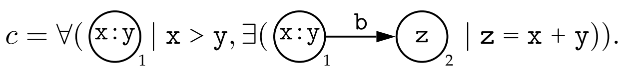
Construction. Define $\mathrm{Norm}(c) = \mathrm{Decomp}(\mathrm{Sort}(\mathrm{Relabel}(\bar{c})))$ where $\bar{c}$ is c but with all morphisms inclusions.

For Boolean formulae over E-conditions, Norm is defined in the usual way, that is, $\mathrm{Norm}(\neg c) = \neg\mathrm{Norm}(c)$, $\mathrm{Norm}(c \wedge d) = \mathrm{Norm}(c) \wedge \mathrm{Norm}(d)$, and $\mathrm{Norm}(c \vee d) = \mathrm{Norm}(c) \vee \mathrm{Norm}(d)$. □

Proof. Follows from the definitions of and lemmata for the intermediate transformations. □

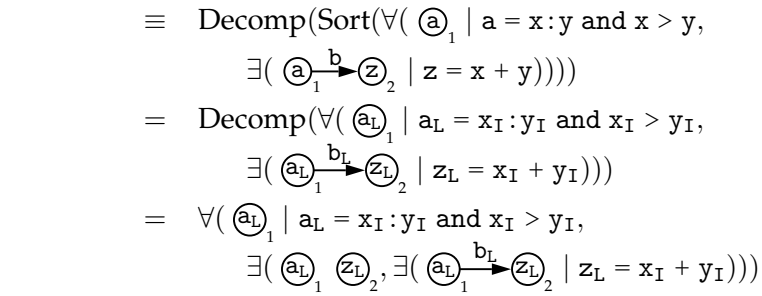
We now demonstrate all steps of the normalisation on an E-constraint. (Note that the result is not unique.)

Example 6.30. Consider the E-condition (assume that the morphisms are inclusions):



Applying the transformation Norm to c yields the following equivalent E-condition:

$$
\begin{aligned}
\mathrm{Norm}(c) &= \mathrm{Decomp}(\mathrm{Sort}(\mathrm{Relabel}(c))) \\
&\equiv \mathrm{Decomp}(\mathrm{Sort}(\forall(\ \textcircled{a}_1 \mid \mathtt{a = x:y \text{ and } x > y}, \\
&\qquad \exists(\ \textcircled{a}_1 \xrightarrow{\mathtt{b}} \textcircled{z}_2 \mid \mathtt{z = x + y}))))
\end{aligned}
$$



$$
\begin{aligned}
&= \mathrm{Decomp}(\forall(\ \textcircled{a_L}_1 \mid \mathtt{a_L = x_I:y_I \text{ and } x_I > y_I}, \\
&\qquad \exists(\ \textcircled{a_L}_1 \xrightarrow{\mathtt{b_L}} \textcircled{z_L}_2 \mid \mathtt{z_L = x_I + y_I})))
\end{aligned}
$$

$$
\begin{aligned}
&= \forall(\ \textcircled{a_L}_1 \mid \mathtt{a_L = x_I:y_I \text{ and } x_I > y_I}, \\
&\qquad \exists(\ \textcircled{a_L}_1\ \textcircled{z_L}_2, \exists(\ \textcircled{a_L}_1 \xrightarrow{\mathtt{b_L}} \textcircled{z_L}_2 \mid \mathtt{z_L = x_I + y_I})))
\end{aligned}
$$

with $\mathtt{a_L}, \mathtt{b_L}, \mathtt{z_L} \in \mathrm{LVar}$ and $\mathtt{x_I}, \mathtt{y_I} \in \mathrm{IVar}$.

□

6.4.2 Normalised E-Constraints to Formulae

Since we have shown that arbitrary E-constraints can be replaced by equivalent ones in the normal form of Definition 6.20, our transformation to many-sorted formulae exploits the characteristics that can be assumed of normalised E-constraints. Nodes and edges are quantified one at a time – once per level of nesting in the E-condition – and assignment constraints, which alone express relations between list components of labels, are already formulae (up to minor syntactic differences).

Proposition 6.31 (Normalised E-conditions to logic)**.** There is a transformation Form, such that for all normalised E-constraints c, and all graphs $G \in \mathcal{G}(\mathcal{L})$,

$$G \models c \text{ if and only if } G \models \mathrm{Form}(c).$$

Construction. For all E-constraints c, we define

$$\mathrm{Form}(c) = \mathrm{Form}'(c, \{\})$$

where the second parameter will carry variables that have already been quantified by the transformation. Let V in the following denote a set of sorted variables.

If $c = \mathtt{true}$, then $\mathrm{Form}'(c, V) = \mathtt{true}$. If $c = \exists(a \mid \gamma, c')$, then there are three possible outputs for $\mathrm{Form}'(c, V)$ defined for the three forms that a may take in normal form.

Suppose that $c = \exists(id\colon P \hookrightarrow P \mid \gamma, c')$, i.e. an E-condition with a morphism that is an identity. Then, $\mathrm{Form}'(c, V)$ is equal to:

$$\mathrm{Quant}(\mathrm{vars}(\gamma) - V)\,.\ \gamma^* \wedge \mathrm{Form}'(c', V \cup \mathrm{vars}(\gamma)).$$

Here and in later cases,

$$\mathrm{Quant}(\{\mathtt{x}, \mathtt{y}, \dots\}) = \exists \mathtt{x} : S_{\mathtt{x}}\,.\ \exists \mathtt{y} : S_{\mathtt{y}} \dots$$

where $S_{\mathtt{x}}$ is the symbol corresponding to the sort of $\mathtt{x}$ (i.e. $\mathtt{L}, \mathtt{I}, \mathtt{S}$). Moreover, γ^* denotes the assignment constraint γ with $\mathtt{and}, \mathtt{or}, \mathtt{not}$ replaced respectively by $\wedge, \vee, \neg$.

Suppose that $c = \exists([v\mathtt{a}] : P \hookrightarrow P' \mid \gamma, c')$, where $[v\mathtt{a}]$ denotes a morphism with codomain P' equal to domain P except for an additional unmarked node v labelled by variable $\mathtt{a}$. Then, $\mathrm{Form}'(c, V)$ is equal to:

$$\begin{aligned}&\exists \mathtt{v} : \mathtt{V}.\ \textstyle\bigwedge_{\mathtt{v}' \in V \cap \mathrm{VVar}} \neg \mathtt{v} = \mathtt{v}' \wedge \neg \mathtt{marked}\,(\mathtt{v}) \\ &\quad \wedge \mathrm{Quant}([\mathrm{vars}(\gamma) \cup \{\mathtt{a}\}] - V) \\ &\qquad .\ \mathtt{l}\,(\mathtt{v}) = \mathtt{a} \wedge \gamma^* \wedge \mathrm{Form}'(c', V \cup \mathrm{vars}(\gamma) \cup \{\mathtt{v}, \mathtt{a}\})\end{aligned}$$

The first line above demands the existence of an unmarked node that is distinct from others already quantified. The second line quantifies new variables in the assignment constraint, as well as the new variable $\mathtt{a}$ labelling v. The third line establishes that $\mathtt{a}$ comprises the list component of v, enforces the relations of the assignment constraint, and transforms the nested part of the E-condition.

Suppose that $c = \exists([euv\mathtt{a}] : P \hookrightarrow P' \mid \gamma, c')$, where $[euv\mathtt{a}]$ denotes a morphism with codomain P' equal to domain P except for an additional unmarked edge e, with source node u, target node v, list component variable $\mathtt{a}$. Then, $\mathrm{Form}'(c, V)$ is equal to:

$$\begin{aligned}&\exists \mathtt{e} : \mathtt{E}.\ \textstyle\bigwedge_{\mathtt{e}' \in V \cap \mathrm{EVar}} \neg \mathtt{e} = \mathtt{e}' \wedge \neg \mathtt{marked}\,(\mathtt{e}) \\ &\quad \wedge \mathrm{Quant}([\mathrm{vars}(\gamma) \cup \{\mathtt{a}\}] - V) \\ &\qquad .\ \mathtt{s}\,(\mathtt{e}) = \mathtt{u} \wedge \mathtt{t}\,(\mathtt{e}) = \mathtt{v} \wedge \mathtt{m}\,(\mathtt{e}) = \mathtt{a} \\ &\qquad\quad \wedge \gamma^* \wedge \mathrm{Form}'(c', V \cup \mathrm{vars}(\gamma) \cup \{\mathtt{e}, \mathtt{a}\})\end{aligned}$$

Note that this exploits the correspondence between node identifiers and node variables established in the previous case.

For cases when the morphism introduces a marked node or edge, then $\mathrm{Form}'(c, V)$ is as above but without negating the $\mathtt{marked}$ predicate.

For Boolean formulae over E-conditions, Form′ is defined in the usual way, that is, $\mathrm{Form}'(\neg c, V) = \neg\mathrm{Form}'(c, V)$, $\mathrm{Form}'(c \wedge d, V) = \mathrm{Form}'(c, V) \wedge \mathrm{Form}'(d, V)$, and $\mathrm{Form}'(c \vee d, V) = \mathrm{Form}'(c, V) \vee \mathrm{Form}'(d, V)$.

□

Example 6.32. We take the normalised E-condition from Example 6.30:

$$\begin{aligned}\mathtt{c} \quad &= \quad \forall(\ \textcircled{\mathtt{a}_\mathtt{L}}_1 \mid \mathtt{a}_\mathtt{L} = \mathtt{x}_\mathtt{I} : \mathtt{y}_\mathtt{I}\ \mathtt{and}\ \mathtt{x}_\mathtt{I} > \mathtt{y}_\mathtt{I}, \\ &\qquad\qquad \exists(\ \textcircled{\mathtt{a}_\mathtt{L}}_1\ \textcircled{\mathtt{z}_\mathtt{L}}_2, \exists(\ \textcircled{\mathtt{a}_\mathtt{L}}_1 \xrightarrow{\mathtt{b}_\mathtt{L}} \textcircled{\mathtt{z}_\mathtt{L}}_2 \mid \mathtt{z}_\mathtt{L} = \mathtt{x}_\mathtt{I} + \mathtt{y}_\mathtt{I}))) \\ &\equiv \quad \neg\exists(\ \textcircled{\mathtt{a}_\mathtt{L}}_1 \mid \mathtt{a}_\mathtt{L} = \mathtt{x}_\mathtt{I} : \mathtt{y}_\mathtt{I}\ \mathtt{and}\ \mathtt{x}_\mathtt{I} > \mathtt{y}_\mathtt{I}, \\ &\qquad\qquad \neg\exists(\ \textcircled{\mathtt{a}_\mathtt{L}}_1\ \textcircled{\mathtt{z}_\mathtt{L}}_2, \exists(\ \textcircled{\mathtt{a}_\mathtt{L}}_1 \xrightarrow{\mathtt{b}_\mathtt{L}} \textcircled{\mathtt{z}_\mathtt{L}}_2 \mid \mathtt{z}_\mathtt{L} = \mathtt{x}_\mathtt{I} + \mathtt{y}_\mathtt{I}))).\end{aligned}$$

Define $c'' = \exists(\ (\mathtt{a_L})_1 \xrightarrow{\mathtt{b_L}} (\mathtt{z_L})_2 \mid \mathtt{z_L} = \mathtt{x_I} + \mathtt{y_I})$ and $c' = \neg\exists(\ (\mathtt{a_L})_1\ (\mathtt{z_L})_2, c'')$. Applying the transformation Form to c results in the following sentence:

$$
\begin{aligned}
&\text{Form}(c)\\
&= \quad \text{Form}'(c, \{\})\\
&= \quad \neg\exists \mathtt{v} : \mathtt{V}.\ \neg \mathtt{marked}(\mathtt{v})\\
&\qquad \wedge\ \text{Quant}(\{\mathtt{a_L}, \mathtt{x_I}, \mathtt{y_I}\})\\
&\qquad\quad .\ \mathtt{l}(\mathtt{v}) = \mathtt{a_L} \wedge \mathtt{a_L} = \mathtt{x_I} : \mathtt{y_I} \ \wedge\ \mathtt{x_I} > \mathtt{y_I}\\
&\qquad\qquad \wedge\ \text{Form}'(c', \{\mathtt{v}, \mathtt{a_L}, \mathtt{x_I}, \mathtt{y_I}\})\\
&= \quad \neg\exists \mathtt{v} : \mathtt{V}.\ \neg \mathtt{marked}(\mathtt{v})\\
&\qquad \wedge\ \exists \mathtt{a_L} : \mathtt{L},\ \mathtt{x_I} : \mathtt{I},\ \mathtt{y_I} : \mathtt{I}\\
&\qquad\quad .\ \mathtt{l}(\mathtt{v}) = \mathtt{a_L} \wedge \mathtt{a_L} = \mathtt{x_I} : \mathtt{y_I} \ \wedge\ \mathtt{x_I} > \mathtt{y_I}\\
&\qquad\qquad \wedge\ \text{Form}'(c', \{\mathtt{v}, \mathtt{a_L}, \mathtt{x_I}, \mathtt{y_I}\})\\
&= \quad \neg\exists \mathtt{v} : \mathtt{V}.\ \neg \mathtt{marked}(\mathtt{v})\\
&\qquad \wedge\ \exists \mathtt{a_L} : \mathtt{L},\ \mathtt{x_I} : \mathtt{I},\ \mathtt{y_I} : \mathtt{I}.\ \mathtt{l}(\mathtt{v}) = \mathtt{a_L} \wedge \mathtt{a_L} = \mathtt{x_I} : \mathtt{y_I} \ \wedge\ \mathtt{x_I} > \mathtt{y_I}\\
&\qquad\quad \wedge\ \neg\exists \mathtt{w} : \mathtt{V}.\ \neg \mathtt{w} = \mathtt{v} \wedge \neg \mathtt{marked}(\mathtt{w}) \wedge \exists \mathtt{z_L} : \mathtt{L}.\ \mathtt{l}(\mathtt{w}) = \mathtt{z_L}\\
&\qquad\qquad \wedge\ \text{Form}'(c'', \{\mathtt{v}, \mathtt{a_L}, \mathtt{x_I}, \mathtt{y_I}, \mathtt{z_L}\})\\
&= \quad \neg\exists \mathtt{v} : \mathtt{V}.\ \neg \mathtt{marked}(\mathtt{v})\\
&\qquad \wedge\ \exists \mathtt{a_L} : \mathtt{L},\ \mathtt{x_I} : \mathtt{I},\ \mathtt{y_I} : \mathtt{I}.\ \mathtt{l}(\mathtt{v}) = \mathtt{a_L} \wedge \mathtt{a_L} = \mathtt{x_I} : \mathtt{y_I} \ \wedge\ \mathtt{x_I} > \mathtt{y_I}\\
&\qquad\quad \wedge\ \neg\exists \mathtt{w} : \mathtt{V}.\ \neg \mathtt{w} = \mathtt{v} \wedge \neg \mathtt{marked}(\mathtt{w}) \wedge \exists \mathtt{z_L} : \mathtt{L}.\ \mathtt{l}(\mathtt{w}) = \mathtt{z_L}\\
&\qquad\qquad \wedge\ \exists \mathtt{e} : \mathtt{E}.\ \neg \mathtt{marked}(\mathtt{e}) \wedge \exists \mathtt{b_L} : \mathtt{L}.\ \mathtt{s}(\mathtt{e}) = \mathtt{v} \wedge \mathtt{t}(\mathtt{e}) = \mathtt{w}\\
&\qquad\qquad\quad \wedge\ \mathtt{m}(\mathtt{e}) = \mathtt{b_L} \wedge \mathtt{z_L} = \mathtt{x_I} + \mathtt{y_I}
\end{aligned}
$$

The resulting sentence can be simplified further using standard manipulation laws for predicate logics.

$$
\begin{aligned}
&\equiv \quad \forall \mathtt{v} : \mathtt{V},\ \mathtt{x_I} : \mathtt{I},\ \mathtt{y_I} : \mathtt{I}.\ (\neg \mathtt{marked}(\mathtt{v}) \wedge \mathtt{l}(\mathtt{v}) = \mathtt{x_I} : \mathtt{y_I} \wedge \mathtt{x_I} > \mathtt{y_I})\\
&\qquad \Rightarrow\ \exists \mathtt{w} : \mathtt{V},\ \mathtt{e} : \mathtt{E}.\ \neg \mathtt{w} = \mathtt{v} \wedge \mathtt{l}(\mathtt{w}) = \mathtt{x_I} + \mathtt{y_I}\\
&\qquad\quad \wedge\ \mathtt{s}(\mathtt{e}) = \mathtt{v} \wedge \mathtt{t}(\mathtt{e}) = \mathtt{w}
\end{aligned}
$$

□

To prove the statement of Proposition 6.31, we first state and prove a more general lemma for E-conditions (not only E-constraints) and Form′ for which we can apply structural induction. Here, we consider satisfaction by morphisms and interpretation functions, rather than only graphs. We show that Proposition 6.31 is simply an instance of the more general lemma.

Lemma 6.33 (E-conditions can be expressed as formulae)**.** For every E-condition c in normal form, and all injective graph morphisms $p\colon P^\alpha \hookrightarrow G$ with $P \in \mathcal{G}(\mathrm{EC}), G \in \mathcal{G}(\mathcal{L})$, and α a well-typed assignment, we have that:

$$p\colon P^\alpha \hookrightarrow G \models c^{\sigma_\alpha} \text{ if and only if } I_G^{p,\alpha} \models \mathrm{Form}'(c, \mathrm{dom}(\alpha) \cup P^*)$$

where P^* is the set of node and edge variables corresponding to the identifiers in P. Additionally, $I_G^{p,\alpha}$ is defined as I_G but with the following mappings: (1) for each variable $\mathtt{x}$ in $\mathrm{dom}(\alpha)$, $I_G^{p,\alpha}(\mathtt{x}) = \alpha(\mathtt{x})$; (2) for each node v in P, $I_G^{p,\alpha}(\mathtt{v}) = p(v)$; and (3) for each edge e in P, $I_G^{p,\alpha}(\mathtt{e}) = p(e)$. □

Proof. See Appendix D.3. □

We now prove the correctness of Proposition 6.31 by showing that it is simply an instance of Lemma 6.33.

Proof of Proposition 6.31. To prove the correctness of the proposition, we show that it is an instance of Lemma 6.33. Define $i\colon \emptyset \hookrightarrow G$ and α_ϵ the assignment undefined on all variables, i.e. $\mathrm{dom}(\alpha_\epsilon) = \emptyset$. Then, by the definition of $\models$, the observation that $I_G = I_G^{i,\alpha_\epsilon}$, and Lemma 6.33, we have that:

$$\begin{aligned} G \models \varphi \text{ iff } & i\colon \emptyset \hookrightarrow G \models c \\ \text{iff } & i\colon \emptyset^{\alpha_\epsilon} \hookrightarrow G \models c^{\sigma_{\alpha_\epsilon}} \\ \text{iff } & I_G^{i,\alpha_\epsilon} \models \mathrm{Form}'(c, \mathrm{dom}(\alpha_\epsilon) \cup \emptyset) \\ \text{iff } & I_G \models \mathrm{Form}'(c, \{\}) \\ \text{iff } & I_G \models \mathrm{Form}(c) \\ \text{iff } & G \models \mathrm{Form}(c). \end{aligned}$$

□

Finally, we state the main result of this section: that any E-constraint (whether normalised or not) can be transformed into a semantically equivalent many-sorted sentence.

Theorem 6.34 (E-constraints can be expressed as sentences)**.** Given any E-constraint c, there is a transformation from c to a sentence φ such that for any graph $G \in \mathcal{G}(\mathcal{L})$,

$$G \models c \text{ if and only if } G \models \varphi.$$

Construction. Take $\varphi = \mathrm{Form}(\mathrm{Norm}(c))$. □

Proof. By Propositions 6.29 and 6.31. □

6.5 Verification with Many-Sorted Formulae

This chapter has contributed a characterisation of E-constraints in terms of a classical logic, which may be more understandable for programmers unfamiliar with graph transformation, and may be be more practical for theorem proving technology. Much of the effort has been in showing their expressive equivalence, but in doing so, we have obtained another result for "free": that is, additional sound Hoare calculi for proving the correctness of graph programs.

The first presentation of the Hoare calculi (Chapter 3) was given in an extensional style (i.e. without a specified assertion language). Since there are instances of the calculi for E-constraints (in particular, there are transformations defining Wlp, SE, FE), and since there are transformations from many-sorted formulae to E-conditions and back, we can define Wlp, SE, and FE for many-sorted formulae simply by transforming them to E-conditions first.

Definition 6.35 (Many-sorted formulae as assertions)**.** Let $\mathfrak{M}$ denote the assertion language $\langle M, \models \rangle$, where $M = \{\varphi \in \text{Formula} \mid \varphi \text{ is a sentence}\}$ and $\models \subseteq \mathcal{G}(\mathcal{L}) \times M$ is the satisfaction relation for sentences. □

Definition 6.36 (Hoare calculi with many-sorted sentences)**.** Let r (resp. $\mathcal{R}$) range over conditional rule schemata (resp. over sets of conditional rule schemata), c, c', d, d', e, inv over assertions in $\mathfrak{M}$, and P, Q over graph programs. Our Hoare calculi with assertions in $\mathfrak{M}$ for partial, weak total, and total correctness comprise the proof rules of Appendix B but with:

1. $\text{Form}(\text{Norm}(\text{Pre}(r, \text{Cond}(c))))$ replacing $\text{Pre}(r, c)$; and
2. $\text{Form}(\text{Norm}(\text{App}(\mathcal{R})))$ replacing $\text{App}(\mathcal{R})$.

If a triple $\{c\}\ P\ \{d\}$ can be instantiated from any of the partial correctness axioms, or derived from a combination of axioms and inference rules, we denote this by $\vdash_{\text{par}} \{c\}\ P\ \{d\}$. (Analogous for wtot, tot.) □

Note that the calculi of Definition 6.36 inherit the restrictions of the calculi with E-constraints, that is, the restriction to programs with sets of (conditional) rule schemata in the guards of conditional constructs (as opposed to arbitrary programs).

Soundness of the calculi is a direct result of the soundness of the calculi with E-constraints, and the correctness of the transformations between the two assertion languages.

Theorem 6.37 (Soundness)**.** Let s denote par, wtot, or tot. Given a program P and assertions c, d from $\mathfrak{M}$,

$$\vdash_s \{c\}\ P\ \{d\} \text{ implies } \models_s \{c\}\ P\ \{d\}.$$

□

Proof. Follows from the soundness Theorems 3.22, 3.23, 3.24, the results for Cond, Norm, and Form (Theorem 6.15, Propositions 6.29, 6.31), and the results that Pre and App define the special assertions of the extensional calculi (Theorems 4.46, 4.47, 4.51). □

These calculi for many-sorted formulae might be more well-suited as a "front end" in some implementation of our work in a proof assistant, with the generation of Wlp, SE, and FE occurring behind the scenes via E-conditions and pushout constructions. However, for non-trivial assertions, it is likely that the formulae resulting from this process are likely to be quite large as a result of the translations and normalisations. Hence, there are two avenues of further work that might help:

1. finding direct transformations for Wlp, SE, and FE without first translating to E-conditions; and

2. devising and implementing inference rules for automatically simplifying formulae.

6.6 Related Work

Rensink [Ren04] showed how first-order logic could be represented as a recursively nested set of graph morphisms – an idea which became central to nested conditions, and hence also E-conditions. Rensink's paper defined transformations back and forth between these two representations.

Habel and Pennemann explored the expressiveness of graph conditions in [HP09], and in particular, gave translations between graph conditions and a logic very similar to the first-order graph formulae studied by Courcelle ([Cou90, Cou97, CE12]), showing that the formalisms are expressively equivalent. Unlike our approach, they quantify over variables that can be interpreted as either nodes or edges, and because they consider only finite label alphabets, they introduce a unary predicate for every symbol of the alphabet (e.g. $\mathrm{lab}_b(d)$ is interpreted as true over a graph if node or edge d is labelled with b). Because variables are untyped, they require an axiomatisation for automated theorem provers with statements such as "source and target nodes cannot be edges" (such formulae cannot be written in our many-sorted logic).

Habel and Radke have been studying the relationship of Courcelle's logics with a formalism similar to nested conditions but extended with hyperedge replacement grammars, which allows for the expression of non-local properties (e.g. paths, connectedness). In [HR10] they give a translation from monadic second-order (MSO) logic to their formalism, and show that there cannot be a another translation back (there are certain counting MSO properties that it can express). In [Rad13], Radke shows that the

formalism can be translated to full second-order logic over graphs, and though he also gives a translation in the other direction from node-counting MSO formulae, the question of whether there is a translation from second-order formulae or not remains open.

6.7 Summary

With this chapter we have:

- defined[3] a many-sorted first-order logic for the graphs of GP;
- shown its expressive equivalence with E-constraints, by defining and proving correct translations between them;
- shown how sentences of the logic can be "plugged in" to our extensional calculi, providing an alternative assertion language for verification that does not rely on abstract notions like morphisms; and
- pointed to some related work on graph logics and their connections to formalisms based on graph morphisms.

[3]The abstract syntax of the logic was contributed by Detlef Plump.

Chapter 7

Conclusions and Future Work

In this chapter we draw the thesis to a close, first with some conclusions, and then with some suggestions for interesting future work.

7.1 Conclusions

In Section 1.2 we proposed a research hypothesis:

> *The graph transformation language GP 2 can effectively be equipped with a reasoning system based on Hoare logic, facilitating proofs about both structural properties of graphs and relations between their labels.*

and some criteria of such a reasoning system that would support it:

- *sound*: every specification one can prove in the system must be valid with respect to the semantics of the language;
- *realistic*: in that it does not require impractical assumptions or restrictions on programs;
- *practical*: it can effectively reason about interesting partial correctness and termination properties of structure- and data-manipulating graph programs;
- *general*: it should not be fixed to particular assertion languages, and hence should be easy to extend.

We believe that the contributions of this thesis do satisfy these criteria, and hence support the hypothesis. In particular: we presented Hoare calculi for verifying partial correctness and termination properties of graph programs, and did so in an extensional style to allow their re-use with any assertion language satisfying certain conditions (*general*). We proved these calculi to be *sound*, relatively complete, and complete for termination. We explored

the relationship between what assertion languages could define and the decidability of the model checking problem, justifying a minor but *realistic* restriction of graph programs for verification. We presented E-conditions, a graph specification formalism allowing for the expression of properties about both structure and data. We instantiated our extensional calculi with E-conditions as the assertions, and demonstrated its *practical* use in reasoning about properties of a number of data-manipulating programs that up until now were only verifiable in an ad hoc way. The constructions and transformations presented are all effective (algorithmic) and could be implemented in a straightforward way. We also studied a classical many-sorted logic over graphs, defining effective translations to E-conditions and back, providing another front-end to the reasoning system that could be integrated with proof assistants (*general*).

Of course, in considering these criteria, we are not claiming to have equipped GP 2 with the *most* effective reasoning system based on Hoare logic. Indeed, much work remains to be done in addressing the deficiencies, amongst which the lack of non-local properties in assertions, and the lack of means to relate input and output graphs, stand out as two of the most pressing – we go into more detail in the following section. But, we hope that this thesis represents a reasonable step in the right direction, and has gone some way to widening the class of problems that we can model and reason about as graph programs.

7.2 Future Work

Here, we discuss some potential avenues of interesting future research, including strongest postconditions, relative completeness, augmenting E-constraints with paths, tracking the nodes and edges of graphs from the input to the output state, applying separation logic, formalising our work in a proof assistant, and larger case studies.

7.2.1 Strongest Postconditions

We defined the core axiom [ruleapp] of our calculi to require the construction of a weakest precondition of a (conditional) rule schema with respect to a postcondition. This followed from the "backwards reasoning" approach of Hoare [Hoa69] (and several successive authors), in which the axiom of assignment takes a postcondition and simply applies a substitution to obtain the precondition. The original semantics of assignment, due to Floyd [Flo67]), instead went in the other direction: from an assignment and a precondition, one obtains a postcondition. For assignments however, the forward semantics appears more complicated than the simple backwards, substitution semantics, because it requires an existential quan-

tification. Hence, the backwards semantics of assignment is more typically known and implemented in provers.

Gordon and Collavizza [GC10] compare and contrast the two semantics of assignment, and more generally, the uses of weakest preconditions and strongest postconditions in verification. They argue that whereas the backwards semantics of weakest preconditions appears simpler, there is a case for using the forward semantics of strongest postconditions, e.g. in discovering potential postconditions of legacy code, and in unifying deductive methods for proving correctness with automatic property checking based on symbolic execution. Hence, it is natural to wonder whether a "forwards version" of [ruleapp] would prove useful in the context of verifying graph programs.

Actually, for ordinary rules (i.e. not rule schemata), Habel and Pennemann [HP09] have shown that computing the strongest postcondition of a rule with respect to a precondition is simple, and can utilise the same transformations for computing the weakest precondition. This relies completely on the fact that rules in the DPO approach can be reversed, i.e. if $r : \langle L \Rightarrow R \rangle$ is a rule, then so is $r^{-1} : \langle R \Rightarrow L \rangle$, and if $G \Rightarrow_r H$, then there is always some direct derivation $H \Rightarrow_{r^{-1}} M$ such that $M \cong G$. Conditional rule schemata, labelled over expressions, clearly cannot be reversed in this way (in general). Hence, it is not currently clear how one would define a forwards version of [ruleapp] for graph programs, and hence we leave it as future work.

Apart from the benefits to our verification calculi, being able to reverse (conditional) rule schemata in general is likely to be important for the static analysis of rule schemata, and more generally, graph programs. To give an example (without a full explanation), the existence of a pair of parallel independent derivations via ordinary DPO rules implies the existence of sequentially independent derivations in which one of those rules is reversed. Studying such properties of derivations involving rule schemata might make interesting future work, especially within the context of checking confluence.

7.2.2 Relative Completeness

In Section 4.5 we discussed the completeness problem for our calculi with E-constraints, noting that while it is clear the assertion language is not expressive, it remains open as to whether our calculi with E-constraints are relatively complete or not. Investigating this problem would hence be interesting future work, as would exploring whether there is a minimum extension to E-constraints that would make the assertion language expressive (whilst maintaining the definability of Wlp, SE, and FE).

7.2.3 Expressing the Existence of Paths

In Chapter 5 we demonstrated the use of E-constraints in verifying a number of graph programs, but hit their limits when we came to consider programs in which global properties of the graph are more important. In `connected?` for example, the most interesting specifications require a way to reason about the connectedness of a graph, i.e. getting the right outcome according to whether or not the input graph is connected. Connectedness is defined in terms of the existence of arbitrary-length paths between all pairs of nodes, which is a property that E-constraints cannot express. This is certainly a weakness, because several properties of graphs are indeed defined in terms of paths (see e.g. [Har69]). Indeed, paths can be thought of a transitive closure property, and several other domains in computer science (e.g. relational databases) have shown the importance of supporting them.

We sketched already an informal proof for `connected?`, using an envisaged extension of E-constraints for arbitrary-length paths. The example suggests that it would be worthwhile to further study such a formalism, in particular, extending the transformation Pre as a basis for practical proofs. While this conservative extension of E-constraints will lack much of the expressiveness of more powerful formalisms like hyperedge replacement conditions [HR10], it should be easier to define and prove correct an extension to Pre for paths, and we feel that paths alone will allow for the verification of many interesting non-local properties.

7.2.4 Tracking the Graph

In Chapter 5 we proved a number properties about graph programs, guaranteeing to programmers that if the input graph they provided had property X, then any output graph(s) would have property Y. We were able to prove, for example, that the program `colouring` resulted in *a correctly coloured graph*. What we were not able to prove however, is that the correctly coloured graph had any relation at all to the provided input graph. To take an extreme example, the empty graph $\emptyset$ is certainly a correctly coloured graph – but unless the input graph is also the empty graph, then it is certainly not an output of `colouring` that we want to see!

The empty graph is an extreme example, because it would be very clear from that proof tree if it is always going to be the output: the "correctly coloured" postcondition would likely be implied, in some application of [cons], from a postcondition that states that the graph is empty. It does however illustrate our point that there is a disconnect between the input state and the output state. In classical program logics we can relate Hoare triples using, for example, ghost variables. But in the context of GP, where programs operate on graphs, we need a different way of relating triples. (One possibility might be to look at the so-called "program conditions" of

[AH08], which can express some useful properties about the relationship between input and output graphs, such as "all nodes and edges are preserved".)

For the `colouring` program we want to be able to additionally assert that any output graph – ignoring node labels – will always be the same as the input graph. For other programs in which label manipulation plays a key role, it may also be helpful to express that all of, some of, or none of the output graph is preserved from the input graph. Moreover, we could write stronger termination functions, e.g. node X is labelled with an integer that decreases by one after each execution. Generally speaking, we want to be able to track changes to the graph as rule schemata are applied to it. An extension of our calculi to allow such reasoning would make interesting further work.

7.2.5 Separation Logic and the Frame Rule

Separation logic [ORY01, Rey02] is an extension of Hoare logic allowing for compositional reasoning about heap-manipulating programs. It supports the idea of *local reasoning*, i.e. specifying and verifying program components with respect to the portion of the heap they use – as opposed to the global state – allowing for the approach to scale more easily. Separation logic has been a "hot topic" of research for more than a decade, and has been the foundation of impressive advances in both the theory and practice of program verification.

A possible topic for future work might be to investigate whether ideas or principles from separation logic might lead to useful extensions in our verification calculi for graph programs. To take a simple example: suppose some part of a graph program operates exclusively on unmarked nodes, and the marked nodes are not altered in any way. Can we use the separating conjunction to describe completely and separately the two parts of the graph, and then use the frame rule to focus on proving a triple about specifically the unmarked part of the graph?

There are several issues to consider that might make such an approach difficult – or perhaps not even useful for graph program verification. Rule schema application might in fact be simpler to reason about in the context of the global state: its application is nondeterministic, there could be several matches, and it might not be so easy to carve the graph into distinct parts (except for simple cases such as the one described in the previous paragraph). This is perhaps reflected in the Pre construction for E-constraints, in which we quantify over *every* possible match of a rule schema. Moreover, graph programs can increase the size of the state – simply by adding nodes and edges – for as long as they like. Hence a program could be said to operate on some distinct subgraph of the graph, but might then add nodes and edges which no longer belong to that part of the graph

(as described by the separating conjunction). Moreover still, consideration would need to be given as to how a graph can be split into two disjoint parts. For example, can edges be left dangling?

It might be interesting to investigate separation logic further in the context of graph transformation (related work has already explored a relationship between separation logic and hyperedge-replacement systems [DP09]). We suspect that it will likely not work in the context of our verification calculi, but it would be nice to have a better understanding as to why. A key issue seems to be that we cannot assume a fixed size of program state, because graph programs can add nodes or edges without end.

7.2.6 Formalisation in Proof Assistants

Automatically checking whether implications $c \Rightarrow d$ of E-constraints are valid could be tackled in at least two ways. One approach would be to use our translation from E-constraints to many-sorted first-order formulae, and pass the implication to a first-order theorem prover. Pennemann explored this approach for (untyped) first-order formulae equivalent to nested conditions, but found that several valid implications could not be proven because he needed to restrict the provers to graphs via axioms, and these axioms became part of the problem to solve. Hence in [Pen08] he proposed a calculus of deductive rules operating directly on nested conditions themselves. Extending his calculus for E-constraints would make interesting future work, and an implementation would help to make the verification calculi more practical (almost every proof tree requires the checking of implications).

A challenging task for future work would be the formalisation of our verification calculi in some interactive proof assistant, e.g. Isabelle [NPW02] or Coq [BC04]. One reason to do this would be to check the soundness of our calculi, including the theorems and lemmata about the transformations Pre and App. (We believe they are correct – and hope that you think so too – but the agreement of a theorem prover adds that extra assurance!) Another reason is to make our theoretical work practical. Constructing proofs by hand can be tedious and error prone, and in the context of E-constraints, applying Pre to postconditions with several levels of nesting is difficult to do by hand, sometimes resulting in large disjunctions of E-conditions. An interactive proof system like Isabelle requires human intervention and creativity to guide the search for proofs, but can often assist with automation (e.g. a higher-order version of resolution, a term rewriting engine, and a semantic tableaux prover). Thus it should be an appropriate system to formalise our work in, with the user providing input for tasks difficult to fully automate (e.g. finding loop invariants), and the proof assistant attempting to automatically discharge others (e.g. applying transformations App and Pre, discharging implications arising from [cons]).

Implementing our calculi in such a proof assistant would require a lot of work. Before even considering the semantics of graph programs and the soundness of our proof rules, we would need to formalise graph transformation itself (in particular the DPO approach with relabelling), which in turn requires the formalisation of e.g. graphs, graph morphisms, and pushouts. How to best complete these tasks is not obvious, and existing work on formalising graph transformation (see e.g. [Str08]) does not address the DPO approach. A complete formalisation of these foundational concepts would not just be useful for the verification of GP, but for other researchers in graph transformation who would seek the use of a proof assistant in their work.

7.2.7 Larger Case Studies

In Chapter 5 we demonstrated our verification calculi on a number of programs, going beyond previous work in graph transformation verification by statically verifying properties that related data in labels with structural properties of graphs. However, the programs we demonstrated our work on were not large ones, and moreover, were limited to computing (or checking) graph properties.

An interesting task for future work would be to apply our techniques to verification problems that could be *abstracted* to graphs and graph programs. One idea would be to apply the work to verifying abstract representations of pointer operations (already graph transformation can naturally specify classes of pointer structures, e.g. [BPR04b]). Another idea would be to model and verify garbage collection algorithms as graph programs.

While our work allows us to go beyond the case studies of [HP09] and verify programs in which data manipulation plays a role, a problem is the limitation of our E-constraint calculi to verifying only local properties. In the two future case studies we suggested, there will be several non-local specifications about the programs that would be desirable to verify. Hence, such case studies are likely to be more successful after first extending E-constraints and their associated transformations with at least paths (see Section 7.2.3), a fundamental non-local property which would be certain to significantly broaden what we can verify about graph programs.

Appendix A

Extensional Proof Rules

$$[\text{ruleapp}]_{\text{wlp}}\ \frac{}{\{\text{Wlp}_{\mathfrak{A}}[r,c]\}\ r\ \{c\}}$$

$$[\text{ruleset}]\ \frac{\{c\}\ r\ \{d\}\ \text{for each}\ r \in \mathcal{R}}{\{c\}\ \mathcal{R}\ \{d\}}$$

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

$$[\text{if}]\ \frac{\{\text{SE}_{\mathfrak{A}}[c,C]\}\ P\ \{d\} \quad \{\text{FE}_{\mathfrak{A}}[c,C]\}\ Q\ \{d\}}{\{c\}\ \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}}$$

$$[\text{try}]\ \frac{\{\text{SE}_{\mathfrak{A}}[c,C]\}\ C; P\ \{d\} \quad \{\text{FE}_{\mathfrak{A}}[c,C]\}\ Q\ \{d\}}{\{c\}\ \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}}$$

$$[!]\ \frac{\{inv\}\ P\ \{inv\}}{\{inv\}\ P!\ \{\text{FE}_{\mathfrak{A}}[inv,P]\}}$$

$$[\text{cons}]\ \frac{\text{impl}(c,c') \quad \{c'\}\ P\ \{d'\} \quad \text{impl}(d',d)}{\{c\}\ P\ \{d\}}$$

$\text{impl}(a_1,a_2)$: prove that $G \models_{\mathfrak{A}} a_1$ implies $G \models_{\mathfrak{A}} a_2$ for all $G \in \mathcal{G}(\mathcal{L})$

Figure A.1: Partial correctness proof rules for core commands

$$[\text{skip}]\ \frac{}{\{c\}\ \texttt{skip}\ \{c\}}$$

$$[\text{fail}]\ \frac{}{\{c\}\ \texttt{fail}\ \{d\}}$$

$$[\text{or}]\ \frac{\{c\}\ P\ \{d\} \quad \{c\}\ Q\ \{d\}}{\{c\}\ P\ \texttt{or}\ Q\ \{d\}}$$

$$[\text{if}_2]\ \frac{\{\text{SE}_{\mathfrak{A}}[c, C]\}\ P\ \{d\} \quad \text{impl}(\text{FE}_{\mathfrak{A}}[c, C], d)}{\{c\}\ \texttt{if}\ C\ \texttt{then}\ P\ \{d\}}$$

$$[\text{try}_2]\ \frac{\{\text{SE}_{\mathfrak{A}}[c, C]\}\ C; P\ \{d\} \quad \text{impl}(\text{FE}_{\mathfrak{A}}[c, C], d)}{\{c\}\ \texttt{try}\ C\ \texttt{then}\ P\ \{d\}}$$

$$[\text{try}_3]\ \frac{\{\text{SE}_{\mathfrak{A}}[c, C]\}\ C\ \{d\} \quad \text{impl}(\text{FE}_{\mathfrak{A}}[c, C], d)}{\{c\}\ \texttt{try}\ C\ \{d\}}$$

$\text{impl}(a_1, a_2)$: prove that $G \models_{\mathfrak{A}} a_1$ implies $G \models_{\mathfrak{A}} a_2$ for all $G \in \mathcal{G}(\mathcal{L})$

Figure A.2: Partial correctness proof rules for derived commands

$[\text{ruleapp}]_{\text{wlp}}$, [ruleset], [comp], [cons] of Figure A.1, and:

$$[\text{if}]_{\text{wtot}}\ \frac{\{c\}\ C\ \{\top_{\mathfrak{A}}\} \quad \{\text{SE}_{\mathfrak{A}}[c, C]\}\ P\ \{d\} \quad \{\text{FE}_{\mathfrak{A}}[c, C]\}\ Q\ \{d\}}{\{c\}\ \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}}$$

$$[\text{try}]_{\text{wtot}}\ \frac{\{c\}\ C\ \{\top_{\mathfrak{A}}\} \quad \{\text{SE}_{\mathfrak{A}}[c, C]\}\ C; P\ \{d\} \quad \{\text{FE}_{\mathfrak{A}}[c, C]\}\ Q\ \{d\}}{\{c\}\ \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}}$$

$$[!]_{\text{wtot}}\ \frac{\{inv\}\ P\ \{inv\} \quad P \text{ is } \#\text{-decreasing under } inv}{\{inv\}\ P!\ \{\text{FE}_{\mathfrak{A}}[inv, P]\}}$$

Figure A.3: Weak total correctness proof rules for core commands

[skip], [fail], [or] of Figure A.2, and:

$$[\text{if}_2]_{\text{wtot}}\ \frac{\{c\}\ C\ \{\top_{\mathfrak{A}}\}\quad \{\text{SE}_{\mathfrak{A}}[c,C]\}\ P\ \{d\}\quad \text{impl}(\text{FE}_{\mathfrak{A}}[c,C],d)}{\{c\}\ \texttt{if}\ C\ \texttt{then}\ P\ \{d\}}$$

$$[\text{try}_2]_{\text{wtot}}\ \frac{\{c\}\ C\ \{\top_{\mathfrak{A}}\}\quad \{\text{SE}_{\mathfrak{A}}[c,C]\}\ C;\ P\ \{d\}\quad \text{impl}(\text{FE}_{\mathfrak{A}}[c,C],d)}{\{c\}\ \texttt{try}\ C\ \texttt{then}\ P\ \{d\}}$$

$$[\text{try}_3]_{\text{wtot}}\ \frac{\{c\}\ C\ \{\top_{\mathfrak{A}}\}\quad \{\text{SE}_{\mathfrak{A}}[c,C]\}\ C\ \{d\}\quad \text{impl}(\text{FE}_{\mathfrak{A}}[c,C],d)}{\{c\}\ \texttt{try}\ C\ \{d\}}$$

$\text{impl}(a_1,a_2)$: prove that $G \models_{\mathfrak{A}} a_1$ implies $G \models_{\mathfrak{A}} a_2$ for all $G \in \mathcal{G}(\mathcal{L})$

Figure A.4: Weak total correctness proof rules for derived commands

[comp], [cons] of Figure A.1, and:

$$[\text{ruleset}]_{\text{tot}}\ \frac{\text{impl}(c,\text{SE}_{\mathfrak{A}}[c,\mathcal{R}])\quad \vdash_{\text{par}} \{c\}\ r\ \{d\}\ \text{for each}\ r\in\mathcal{R}}{\{c\}\ \mathcal{R}\ \{d\}}$$

$$[\text{if}]_{\text{tot}}\ \frac{\vdash_{\text{wtot}} \{c\}\ C\ \{\top_{\mathfrak{A}}\}\quad \{\text{SE}_{\mathfrak{A}}[c,C]\}\ P\ \{d\}\quad \{\text{FE}_{\mathfrak{A}}[c,C]\}\ Q\ \{d\}}{\{c\}\ \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}}$$

$$[\text{try}]_{\text{tot}}\ \frac{\vdash_{\text{wtot}} \{c\}\ C\ \{\top_{\mathfrak{A}}\}\quad \{\text{SE}_{\mathfrak{A}}[c,C]\}\ C;\ P\ \{d\}\quad \{\text{FE}_{\mathfrak{A}}[c,C]\}\ Q\ \{d\}}{\{c\}\ \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}}$$

$$[!]_{\text{tot}}\ \frac{\vdash_{\text{wtot}} \{inv\}\ P\ \{inv\}\quad P\ \text{is}\ \#\text{-decreasing under}\ inv}{\{inv\}\ P!\ \{\text{FE}_{\mathfrak{A}}[inv,P]\}}$$

$\text{impl}(a_1,a_2)$: prove that $G \models_{\mathfrak{A}} a_1$ implies $G \models_{\mathfrak{A}} a_2$ for all $G \in \mathcal{G}(\mathcal{L})$

Figure A.5: Total correctness proof rules for core commands

[skip], [or] of Figure A.2, and:

$$[\text{if}_2]_{\text{tot}}\ \frac{\vdash_{\text{wtot}} \{c\}\ C\ \{\top_{\mathfrak{A}}\} \quad \{\text{SE}_{\mathfrak{A}}[c, C]\}\ P\ \{d\} \quad \text{impl}(\text{FE}_{\mathfrak{A}}[c, C], d)}{\{c\}\ \texttt{if}\ C\ \texttt{then}\ P\ \{d\}}$$

$$[\text{try}_2]_{\text{tot}}\ \frac{\vdash_{\text{wtot}} \{c\}\ C\ \{\top_{\mathfrak{A}}\} \quad \{\text{SE}_{\mathfrak{A}}[c, C]\}\ C; P\ \{d\} \quad \text{impl}(\text{FE}_{\mathfrak{A}}[c, C], d)}{\{c\}\ \texttt{try}\ C\ \texttt{then}\ P\ \{d\}}$$

$$[\text{try}_3]_{\text{tot}}\ \frac{\vdash_{\text{wtot}} \{c\}\ C\ \{\top_{\mathfrak{A}}\} \quad \{\text{SE}_{\mathfrak{A}}[c, C]\}\ C\ \{d\} \quad \text{impl}(\text{FE}_{\mathfrak{A}}[c, C], d)}{\{c\}\ \texttt{try}\ C\ \{d\}}$$

$\text{impl}(a_1, a_2)$: prove that $G \models_{\mathfrak{A}} a_1$ implies $G \models_{\mathfrak{A}} a_2$ for all $G \in \mathcal{G}(\mathcal{L})$

Figure A.6: Total correctness proof rules for derived commands

Appendix B

Proof Rules with E-Constraints

$$[\text{ruleapp}]_{\text{wlp}}\ \frac{}{\{\text{Pre}(r,c) \vee \neg\text{App}(\{r\})\}\ r\ \{c\}}$$

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

$$[\text{nonapp}]\ \frac{}{\{\neg\text{App}(\{r\})\}\ r\ \{\texttt{false}\}}$$

$$[\text{ruleset}]\ \frac{\{c\}\ r\ \{d\} \text{ for each } r \in \mathcal{R}}{\{c\}\ \mathcal{R}\ \{d\}}$$

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

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

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

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

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

Figure B.1: Partial correctness proof rules with E-constraints for core commands

$$[\text{skip}]\ \frac{}{\{c\}\ \texttt{skip}\ \{c\}}$$

$$[\text{fail}]\ \frac{}{\{\texttt{true}\}\ \texttt{fail}\ \{\texttt{false}\}}$$

$$[\text{or}]\ \frac{\{c\}\ P\ \{d\} \quad \{c\}\ Q\ \{d\}}{\{c\}\ P\ \texttt{or}\ Q\ \{d\}}$$

$$[\text{if}_2]\ \frac{\{c \wedge \text{App}(\mathcal{R})\}\ P\ \{d\} \quad c \wedge \neg\text{App}(\mathcal{R}) \Rightarrow d}{\{c\}\ \texttt{if}\ \mathcal{R}\ \texttt{then}\ P\ \{d\}}$$

$$[\text{try}_2]\ \frac{\{c \wedge \text{App}(\mathcal{R})\}\ \mathcal{R}; P\ \{d\} \quad c \wedge \neg\text{App}(\mathcal{R}) \Rightarrow d}{\{c\}\ \texttt{try}\ \mathcal{R}\ \texttt{then}\ P\ \{d\}}$$

$$[\text{try}_3]\ \frac{\{c \wedge \text{App}(\mathcal{R})\}\ \mathcal{R}\ \{d\} \quad c \wedge \neg\text{App}(\mathcal{R}) \Rightarrow d}{\{c\}\ \texttt{try}\ \mathcal{R}\ \{d\}}$$

Figure B.2: Partial correctness proof rules with E-constraints for derived commands

[ruleapp]$_{\text{wlp}}$, [ruleapp], [nonapp], [ruleset], [comp], [cons] of Figure B.1, and:

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

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

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

Figure B.3: Weak total correctness proof rules with E-constraints for core commands

[skip], [fail], [or] of Figure B.2, and:

$$[\text{if}_2]_{\text{wtot}}\ \frac{\{c \wedge \text{App}(\mathcal{R})\}\ P\ \{d\} \quad c \wedge \neg\text{App}(\mathcal{R}) \Rightarrow d}{\{c\}\ \texttt{if}\ \mathcal{R}\ \texttt{then}\ P\ \{d\}}$$

$$[\text{try}_2]_{\text{wtot}}\ \frac{\{c \wedge \text{App}(\mathcal{R})\}\ \mathcal{R}; P\ \{d\} \quad c \wedge \neg\text{App}(\mathcal{R}) \Rightarrow d}{\{c\}\ \texttt{try}\ \mathcal{R}\ \texttt{then}\ P\ \{d\}}$$

$$[\text{try}_3]_{\text{wtot}}\ \frac{\{c \wedge \text{App}(\mathcal{R})\}\ \mathcal{R}\ \{d\} \quad c \wedge \neg\text{App}(\mathcal{R}) \Rightarrow d}{\{c\}\ \texttt{try}\ \mathcal{R}\ \{d\}}$$

Figure B.4: Weak total correctness proof rules with E-constraints for derived commands

[comp], [cons] of Figure B.1, and:

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

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

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

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

Figure B.5: Total correctness proof rules with E-constraints for core commands

[skip], [or] of Figure B.2, and:

$$[\text{if}_2]_{\text{tot}}\ \frac{\{c \wedge \text{App}(\mathcal{R})\}\ P\ \{d\} \quad c \wedge \neg\text{App}(\mathcal{R}) \Rightarrow d}{\{c\}\ \texttt{if}\ \mathcal{R}\ \texttt{then}\ P\ \{d\}}$$

$$[\text{try}_2]_{\text{tot}}\ \frac{\{c \wedge \text{App}(\mathcal{R})\}\ \mathcal{R}; P\ \{d\} \quad c \wedge \neg\text{App}(\mathcal{R}) \Rightarrow d}{\{c\}\ \texttt{try}\ \mathcal{R}\ \texttt{then}\ P\ \{d\}}$$

$$[\text{try}_3]_{\text{tot}}\ \frac{\{c \wedge \text{App}(\mathcal{R})\}\ \mathcal{R}\ \{d\} \quad c \wedge \neg\text{App}(\mathcal{R}) \Rightarrow d}{\{c\}\ \texttt{try}\ \mathcal{R}\ \{d\}}$$

Figure B.6: Total correctness proof rules with E-constraints for derived commands

Appendix C

Facts about Pushouts and Pullbacks

This appendix is a reference for some well-known definitions and facts about pushout and pullback constructions. These are made use of in some of the constructions and proofs given for transformations of E-conditions in Chapter 4. We do not cover the double-pushout approach to graph transformation rules in this appendix, which is covered more throughly in Chapter 2.

The definitions and facts presented here are given in [HMP01, HP02]. The first of the papers contains in its own appendix several more facts which, while not required for this thesis, may help to provide a broader understanding of the constructions. Readers interested in category theory – the theoretical background of these concepts – can consult a number of text books, for example, [Lan98, Awo10].

Assume that in the graphs that follow, edges are totally labelled but nodes are partially labelled.

Definition C.1 (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 C.1 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 $C \to D \to D' = C \to D'$.

□

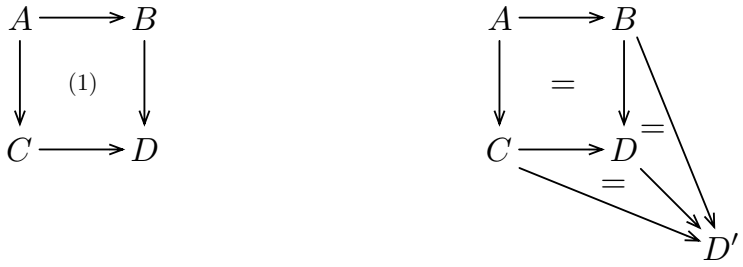


Figure C.1: A pushout and the universal property of pushouts

Definition C.2 (Pushout complement)**.** Given graph morphisms $A \to B$ and $B \to D$, a *pushout complement* of these morphisms is a graph C together with two morphisms $A \to C$ and $C \to D$ if the resulting diagram (as in (1) of Figure C.1) is a pushout. □

Definition C.3 (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 C.2 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 $A' \to A \to C = A' \to C$.

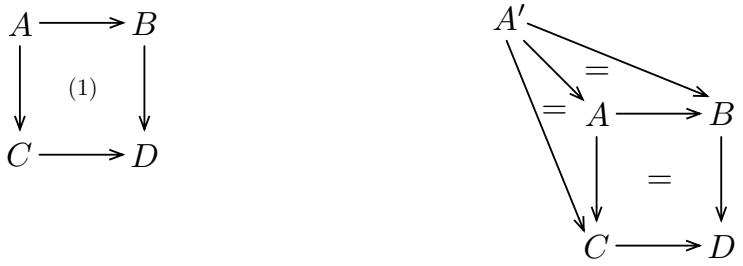


Figure C.2: A pullback and the universal property of pullbacks

□

Definition C.4 (Natural pushout)**.** A diagram (1) as in Figure C.1 is a *natural pushout* if it is both a pushout and a pullback. □

Lemma C.5 (Existence of pullbacks)**.** Given graph morphisms $B \to D$ and $C \to D$, there exists a graph A and morphisms $A \to B$ and $A \to C$ such that the arising diagram is a pullback. □

Proof. By Lemma 1 of [HP02]. □

The following lemma describes the conditions necessary for a pushout to exist for our graphs with partially labelled nodes. For simplicity, we give a more restricted version of the lemma in the cited paper, since the rules in this thesis comprise injective morphisms only, and both L and R are required to be totally labelled.

Lemma C.6 (Existence of pushouts)**.** Given an injective graph morphism $A \hookrightarrow B$ with B totally labelled, and an injective morphism $A \hookrightarrow C$ that preserves undefinedness, there exists a graph D and morphisms $B \hookrightarrow D$ and $C \hookrightarrow D$ such that the arising diagram is a pushout. □

Proof. By Lemma 2 of [HP02]. □

Lemma C.7 (Existence, uniqueness of natural PO complements)**.** Given an injective graph morphisms $A \to B$ and $B \to D$ with B, D totally labelled. There exists a graph C with injective morphisms $A \to C$ and $C \to D$ such that the arising diagram is a natural pushout if and only if $B \to D$ satisfies the dangling condition with respect to $A \to B$. In this case, C is unique up to isomorphism. □

Proof. By Lemma 4 of [HP02]. □

The following lemmata are particularly useful in the correctness proofs for transformations of E-conditions.

Lemma C.8 (Basic decompositions)**.** The following decomposition properties relate to the commutative diagram in Figure C.3, in which all morphisms are injective, and all graphs are from $\mathcal{G}(\mathcal{L}_\perp)$.

Pushout decomposition If (1)+(2) and (1) are pushouts, then (2) is also a pushout.

Pullback decomposition If (1)+(2) and (2) are pullbacks, then (1) is also a pullback.

□

Proof. Diagram chase (see e.g. [Awo10]). □

The following decompositions are more specialised to our context, in that they consider whether morphisms preserve undefinedness and also the naturalness of pushouts.

Lemma C.9 (Special decompositions)**.** The following decomposition properties relate to the commutative diagram in Figure C.3, in which all morphisms are injective, and B, D, F are totally labelled.

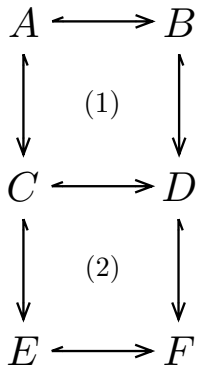


Figure C.3: Commutative diagram of morphisms

Pushout-pullback decomposition If (1)+(2) is a pushout with $A \to C \to E$ undefinedness preserving and (2) a pullback, then (1) and (2) are natural pushouts.

Pullback decomposition If (1)+(2) and (1) are pushouts and $A \to C \to E$ is undefinedness preserving, then (1) and (2) are pullbacks.

□

Proof. The paper [HP12] proves that the category of partially labelled graphs and their morphisms form a so-called $\mathcal{M}, \mathcal{N}$-adhesive category. The decomposition properties are properties of $\mathcal{M}, \mathcal{N}$-adhesive categories (see Theorem 1 in [HP12][1]). □

[1]The second property is given only in the long version of the authors' paper.

Appendix D

Additional Lemmata

This appendix contains some additional technical lemmata applied in proofs within the main text.

D.1 Additional Lemmata for Cond

Lemma D.1 (Many-sorted expressions to assignment constraints)**.** For all many-sorted expressions t in List, and all injective graph morphisms $z : X^\alpha \hookrightarrow G$ with X, α corresponding to the free node and edge variables of φ as in Remark 6.12, and α a well-typed assignment such that no variable in $\mathrm{dom}(\alpha)$ is quantified in φ, we have that:

$$I_G^{z,\alpha}(t) = \mathrm{AC}(t)^\alpha.$$

Here, $I_G^{z,\alpha}$ is defined as I_G but with the following mappings for free variables in φ: (1) for each variable $\mathtt{x}$ in $\mathrm{dom}(\alpha)$, $I_G^{z,\alpha}(\mathtt{x}) = \alpha(\mathtt{x})$; (2) for each node v in X, $I_G^{z,\alpha}(\mathtt{v}) = z(v)$; and (3) for each edge e in X, $I_G^{z,\alpha}(\mathtt{e}) = z(e)$. □

Proof. We prove that the equality holds for all expressions that t might take. Suppose that $t = \mathtt{l(v)}$ for some v in VVar. With the definition of graph morphisms and assignment:

$$I_G^{z,\alpha}(t) = l_G(z(v)) = l_{X^\alpha}(v) = l_X(v)^\alpha = (\mathtt{v}')^\alpha = \mathrm{AC}(t)^\alpha.$$

The proof is analogous for the cases when t is $\mathtt{l(s(e))}$ and $\mathtt{l(s(e))}$.

Suppose that $t = \mathtt{m(e)}$ with e in EVar. With the definition of graph morphisms and assignment:

$$I_G^{z,\alpha}(t) = m_G(z(e)) = m_{X^\alpha}(e) = m_X(e)^\alpha = (\mathtt{e}')^\alpha = \mathrm{AC}(t)^\alpha.$$

All other cases follow from the induction hypothesis, the definition of $I_G^{z,\alpha}$, and the definition of assignment. □

Lemma D.2 (Formulae can be expressed as E-conditions)**.** Let φ denote a formula without primed variables and variables quantified more than once. For all injective graph morphisms $z\colon X^\alpha \hookrightarrow G$ with X, α corresponding to the free node and edge variables of φ as in Remark 6.12, and α a well-typed assignment such that no variable in $\mathrm{dom}(\alpha)$ is quantified in φ, we have that:

$$I_G^{z,\alpha} \models \varphi \text{ if and only if } z\colon X^\alpha \hookrightarrow G \models \mathrm{Cond}'(\varphi, X)^{\sigma_\alpha}.$$

Here, $I_G^{z,\alpha}$ is defined as I_G but with the following mappings for free variables in φ: (1) for each variable `x` in $\mathrm{dom}(\alpha)$, $I_G^{z,\alpha}(\mathtt{x}) = \alpha(\mathtt{x})$; (2) for each node v in X, $I_G^{z,\alpha}(\mathtt{v}) = z(v)$; and (3) for each edge e in X, $I_G^{z,\alpha}(\mathtt{e}) = z(e)$. □

Proof. Induction basis. Assume that $I_G^{z,\alpha} \models \varphi$. Suppose that φ is equal to `true` or `false`; the statement trivially holds.

Suppose that φ is equal to $t\,\mathtt{(}\,l\,\mathtt{)}$ with t in Type and l in List. By the definition of $\models$, Lemma D.1 and the semantics of assignment constraints:

$$I_G^{z,\alpha}(t)(I_G^{z,\alpha}(l)) = t\,\mathtt{(}\,\mathrm{AC}(l)\,\mathtt{)}^{\,\alpha}.$$

By construction, $\mathrm{Cond}'(\varphi, X)^{\sigma_\alpha} = \exists(id_X : X^{\sigma_\alpha} \hookrightarrow X^{\sigma_\alpha} \mid t\,\mathtt{(}\,\mathrm{AC}(l)\,\mathtt{)}^{\,\sigma_\alpha})$. Since for any assignment α', $z \circ id_X^{\alpha'} = z$, the satisfaction of the E-condition depends entirely on the assignment constraint $t\,\mathtt{(}\,\mathrm{AC}(l)\,\mathtt{)}^{\,\sigma_\alpha}$, which evaluates to true under any assignment if and only if $I_G^{z,\alpha} \models \varphi$.

Suppose that φ is equal to $i_1 \bowtie i_2$ (resp. $l_1 = l_2$) with i_1, i_2 in Integer and $\bowtie$ in IntRel (resp. l_1, l_2 in List). Cases follow analogously to the previous one.

Suppose that φ is equal to `e = f` with `e`, `f` in EVar. Then by the definition of $I_G^{z,\alpha}$, the injectivity of z, and the construction:

$$I_G^{z,\alpha}(=)(I_G^{z,\alpha}(\mathtt{e}), I_G^{z,\alpha}(\mathtt{f})) = (z(e) = z(f)) = (e = f) = \mathrm{Cond}'(\varphi, X)^\alpha$$

hence $I_G^{z,\alpha} \models \varphi$ if and only if $z \models \mathrm{Cond}'(\varphi, X)^{\sigma_\alpha}$.

Suppose that φ is equal to $v_1 = v_2$ with v_1, v_2 in Vertex. The case follow analogously to the previous via the helper function VertexID.

Finally, suppose that φ is equal to `marked(e)` with `e` in EVar. Then by the definition of $I_G^{z,\alpha}$ and the construction:

$$\begin{aligned} I_G^{z,\alpha}(\mathtt{marked})(I_G^{z,\alpha}(\mathtt{e})) &= (l_G(z(e)) = (\_\ \mathsf{true})) \\ &= (l_{X^\alpha}(e) = (\_\ \mathsf{true})) \\ &= \mathrm{Cond}'(\varphi, X)^\alpha \end{aligned}$$

hence $I_G^{z,\alpha} \models \varphi$ if and only if $z \models \mathrm{Cond}'(\varphi, X)^{\sigma_\alpha}$.

Induction step. Only if. Assume that $I_G^{z,\alpha} \models \varphi$. For cases when φ is equal to $\neg\varphi_1$ or $\varphi_1 \oplus \varphi_2$ with φ_1, φ_2 in Formula and $\oplus$ in BoolOp, the proof follows in the standard way via the induction hypothesis.

Suppose that φ is equal to $\exists \mathtt{x} : \mathtt{L}.\ \varphi'$ with φ' in Formula. Then there is some $l \in \mathbb{L}$ such that $I_G^{z,\alpha'} \models \varphi'$ where α' is equal to α but with the additional mapping $\mathtt{x} \mapsto l$. By induction hypothesis, $z : X^{\alpha'} \hookrightarrow G \models \mathrm{Cond}'(\varphi', X)^{\sigma_{\alpha'}}$. With the construction, the fact that $\mathtt{x} = \mathtt{x}$ under α' evaluates to true, that $X^\alpha = X^{\alpha'}$, $z \circ id_X^{\alpha'} = z$, and the definition of $\models$, we get:

$$z \colon X^\alpha \hookrightarrow G \models_{\alpha'} \exists(id_X \colon X \hookrightarrow X \mid \mathtt{x} = \mathtt{x}, \mathrm{Cond}'(\varphi', X)) = \mathrm{Cond}'(\varphi, X).$$

and then using the definition of induced substitutions, the result that $z \models \mathrm{Cond}'(\varphi, X)^{\sigma_\alpha}$.

For cases when atom, integer, or string variables are existentially quantified, the proofs follow analogously, using the fact that the sorting forces variables to be interpreted as elements of the domains with which the corresponding type predicates evaluate to true.

Suppose that φ is equal to $\exists \mathtt{v} : \mathtt{V}.\ \varphi'$ with φ' in Formula. Then there is a node $x \in V_G$ such that $I_G^{z,\alpha} \cup \{\mathtt{v} \mapsto x\} \models \varphi'$. That node could already be in the image of z; but it might instead be outside of it. If the former, i.e. there is some node $y \in V_X$ such that $z(y) = x$, then define X' to be X but with y and v identified. If the latter, disjointly add v to X to yield X'. Define $l_{X'}(v) = \mathtt{v}'$ and α' to be α with the additional mapping $\mathtt{v}' \mapsto l_G(x)$. (Clearly, X', α' conform to Remark 6.12.) Define $z' : (X')^{\alpha'} \hookrightarrow G$ as z but with $z'(v) = x$. We have that $I_G^{z',\alpha'} \models \varphi'$; by induction hypothesis, $z' : (X')^{\alpha'} \hookrightarrow G \models \mathrm{Cond}'(\varphi', X')^{\sigma_{\alpha'}}$. Furthermore, $(\mathtt{v}' = l_{X'}(v))^{\alpha'} = \mathrm{true}$, $X^\alpha = X^{\alpha'}$, and $z' \circ (X \hookrightarrow X')^{\alpha'} = z$; together we get that:

$$z \colon X^\alpha \hookrightarrow G \models_{\alpha'} \exists(X \hookrightarrow X' \mid \mathtt{v}' = l_{X'}(v), \mathrm{Cond}'(\varphi', X'))$$

i.e. a disjunct of $\mathrm{Cond}'(\varphi, X)$. With the definition of induced substitutions, we get the result that $z \models \mathrm{Cond}'(\varphi, X)^{\sigma_\alpha}$.

Suppose that φ is equal to $\exists \mathtt{e} : \mathtt{E}.\ \varphi'$ with φ' in Formula. The proof for this case follows analogously to the previous one.

Finally, suppose that φ is equal to $\forall \mathtt{x} : S.\ \varphi'$ with φ' in Formula and S any of the sort symbols. The result follows in the standard way via the induction hypothesis and the well-known property of universally quantified formulae.

Induction step. If. Assume that $z \colon X^\alpha \hookrightarrow G \models \mathrm{Cond}'(\varphi, X)^{\sigma_\alpha}$. For cases when φ is equal to $\neg\varphi_1$ or $\varphi_1 \oplus \varphi_2$ with φ_1, φ_2 in Formula and $\oplus$ in BoolOp, the proof follows in the standard way via the induction hypothesis.

Suppose that φ is equal to $\exists \mathtt{x} : \mathtt{L}.\ \varphi'$ with φ' in Formula. By construction and assumption, $z \colon X^\alpha \hookrightarrow G \models_{\alpha'} \exists(id_X \colon X \hookrightarrow X \mid \mathtt{x} = \mathtt{x}, \mathrm{Cond}'(\varphi', X))^{\sigma_\alpha}$

where α' is defined as α but with an additional mapping $\mathtt{x} \mapsto l$ for some $l \in \mathbb{L}$; in particular, $z \circ id_X^{\alpha'} = z$ and $z : X^{\alpha'} \hookrightarrow G \models \mathrm{Cond}'(\varphi', X)^{\sigma_{\alpha'}}$. By induction hypothesis, $I_G^{z,\alpha'} \models \varphi'$. Extracting the mapping for $\mathtt{x}$, $I_G^{z,\alpha} \cup \{\mathtt{x} \mapsto l\} \models \varphi'$; by definition of $\models$, we get $I_G^{z,\alpha} \models \varphi$, the result.

For cases when atom, integer, or string variables are existentially quantified, the proofs follow analogously, using the fact that the type predicates correspond to the sorts of the variables in the many-sorted formulae.

Suppose that φ is equal to $\exists \mathtt{v} : \mathtt{V}\,.\ \varphi'$ with φ' in Formula. By construction and assumption, $z : X^{\alpha} \hookrightarrow G \models_{\alpha'} \exists(X \hookrightarrow X' \mid \mathtt{v}' = l_{X'}(v), \mathrm{Cond}'(\varphi', X'))^{\sigma_\alpha}$, where the E-condition is a disjunct arising from $\mathrm{Cond}'(\varphi, X)$, X' is as X but containing a new node identifier v (either identified with an existing node, or a distinct node), and α' is defined as α but with an additional mapping $\mathtt{v}' \mapsto l$ for some $l \in \mathbb{L}$. In particular, there is a morphism $q : (X')^{z,\alpha'} \hookrightarrow G$ such that $q' \circ (X \hookrightarrow X')^{\alpha'} = z$, and $q \models \mathrm{Cond}'(\varphi', X')^{\sigma_{\alpha'}}$. (Clearly, by the construction and assignment constraint, X', α' conform to Remark 6.12.) By induction hypothesis, $I_G^{q,\alpha'} \models \varphi'$. Extracting the node v from q, we get $I_G^{z,\alpha'} \cup \{\mathtt{v} \mapsto v\} \models \varphi'$; by definition of $\models$, and the assumption that φ' is free from primed variables, we get $I_G^{z,\alpha} \models \varphi$, the result.

Suppose that φ is equal to $\exists \mathtt{e} : \mathtt{E}\,.\ \varphi'$ with φ' in Formula. The proof for this case follows analogously to the previous one.

Finally, suppose that φ is equal to $\forall \mathtt{x} : S\,.\ \varphi'$ with φ' in Formula and S any of the sort symbols. The result follows in the standard way via the induction hypothesis and the well-known property of universally quantified formulae. □

D.2 Additional Lemmata for the Normalisation of E-Conditions

Lemma D.3 (Replacing lists)**.** Let c denote an E-condition. There is a transformation Relabel that yields an E-condition meeting the second requirement of Definition 6.20, such that for all injective graph morphisms s over $\mathcal{G}(\mathcal{L})$,

$$s \models c \text{ if and only if } s \models \mathrm{Relabel}(c).$$

□

Proof. Define $s : P^{\alpha} \hookrightarrow G$ with $G \in \mathcal{G}(\mathcal{L})$ and α a well-typed assignment. We proceed by structural induction.

Induction basis. Let $c = \mathtt{true}$. By construction, $\mathrm{Relabel}(c) = \mathtt{true}$. Trivially the statement holds.

Induction step. Only if. Assume that $s \models c = \exists(a : P \hookrightarrow C \mid \gamma, c')$, i.e. there exists a well-typed assignment α (we overload the assignment

in s) and morphism $q : C^\alpha \hookrightarrow G$ such that $q \circ a^\alpha = s$, $\gamma^\alpha = \text{true}$, and $q \models (c')^{\sigma_\alpha}$. Clearly we also have $q \models c'$ with a satisfying assignment that has least the mappings of α; applying the induction hypothesis gives us $q \models \text{Relabel}(c')^{\sigma_\alpha}$.

Define $\overline{a} : \overline{P} \hookrightarrow \overline{C}$ where $\overline{P}, \overline{C}$ are P, C but with all list components replaced by fresh and distinct variables. Define $\overline{\gamma}$ as a conjunction of γ with $l_{\overline{C}}(v) = l_C(v)$ for all nodes v in C where $l_{\overline{C}}(v) \neq l_C(v)$ (and analogously for edges). Define $\overline{c}'$ to be c', but for the nodes and edges of its morphisms that are also in C, replace their labels with the corresponding labels of $\overline{C}$ (recall that the morphisms are inclusions). There is a well-typed assignment α' with at least the mappings of α such that $\overline{\gamma}^{\alpha'} = \text{true}$, $\overline{a}^{\alpha'} = a^\alpha$, and $\sigma_{\alpha'}(l_{\overline{C}}(v)) = l_C(v)^{\sigma_\alpha}$ for all nodes v in C (and analogously for edges). Together we get that $s \models_{\alpha'} \exists(\overline{a} \mid \overline{\gamma}, \text{Relabel}(\overline{c}'))$; we take this to be the E-condition from the construction, and hence the result that $s \models \text{Relabel}(c)$.

Induction step. If. Assume that $s \models \text{Relabel}(c)$. The proof is similar to the "only if" direction, but simpler, as the assignment by which s satisfies $\text{Relabel}(c)$ is also an assignment by which s satisfies c (due to the conjuncts in the assignment constraints added by the construction).

For Boolean formulae over E-conditions, the result follows from the construction and induction hypothesis. □

Lemma D.4 (Sorting assignment constraints)**.** Let c be an E-condition adhering to the first two requirements of normal form (Definition 6.20). There is a transformation Sort which yields an E-condition meeting the third requirement, such that for all injective graph morphisms s over $\mathcal{G}(\mathcal{L})$,

$$s \models c \text{ if and only if } s \models \text{Sort}(c).$$

□

Proof sketch. The statement follows easily when it is observed that (1) Sort replaces variables with fresh variables, which can be mapped to the same values in $\mathbb{L}$; and (2) it is impossible to satisfy an E-condition $\exists(a \mid \gamma, c')$ if a variable appears in both a string and an integer expression in a or γ, and moreover, if it appears in both types of expressions in c' (since substitutions induced by satisfying assignments must be well-typed, by definition of $\models$).

That the assignment constraints of $\text{Sort}(c)$ are many-sorted (up to a renaming of `and`, `or` for $\wedge, \vee$) can be shown by appealing to the subsort hierarchy, the definition of assignment constraints, the definition of many-sorted formulae, and the fact that E-conditions containing variables in both types of expressions are replaced by `false`. □

D.3 Additional Lemmata for Form

Lemma D.5 (E-conditions can be expressed as formulae)**.** For any given E-condition c in normal form, and all injective graph morphisms $p\colon P^\alpha \hookrightarrow G$ with $P \in \mathcal{G}(\mathrm{EC}), G \in \mathcal{G}(\mathcal{L})$, and α a well-typed assignment, we have that:

$$p\colon P^\alpha \hookrightarrow G \models c^{\sigma_\alpha} \text{ if and only if } I_G^{p,\alpha} \models \mathrm{Form}'(c, \mathrm{dom}(\alpha) \cup P^*)$$

where P^* is the set of node and edge variables corresponding to the identifiers in P. Additionally, $I_G^{p,\alpha}$ is defined as I_G but with the following mappings: (1) for each variable $\mathtt{x}$ in $\mathrm{dom}(\alpha)$, $I_G^{p,\alpha}(\mathtt{x}) = \alpha(\mathtt{x})$; (2) for each node v in P, $I_G^{p,\alpha}(\mathtt{v}) = p(v)$; and (3) for each edge e in P, $I_G^{p,\alpha}(\mathtt{e}) = p(e)$. □

Proof. Induction basis. Let $c = \mathtt{true}$. Then by the construction and definition of $\models$, the statement trivially holds.

Induction step. Only if. Assume that $p\colon P^\alpha \hookrightarrow G \models c^{\sigma_\alpha}$. Suppose that $c = \exists(id : P \hookrightarrow P \mid \gamma, c')$. Then there exists a well-typed assignment α' such that $p \circ id^{\alpha'} = p$, $\gamma^{\alpha'} = \mathrm{true}$, and $p\colon P^{\alpha'} \hookrightarrow G \models (c')^{\sigma_{\alpha'}}$. By induction hypothesis, $I_G^{p,\alpha'} \models \mathrm{Form}'(c', \mathrm{dom}(\alpha') \cup P^*)$. We assume that α' is minimal, in that it only introduces mappings for variables in γ not also in P; hence $\mathrm{dom}(\alpha') \cup P^* = \mathrm{dom}(\alpha) \cup P^* \cup \mathrm{vars}(\gamma)$ and:

$$I_G^{p,\alpha'} \models \mathrm{Form}'(c', \mathrm{dom}(\alpha) \cup P^* \cup \mathrm{vars}(\gamma)).$$

With the assumption that $\gamma^{\alpha'} = \mathrm{true}$ and the definitions of $\gamma^*, \models$, we get:

$$I_G^{p,\alpha'} \models \gamma^* \wedge \mathrm{Form}'(c', \mathrm{dom}(\alpha) \cup P^* \cup \mathrm{vars}(\gamma)).$$

Finally, we "extract" the additional mappings of α' for γ^* via the definition of $\models$ to get:

$$I_G^{p,\alpha} \models \mathrm{Quant}(\mathrm{vars}(\gamma) - [\mathrm{dom}(\alpha) \cup P^*])\,.\, \gamma^* \wedge \mathrm{Form}'(c', \mathrm{dom}(\alpha) \cup P^* \cup \mathrm{vars}(\gamma))$$

which is the formulae from the construction, i.e. the result that $I_G^{p,\alpha} \models \mathrm{Form}'(c, \mathrm{dom}(\alpha) \cup P^*)$.

Suppose that $c = \exists([v\mathtt{a}] : P \hookrightarrow P' \mid \gamma, c')$ with P' equal to P except for an additional unmarked node v labelled by variable $\mathtt{a}$. Then there exists a well-typed assignment α' and morphism $q : (P')^{\alpha'} \hookrightarrow G$ such that $q \circ [v\mathtt{a}]^{\alpha'} = p$, $\gamma^{\alpha'} = \mathrm{true}$, and $q\colon (P')^{\alpha'} \hookrightarrow G \models (c')^{\sigma_{\alpha'}}$. By induction hypothesis, $I_G^{q,\alpha'} \models \mathrm{Form}'(c', \mathrm{dom}(\alpha') \cup (P')^*)$. We assume that α' is minimal, in that it only introduces mappings for $\mathtt{a}$ and variables in γ not also in P; hence $\mathrm{dom}(\alpha') \cup (P')^* = \mathrm{dom}(\alpha) \cup P^* \cup \mathrm{vars}(\gamma) \cup \{\mathtt{v}, \mathtt{a}\}$ and:

$$I_G^{q,\alpha'} \models \mathrm{Form}'(c', \mathrm{dom}(\alpha) \cup P^* \cup \mathrm{vars}(\gamma) \cup \{\mathtt{v}, \mathtt{a}\}).$$

For brevity in the following, let $F \equiv \text{Form}'(c', \text{dom}(\alpha) \cup P^* \cup \text{vars}(\gamma) \cup \{\mathtt{v}, \mathtt{a}\})$. With the assumption and definition of γ^*, $I_G^{q,\alpha'} \models \gamma^*$. With the definitions of $I_G^{q,\alpha'}$ and label preservation of morphisms, we have:

$$\begin{aligned} I_G^{q,\alpha'}(=)(I_G^{q,\alpha'}(\mathtt{l\,(v)}), I_G^{q,\alpha'}(\mathtt{a})) &= (l_G(q(v)) = \alpha'(\mathtt{a})) \\ &= (l_G(q(v)) = l_{(P')^{\alpha'}}(v)) \\ &= \text{true} \end{aligned}$$

and so $I_G^{q,\alpha'} \models \mathtt{l\,(v)}$; together we get:

$$I_G^{q,\alpha'} \models \mathtt{l\,(v)} \wedge \gamma^* \wedge F.$$

"Extracting" the additional mappings (as in the previous case) yields:

$$I_G^{q,\alpha} \models \text{Quant}([\text{vars}(\gamma) \cup \{\mathtt{a}\}] - \text{dom}(\alpha) \cup P^*)\,\mathtt{.\ l\,(v)} \wedge \gamma^* \wedge F.$$

For brevity let F' denote this E-condition. Since v in P' is unmarked and morphisms are label preserving, $q(v)$ is also unmarked. Hence:

$$I_G^{q,\alpha}(\mathtt{marked})(I_G^{q,\alpha}(\mathtt{v})) = (l_G(q(v)) = (\_\,\text{true})) = \text{false}$$

and so $I_G^{q,\alpha} \models \neg\mathtt{marked\,(v)}$. Moreover, since v is distinct and q is injective, $q(v)$ is distinct from all other nodes in the image of p, and so $I_G^{q,\alpha} \models \bigwedge_{\mathtt{v}' \in P^* \cap \text{VVar}} \neg\mathtt{v} = \mathtt{v}'$. "Extracting" $q(v)$ and with the definition of $\models$, we get:

$$I_G^{p,\alpha} \models \exists \mathtt{v} : \mathtt{V}\,\mathtt{.} \bigwedge_{\mathtt{v}' \in P^* \cap \text{VVar}} \neg\mathtt{v} = \mathtt{v}' \wedge \neg\mathtt{marked\,(v)} \wedge F'$$

which is the E-condition from the construction, i.e. the result that $I_G^{p,\alpha} \models \text{Form}'(c, \text{dom}(\alpha) \cup P^*)$.

For other cases – when P' introduces a marked node, marked edge, or unmarked edge – the proof follows analogously to the previous case.

If. Assume that $I_G^{p,\alpha} \models \text{Form}'(c, \text{dom}(\alpha) \cup P^*)$. Suppose that $c = \exists(id : P \hookrightarrow P \mid \gamma, c')$. Then, by the construction, we have:

$$I_G^{p,\alpha} \models \text{Quant}(\text{vars}(\gamma) - [\text{dom}(\alpha) \cup P^*])\,\mathtt{.}\ \gamma^* \wedge \text{Form}'(c', \text{dom}(\alpha) \cup P^* \cup \text{vars}(\gamma)).$$

With α' denote the assignment comprising the mappings of α, and also additional mappings for variables in γ^* such that:

$$I_G^{p,\alpha'} \models \gamma^* \wedge \text{Form}'(c', \text{dom}(\alpha) \cup P^* \cup \text{vars}(\gamma)).$$

From the assumption, and the definition of γ^*, it is clear that $\gamma^{\alpha'} = \text{true}$. By the induction hypothesis, $p \colon P^{\alpha'} \hookrightarrow G \models (c')^{\sigma_\alpha}$. Together with the facts

that $P^{\alpha} = P^{\alpha'}$, $p \circ id^{\alpha'} = p$, and α denotes a subset of the mappings of α', we get the result that $p\colon P^{\alpha} \hookrightarrow G \models c^{\sigma_\alpha}$.

Suppose that $c = \exists([v\mathtt{a}] : P \hookrightarrow P' \mid \gamma, c')$ with P' equal to P except for an additional unmarked node v labelled by variable `a`. Then, by the construction, we have:

$$\begin{aligned} I_G^{p,\alpha} \models \exists \mathtt{v} : \mathtt{V}\mathtt{.}\ & \textstyle\bigwedge_{\mathtt{v}' \in V \cap \mathrm{VVar}} \neg \mathtt{v} = \mathtt{v}' \wedge \neg \mathtt{marked}\,(\mathtt{v}) \\ & \wedge \mathrm{Quant}([\mathrm{vars}(\gamma) \cup \{\mathtt{a}\}] - \mathrm{dom}(\alpha) \cup P^*) \\ & \quad \mathtt{.}\ \mathtt{l}\,(\mathtt{v})\ = \mathtt{a} \wedge \gamma^* \wedge \mathrm{Form}'(c', \mathrm{dom}(\alpha) \cup P^* \cup \mathrm{vars}(\gamma) \cup \{\mathtt{v}, \mathtt{a}\}) \end{aligned}$$

By the definition of $\models$, there is a morphism q, the domain of which contains P and some node v, and the codomain of which is G. From the definition of $I_G^{q,\alpha}$ and the assumption that:

$$I_G^{q,\alpha} \models \bigwedge_{\mathtt{v}' \in V \cap \mathrm{VVar}} \neg \mathtt{v} = \mathtt{v}' \wedge \neg \mathtt{marked}\,(\mathtt{v})$$

it must be the case that $q(v)$ is unmarked and distinct from all nodes in the image of p, i.e. q is injective. Moreover, there is an assignment α' comprising at least the assignments of α such that:

$$I_G^{q,\alpha'} \models \mathtt{l}\,(\mathtt{v})\ = \mathtt{a} \wedge \gamma^* \wedge \mathrm{Form}'(c', \mathrm{dom}(\alpha) \cup P^* \cup \mathrm{vars}(\gamma) \cup \{\mathtt{v}, \mathtt{a}\})$$

With the semantics of $I_G^{q,\alpha'}$, the first conjunct informs us that $l_G(q(v)) = \alpha'(\mathtt{a})$; and hence we can take $(P')^{\alpha'}$ to be the domain of $q : (P')^{\alpha'} \hookrightarrow G$, with $q \circ [v\mathtt{a}]^{\alpha'} = p$. Moreover, with the assumption and definition of γ^*, we get $\gamma^{\alpha'} = \text{true}$. Finally, noting that:

$$\mathrm{dom}(\alpha) \cup P^* \cup \mathrm{vars}(\gamma) \cup \{\mathtt{v}, \mathtt{a}\} = \mathrm{dom}(\alpha') \cup (P')^*$$

we apply the induction hypothesis to get $q : (P')^{\alpha'} \hookrightarrow G \models (c')^{\sigma_{\alpha'}}$. Together, and with the fact that α denotes a subset of the mappings of α', we get the result that $p\colon P^{\alpha} \hookrightarrow G \models c^{\sigma_\alpha}$.

For other cases – when P' introduces a marked node, marked edge, or unmarked edge – the proof follows analogously to the previous case.

For Boolean formulae over E-conditions, the result follows from the construction and induction hypothesis. □

Bibliography

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

[AH08] Karl Azab and Annegret Habel. High-level programs and program conditions. In *Proc. International Conference on Graph Transformation (ICGT 2008)*, volume 5214 of *LNCS*, pages 211–225. Springer, 2008.

[Apt81] Krzysztof R. Apt. Ten years of Hoare's logic: A survey - part I. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 3(4):431–483, 1981.

[Apt84] Krzysztof R. Apt. Ten years of Hoare's logic: A survey - part II: Nondeterminism. *Theoretical Computer Science*, 28:83–109, 1984.

[Awo10] Steve Awodey. *Category Theory*. Oxford Logic Guides. Oxford University Press, second edition, 2010.

[Bau06] Jörg Bauer. *Analysis of Communication Topologies by Partner Abstraction*. PhD thesis, Universität des Saarlandes, 2006.

[BBKR08] Jörg Bauer, Iovka Boneva, Marcos E. Kurbán, and Arend Rensink. A modal-logic based graph abstraction. In *Proc. International Conference on Graph Transformation (ICGT 2008)*, volume 5214 of *LNCS*, pages 321–335. Springer, 2008.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.

[BCHK11] H. J. Sander Bruggink, Raphaël Cauderlier, Mathias Hülsbusch, and Barbara König. Conditional Reactive Systems. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2011)*,

volume 13 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 191–203. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2011.

[BCK01] Paolo Baldan, Andrea Corradini, and Barbara König. A static analysis technique for graph transformation systems. In *Proc. Concurrency Theory (CONCUR 2001)*, volume 2154 of *LNCS*, pages 381–395. Springer, 2001.

[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.

[BET08] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Precise semantics of EMF model transformations by graph transformation. In *Proc. Model Driven Engineering Languages and Systems (MoDELS 2008)*, volume 5301 of *LNCS*, pages 53–67. Springer, 2008.

[BHP+92] Christoph Beierle, Ulrich Hedtstück, Udo Pletat, Peter H. Schmitt, and Jörg H. Siekmann. An order-sorted logic for knowledge representation systems. *Artificial Intelligence*, 55(2):149–191, 1992.

[BK02] Paolo Baldan and Barbara König. Approximating the behaviour of graph transformation systems. In *Proc. International Conference on Graph Transformation (ICGT 2002)*, volume 2505 of *LNCS*, pages 14–29. Springer, 2002.

[BP12] Christopher Bak and Detlef Plump. Rooted graph programs. In *Proc. International Workshop on Graph Based Tools (GraBaTs 2012)*, volume 54 of *Electronic Communications of the EASST*, 2012.

[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.

[BPR04a] Adam Bakewell, Detlef Plump, and Colin Runciman. Checking the shape safety of pointer manipulations. In *Int. Seminar on Relational Methods in Computer Science (RelMiCS 7), Revised Selected Papers*, volume 3051 of *LNCS*, pages 48–61. Springer, 2004.

[BPR04b] Adam Bakewell, Detlef Plump, and Colin Runciman. Specifying pointer structures by graph reduction. In *Applications of*

Graph Transformations With Industrial Relevance (AGTIVE 2003), Revised Selected and Invited Papers, volume 3062 of *LNCS*, pages 30–44. Springer, 2004.

[BRRS08] Luciano Baresi, Vahid Rafe, Adel Torkaman Rahmani, and Paola Spoletini. An efficient solution for model checking graph transformation systems. In *Proc. Workshop on Graph Transformation for Concurrency and Verification (GT-VC 2007)*, volume 213 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2008.

[BS06] Luciano Baresi and Paola Spoletini. On the use of Alloy to analyze graph transformation systems. In *Proc. International Conference on Graph Transformation (ICGT 2006)*, volume 4178 of *LNCS*, pages 306–320. Springer, 2006.

[CE12] Bruno Courcelle and Joost Engelfriet. *Graph Structure and Monadic Second-Order Logic: A Language-Theoretic Approach*. Cambridge University Press, 2012.

[Cla85] E. M. Clarke. The characterization problem for Hoare logics. In *Proc. of a discussion meeting of the Royal Society of London on Mathematical logic and programming languages*, pages 89–106, Upper Saddle River, NJ, USA, 1985. Prentice-Hall, Inc.

[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. In *Handbook of Graph Grammars*, pages 163–246. World Scientific, 1997.

[Coo78] Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal of Computing*, 7(1):70–90, 1978.

[Cou90] Bruno Courcelle. Graph rewriting: An algebraic and logic approach. In *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 *Handbook of Graph Grammars*, pages 313–400. World Scientific, 1997.

[Cro72] John N. Crossley. *What is Mathematical Logic?* Oxford University Press, 1972.

[dCR12] Simone André da Costa and Leila Ribeiro. Verification of graph grammars using a logical approach. *Science of Computer Programming*, 77(4):480–504, 2012.

[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.

[DP09] Mike Dodds and Detlef Plump. From hyperedge replacement to separation logic and back. In *Proc. Doctoral Symposium at the International Conference on Graph Transformation (ICGT 2008)*, volume 16 of *Electronic Communications of the EASST*, 2009.

[EEdL+05] Hartmut Ehrig, Karsten Ehrig, Juan de Lara, Gabriele Taentzer, Dániel Varró, and Szilvia Varró-Gyapay. Termination criteria for model transformation. In *Proc. Fundamental Approaches to Software Engineering (FASE 2005)*, volume 3442 of *LNCS*, pages 49–63. Springer, 2005.

[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.

[EGH+12] Hartmut Ehrig, Ulrike Golas, Annegret Habel, Leen Lambers, and Fernando Orejas. $\mathcal{M}$-adhesive transformation systems with nested application conditions. Part 2: Embedding, critical pairs and local confluence. *Fundamenta Informaticae*, 118(1-2):35–63, 2012.

[EGH+13] Hartmut Ehrig, Ulrike Golas, Annegret Habel, Leen Lambers, and Fernando Orejas. $\mathcal{M}$-adhesive transformation systems with nested application conditions. Part 1: Parallelism, concurrency and amalgamation. *Mathematical Structures in Computer Science*, 2013. To appear.

[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.

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

[Gai82] H. Gaifman. On local and non-local properties. *Studies in Logic and the Foundations of Mathematics*, 107:105–135, 1982.

[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 of *LNCS*, pages 383–397. Springer, 2006.

[GC10] Mike Gordon and Hélène Collavizza. Forward with Hoare. *Reflections on the Work of C.A.R. Hoare*, pages 101–121, 2010.

[GdMR+12] Amir Hossein Ghamarian, Maarten de Mol, Arend Rensink, Eduardo Zambon, and Maria Zimakova. Modelling and analysis using GROOVE. *International Journal on Software Tools for Technology Transfer*, 14(1):15–40, 2012.

[GGZ+05] Lars Grunske, Leif Geiger, Albert Zündorf, Niels Van Eetvelde, Pieter Van Gorp, and Dániel Varró. Using graph transformation for practical model-driven software engineering. In Sami Beydeda, Matthias Book, and Volker Gruhn, editors, *Model-Driven Software Development*, pages 91–117. Springer, 2005.

[GL12] Holger Giese and Leen Lambers. Towards automatic verification of behavior preservation for model transformation via invariant checking. In *Proc. International Conference on Graph Transformation (ICGT 2012)*, volume 7562 of *LNCS*, pages 249–263. Springer, 2012.

[GM92] Joseph A. Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992.

[Hab92] Annegret Habel. *Hyperedge Replacement: Grammars and Languages*, volume 643 of *LNCS*. Springer, 1992.

[Har69] Frank Harary. *Graph Theory*. Addison-Wesley, 1st edition, 1969.

[HEOG10] Frank Hermann, Hartmut Ehrig, Fernando Orejas, and Ulrike Golas. Formal analysis of functional behaviour for model transformations based on triple graph grammars. In *Proc. Graph Transformations (ICGT 2010)*, volume 6372 of *LNCS*, pages 155–170. Springer, 2010.

[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.

[HHT96] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3/4):287–313, 1996.

[HJ00] Marieke Huisman and Bart Jacobs. Java program verification via a Hoare Logic with abrupt termination. In *Proc. Fundamental Approaches to Software Engineering (FASE 2000)*, volume 1783 of *LNCS*, pages 284–303. Springer, 2000.

[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 of *LNCS*, pages 230–245. Springer, 2001.

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

[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.

[HP12] Annegret Habel and Detlef Plump. *M,N*-adhesive transformation systems. In *Proc. International Conference on Graph Transformation (ICGT 2012)*, volume 7562 of *LNCS*, pages 218–233. Springer, 2012.

[HPR06] Annegret Habel, Karl-Heinz Pennemann, and Arend Rensink. Weakest preconditions for high-level programs. In *Proc. Graph Transformations (ICGT 2006)*, volume 4178 of *LNCS*, pages 445–460. Springer, 2006.

[HR10] Annegret Habel and Hendrik Radke. Expressiveness of graph conditions with variables. In *Proc. International Colloquium on Graph and Model Transformation (GraMoT 2010)*, volume 30 of *Electronic Communications of the EASST*, 2010.

[HW95] Reiko Heckel and Annika Wagner. Ensuring consistency of conditional graph rewriting - a constructive approach. *Electronic Notes in Theoretical Computer Science*, 2, 1995.

[HZG06] Karsten Hölscher, Paul Ziemann, and Martin Gogolla. On translating UML models into graph transformation systems. *Journal of Visual Languages Computing*, 17(1):78–105, 2006.

[Jac12] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, revised edition, 2012.

[KGKZ09] Sabine Kuske, Martin Gogolla, Hans-Jörg Kreowski, and Paul Ziemann. Towards an integrated graph-based semantics for UML. *Software and Systems Modeling*, 8:403–422, 2009.

[KK06] Barbara König and Vitali Kozioura. Counterexample-guided abstraction refinement for the analysis of graph transformation systems. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2006)*, volume 3920 of *LNCS*, pages 197–211, 2006.

[KK08] Barbara König and Vitali Kozioura. Towards the verification of attributed graph transformation systems. In *Proc. International Conference on Graph Transformation (ICGT 2008)*, volume 5214 of *LNCS*, pages 305–320, 2008.

[KKvT12] Hans-Jörg Kreowski, Sabine Kuske, and Caroline von Totth. Combining graph transformation and algebraic specification into model transformation. In *Proc. International Workshop on Algebraic Development Techniques (WADT 2010)*, volume 7137 of *LNCS*, pages 193–208. Springer, 2012.

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

[Lan98] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer, second edition, 1998.

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

[Man96] Maria Manzano. *Extensions of First-Order Logic*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1996.

[MP08] 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.

[NCMM09] Martin Nordio, Cristiano Calcagno, Peter Müller, and Bertrand Meyer. A sound and complete program logic for Eiffel. In *Proc. International Conference on Objects, Components, Models and Patterns (TOOLS EUROPE 2009)*, volume 33 of *Lecture Notes in Business Information Processing*, pages 195–214. Springer, 2009.

[Nip02] Tobias Nipkow. Hoare logics in Isabelle/HOL. In Helmut Schwichtenberg and Ralf Steinbrüggen, editors, *Proof and System-Reliability*, pages 341–367. Kluwer Academic Publishers, 2002.

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

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[OEP10] Fernando Orejas, Hartmut Ehrig, and Ulrike Prange. Reasoning with graph constraints. *Formal Aspects of Computing*, 22(3-4):385–422, 2010.

[Ore08] Fernando Orejas. Attributed graph constraints. In *Proc. International Conference on Graph Transformation (ICGT 2008)*, volume 5214 of *LNCS*, pages 274–288. Springer, 2008.

[ORY01] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Proc. Computer Science Logic (CSL 2001)*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001.

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

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

[Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[PHM99] Arnd Poetzsch-Heffter and Peter Müller. A programming logic for sequential Java. In *Proc. Programming Languages and Systems (ESOP 1999)*, volume 1576 of *LNCS*, pages 162–176. Springer, 1999.

[Plo04] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, 2004.

[Plu98] Detlef Plump. Termination of graph rewriting is undecidable. *Fundamenta Informaticae*, 33(2):201–209, 1998.

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

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

[Plu13] Detlef Plump. Computing by Graph Transformation. University of York, 2013. Course notes. `http://www-module.cs.york.ac.uk/grat/`.

[Pos12] Christopher M. Poskitt. Verification of graph programs. In *Proc. International Conference on Graph Transformation (ICGT 2012)*, volume 7562 of *LNCS*, pages 420–422. Springer, 2012.

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

[PP10b] 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.

[PP12] Christopher M. Poskitt and Detlef Plump. Hoare-style verification of graph programs. *Fundamenta Informaticae*, 118(1-2):135–175, 2012.

[PP13] Christopher M. Poskitt and Detlef Plump. Verifying total correctness of graph programs. In *Revised Selected Papers, Graph Computation Models (GCM 2012)*, volume 61 of *Electronic Communications of the EASST*, 2013.

[PS04] Detlef Plump and Sandra Steinert. Towards graph programs for graph algorithms. In *Proc. International Conference on Graph Transformation (ICGT 2004)*, volume 3256 of *LNCS*, pages 128–143. Springer, 2004.

[PST13] Christian Percebois, Martin Strecker, and Hanh Nhi Tran. Rule-level verification of graph transformations for invariants based on edges' transitive closure. In *Proc. International Conference on Software Engineering and Formal Methods (SEFM 2013)*, volume 8137 of *LNCS*, pages 106–121. Springer, 2013.

[PvE93] Rinus Plasmeijer and Marko van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.

[Rad10] Hendrik Radke. Correctness of graph programs relative to HR^+ conditions. In *Proc. International Conference on Graph Transformation (ICGT 2010)*, volume 6372 of *LNCS*, pages 410–412. Springer, 2010.

[Rad13] Hendrik Radke. HR^* graph conditions between counting monadic second-order and second-order graph formulas. In *Revised Selected Papers, Graph Computation Models (GCM 2012)*, volume 61 of *Electronic Communications of the EASST*, 2013.

[RD06] Arend Rensink and Dino Distefano. Abstract graph transformation. In *Proc. International Workshop on Software Verification and Validation (SVV 2005)*, volume 157 of *Electronic Notes in Theoretical Computer Scince*, pages 39–59. Elsevier, 2006.

[Ren03] Arend Rensink. The GROOVE simulator: A tool for state space generation. In *Proc. Applications of Graph Transformations with Industrial Relevance (AGTIVE 2003)*, volume 3062 of *LNCS*, pages 479–485. Springer, 2003.

[Ren04] Arend Rensink. Representing first-order logic using graphs. In *Proc. Graph Transformations (ICGT 2004)*, volume 3256 of *LNCS*, pages 187–190. Springer, 2004.

[Ren10] Arend Rensink. The edge of graph transformation - graphs for behavioural specification. In *Graph Transformations and Model-Driven Engineering*, volume 5765 of *LNCS*, pages 6–32. Springer, 2010.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 55–74, 2002.

[RN08] Stefan Rieger and Thomas Noll. Abstracting complex data structures by hyperedge replacement. In *Proc. Graph Transformations (ICGT 2008)*, volume 5214 of *LNCS*, pages 69–83. Springer, 2008.

[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 of *LNCS*, pages 226–241. Springer, 2004.

[RZ10] Arend Rensink and Eduardo Zambon. Neighbourhood abstraction in groove. In *Proc. International Workshop on Graph-Based Tools (GraBaTs 2010)*, volume 32 of *Electronic Communications of the EASST*, 2010.

[RZ12] Arend Rensink and Eduardo Zambon. Pattern-based graph abstraction. In *Proc. International Conference on Graph Transformation (ICGT 2012)*, volume 7562 of *LNCS*, pages 66–80. Springer, 2012.

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

[Str08] Martin Strecker. Modeling and verifying graph transformations in proof assistants. In *Proc. Computing with Terms and Graphs (TERMGRAPH 2007)*, volume 201, pages 135–148. Elsevier, 2008.

[SV03] Ákos Schmidt and Dániel Varró. CheckVML: A tool for model checking visual modeling languages. In *Proc. The Unified Modeling Language, Modeling Languages and Applications (UML 2003)*, 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 of *LNCS*, pages 446–453. Springer, 2004.

[TP12] Hanh Nhi Tran and Christian Percebois. Towards a rule-level verification framework for property-preserving graph transformations. In *Proc. ICST Workshop on Verification and Validation of Model Transformations (VOLT 2012)*, pages 946–953, 2012.

[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.

[vO01] David von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.

[VVGE+06] Dániel Varró, Szilvia Varró-Gyapay, Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. Termination analysis of model transformations by Petri nets. In *Proc. International Conference on Graph Transformation (ICGT 2006)*, volume 4178 of *LNCS*, pages 260–274. Springer, 2006.

[VVP02] Dániel Varró, Gergely Varró, and András Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, 44(2):205–227, 2002.

[Win93] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, Cambridge, MA, USA, 1993.

[ZR12] Eduardo Zambon and Arend Rensink. Graph subsumption in abstract state space exploration. In *Proc. Workshop on GRAPH Inspection and Traversal Engineering (GRAPHITE 2012)*, volume 99 of *Electronic Proceedings in Theoretical Computer Science*, pages 35–49, 2012.

[ZWL13] Liang Zhao, Shuling Wang, and Zhiming Liu. Graph-based object-oriented Hoare logic. In *Proc. Theories of Programming and Formal Methods: Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*, volume 8051 of *LNCS*, pages 374–393. Springer, 2013.