THE GRAPH PROGRAMMING LANGUAGE GP

Sandra Steinert

Submitted for the degree of Doctor of Philosophy

The University of York

Department of Computer Science

October 2007
Abstract

Graph transformation languages are declarative, rule-based languages that abstract from low-level representations of graphs and graph manipulations. Based on formal rewriting approaches, graph transformation languages have a strong formal underpinning which facilitates formal reasoning. However, graph transformation languages that are both expressive enough for elegantly solving complex graph problems and also have a complete formal semantics to facilitate reasoning about programs, are surprisingly rare. In this thesis, we introduce the graph programming language GP (for Graph Programs). GP is a practical graph transformation language with a simple syntax and complete formal semantics that facilitates formal reasoning. GP is a further development of the language presented in [HP01]. We introduce the basis of GP in form of conditional rule schemata [PS04]. The language is based on four core constructs whose semantics is defined in the style of Plotkin’s structural operational semantics [Plot04]. Practicality and suitability for verification is demonstrated on various case studies in the domain of graph algorithms. The language GP is extended with procedures and parameterised calls to form the language GP+. GP+ provides better structuring of programs and more programming comfort which is demonstrated by a case study on planarity testing.
# Contents

Abstract 3

List of Figures 9

Acknowledgements 13

Author’s Declaration 15

1 Introduction 17

2 Graph Transformation 21
   2.1 Preliminaries ........................................ 21
   2.2 Rules and Derivations ................................ 23
   2.3 Related Work ......................................... 28

3 GP Rule Schemata 31
   3.1 Syntax .............................................. 33
   3.2 Semantics ............................................ 40
   3.3 Uniqueness of Induced Rules ......................... 44
   3.4 Related Work ........................................ 48

4 The Language GP 51
   4.1 Syntax .............................................. 52
   4.2 Structural Operational Semantics .................... 61
      4.2.1 Configurations and Inference Rules ............. 61
      4.2.2 Definition of the Semantics .................... 63
      4.2.3 Some Program Equivalences ..................... 71
   4.3 Related Work ........................................ 73
      4.3.1 Rule-based Languages ............................ 73
      4.3.2 Graph Transformation Languages ................ 74
5 Small Case Studies

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Preliminaries</td>
<td>80</td>
</tr>
<tr>
<td>5.2 The GP Environment</td>
<td>82</td>
</tr>
<tr>
<td>5.3 Dijkstra’s Single-Source Shortest Path Algorithm</td>
<td>85</td>
</tr>
<tr>
<td>5.3.1 The GP Program Simple_Dijkstra</td>
<td>86</td>
</tr>
<tr>
<td>5.3.2 The GP Program Dijkstra</td>
<td>92</td>
</tr>
<tr>
<td>5.3.3 GP System Runs</td>
<td>97</td>
</tr>
<tr>
<td>5.4 Floyd-Warshall’s All-Pairs Shortest Path Algorithm</td>
<td>100</td>
</tr>
<tr>
<td>5.4.1 The GP Program Simple_Floyd_Warshall</td>
<td>101</td>
</tr>
<tr>
<td>5.4.2 The GP Program Floyd_Warshall</td>
<td>106</td>
</tr>
<tr>
<td>5.5 Vertex Colouring</td>
<td>116</td>
</tr>
<tr>
<td>5.6 Related Work</td>
<td>121</td>
</tr>
</tbody>
</table>

6 Adding Procedures: The Language GP+

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Syntax</td>
<td>124</td>
</tr>
<tr>
<td>6.2 Parameter Passing</td>
<td>136</td>
</tr>
<tr>
<td>6.3 SOS Inference Rules</td>
<td>141</td>
</tr>
<tr>
<td>6.4 Related Work</td>
<td>145</td>
</tr>
</tbody>
</table>

7 Case Study: Planarity Testing

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Preliminaries</td>
<td>150</td>
</tr>
<tr>
<td>7.2 The Auslander-Parter Planarity-Test Algorithm</td>
<td>155</td>
</tr>
<tr>
<td>7.3 The GP+ Program PlanarityTest</td>
<td>161</td>
</tr>
<tr>
<td>7.3.1 The Procedure CompCycle</td>
<td>168</td>
</tr>
<tr>
<td>7.3.2 The Procedure CompPieces</td>
<td>173</td>
</tr>
<tr>
<td>7.3.3 The Procedure SepCycle</td>
<td>178</td>
</tr>
<tr>
<td>7.3.4 The Procedure CompInter</td>
<td>184</td>
</tr>
<tr>
<td>7.3.5 The Procedure Bipartite</td>
<td>192</td>
</tr>
<tr>
<td>7.4 Related Work</td>
<td>196</td>
</tr>
</tbody>
</table>

8 Conclusions

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1 Contributions</td>
<td>199</td>
</tr>
<tr>
<td>8.2 Future Work</td>
<td>203</td>
</tr>
</tbody>
</table>

Bibliography                                                               | 207  |
Index 217
List of Figures

2.1 A graph ................................................................. 22
2.2 A rule ................................................................. 24
2.3 Short version of the rule in Fig. 2.2 ....................... 24
2.4 Example rule application ................................. 25
2.5 A pushout .............................................................. 26
2.6 A direct derivation ................................. 26
2.7 A conditional rule .............................................. 28

3.1 A conditional rule schema ........................................ 32
3.2 A conditional rule induced by the rule schema in Figure 3.1 32
3.3 A rule schema with tags ........................................ 37
3.4 A conditional rule induced by the rule schema in Figure 3.3 38
3.5 Input graph for the conditional rule schema in Fig. 3.1 ... 43
3.6 Induced conditional rules ........................................ 43
3.7 Application of a conditional rule schema ...................... 44

4.1 Example program MakeConnected ............................. 53
4.2a An example run of MakeConnected ............................ 55
4.2b An example run of MakeConnected (continued) .......... 56
4.3 Possible results for MakeConnected ........................... 57
4.4 Improved program of MakeConnected ........................ 58
4.5 The SOS inference rules ........................................... 64
4.6 Inference rules for derived constructs ........................ 66
4.7 Example program Undirected ..................................... 68
4.8 A “Yes”-run of Undirected ...................................... 69
4.9 A “No”-run of Undirected ...................................... 69
4.10 Example program with infinite set of output graphs ...... 71
5.1 The GP environment .................................. 83
5.2 The program Simple_Dijkstra .......................... 86
5.3 An example run of Simple_Dijkstra and Dijkstra ... 89
5.4 Input permitting an exponential run of Simple_Dijkstra 90
5.5 The program Dijkstra .................................. 93
5.6 The macro Prepare ..................................... 94
5.7 The program GenerateComplex ......................... 98
5.8 The program Simple_Floyd_Warshall .................. 102
5.9 The program Floyd_Warshall ........................... 108
5.10 The macro PrepareFW ................................. 109
5.11 A worst-case run of Floyd_Warshall ................. 110
5.12 The program VertexColouring ......................... 117
5.13 An example run of VertexColouring .................. 119
5.14 An output for a best-case run of VertexColouring ... 120

6.1 Flat block structure of a GP⁺ program ............... 125
6.2 Example program SpanningTree ....................... 126
6.3 An example run of SpanningTree ..................... 128
6.4 The SOS inference rules for GP⁺ ........................ 143

7.1 A graph that is not biconnected ...................... 150
7.2 A bipartite graph .................................... 151
7.3 An example graph for attachments ................... 151
7.4 The pieces of the graph in Figure 7.3 ............... 152
7.5 A non-separating cycle ................................ 153
7.6 A graph with interlacing pieces ...................... 154
7.7 Interlacement graph of the graph in Figure 7.6 .... 154
7.8 A planar graph ....................................... 155
7.9 A non-planar graph ................................... 156
7.10 The Auslander-Parter planarity-test algorithm .... 157
7.11a An example run of the Auslander-Parter algorithm .. 159
7.11b An example run of the A-P algorithm (continued) .. 160
7.11c An example run of the A-P algorithm (interlacement graphs) 160
7.12 A conditional rule schema ............................ 162
7.13  The rule schema of Figure 7.12 (Assumption 6) . . . . . . 162
7.14a The program PlanarityTest . . . . . . . . . . . . . . . . . 164
7.14b The program PlanarityTest (continued) . . . . . . . . . . 165
7.14c The program PlanarityTest (continued) . . . . . . . . . . 166
7.15a The procedure CompCycle . . . . . . . . . . . . . . . . . 169
7.15b The procedure DFS . . . . . . . . . . . . . . . . . . . . . . . 170
7.16  An example run of CompCycle . . . . . . . . . . . . . . . . 172
7.17a The procedure CompPieces . . . . . . . . . . . . . . . . . 174
7.17b The procedure TagPiece . . . . . . . . . . . . . . . . . . . 175
7.18  An example run of TagPiece . . . . . . . . . . . . . . . . . 177
7.19a The procedure SepCycle . . . . . . . . . . . . . . . . . . . 179
7.19b The procedure SepCycle (continued) . . . . . . . . . . . 180
7.20  Separating-cycle construction (piece with three attachments) 181
7.21  Separating-cycle construction (piece with two attachments) 182
7.22  An example run of SepCycle . . . . . . . . . . . . . . . . . 185
7.23a The procedure CompInter . . . . . . . . . . . . . . . . . . 187
7.23b The procedure CompInter (continued) . . . . . . . . . . 188
7.23c The procedure CompInter (continued) . . . . . . . . . . 189
7.23d The procedure CompInter (continued) . . . . . . . . . . 190
7.24  An example run of CompInter . . . . . . . . . . . . . . . . 191
7.25  The procedure Bipartite . . . . . . . . . . . . . . . . . . . . 193
7.26  An example run of Bipartite . . . . . . . . . . . . . . . . . 195
7.27  The Kuratowski graphs . . . . . . . . . . . . . . . . . . . . 196
Acknowledgements

I would like to thank my supervisor, Dr. Detlef Plump, for his extremely valuable comments on my work, his strong support, and his advice in times of struggle. We had brilliant discussions during the time of my PhD, which I will thoroughly miss.

I would also like to thank Colin Runciman for important comments after discussions and talks and the personal advice he has always given to me. The members of the Programming Languages and Systems group at The University of York for their support and lively discussions on my topic. Prof. Dr. Andy Schürr for his time and effort to explain the language PROGRES to me. Prof. Dr. Annegret Habel, who first introduced me to the graph transformation community.

Judith Isayenkova, Simon Giesecke, Jan Tobias Mühlberg and Rachael Madison have strongly supported me personally. I cannot thank them enough for their tea and in Judith’s case a lovely home to come back to in the final time of my PhD.

Additionally, I would like to thank my family for always being there for me.

Finally, I could not have completed this PhD without the love, patience and support of my partner, Thorsten Meyer. Thank you so much!
Author’s Declaration

Some of the material presented in this thesis has previously been published in [PS04]. Some of the material about the syntax and semantics of GP in Chapter 4 will be published in [PS07]. Except for the above cases and where stated, all of the work contained within this thesis represents the original contribution of the author.
Chapter 1

Introduction

Graphs are powerful and versatile structures used in many areas of computer science. Entity-relationship diagrams, class diagrams, software architecture models, object relations, pointer structures, control flow, system dependencies and networks are all described by some form of graph and these represent only a few examples. A rich application domain for the transformation of graph representations is the domain of graph algorithms. Graph algorithms such as shortest-path, maximum-flow, spanning-tree, cycle-detection and planarity-test algorithms compute or test properties on graphs by transforming graph structures or traversing through them. The result of such transformation should be verifiable. However, graph structures and their transformations are often implemented in low-level languages such as the C programming language [Sed92] and therefore graph algorithms are often difficult to implement, to comprehend and to verify.

Programming languages dealing with graph data structures in a simple and formally defined way are required. Here, graph transformation languages offer a solution. These are rule-based languages that abstract from low-level representations of graph structures and graph manipulations. Thus, they combine the strengths of graphs with declarative, rule-based languages. Based on formal graph rewriting approaches [Ehr79, EH97], graph transformation languages have a strong formal underpinning which facilitates formal reasoning.
Surprisingly, graph transformation languages that are both expressive enough for elegantly solving complex graph problems and also have a complete formal semantics to facilitate reasoning about programs, are rare. Practical languages such as PROGRES [SWZ99] and Fujaba [FNTZ98] are too complex to facilitate formal reasoning. Other languages like AGG [ERT99] are too simple to be of practical value or lack a complete formal semantics. Formal reasoning has only been successfully demonstrated in a case study in [KK99] for the language GRACE [KBK00], which is based on so-called transformation units [Kus00].

The language GP (for Graph Programs) that is introduced in this thesis, is a practical graph transformation language with a simple syntax and formal semantics that facilitates formal reasoning. Thus, GP combines practicality with simplicity of semantics.

GP is a further development of the minimal and computationally complete core language introduced in [HP01, HP02a]. A GP program consists of conditional rules-schema declarations [PS04] together with control sequences that guide the application of the rule schemata. Rule schemata are rules in the double-pushout approach [CMR+97] whose graphs are labelled with expressions over variables instead of constant values. A conditional rule schema is a rule schema with an additional predicate on graph morphisms that must hold true for the rule schema to be applicable. Semantically, conditional rule schemata represent sets of conditional double-pushout rules [CMR+97]. These are then applied in a non-deterministic fashion to input graphs. The control concepts defined for GP comprise the four core constructs rule-set calls, sequential composition, branching and iteration, and four derived constructs. The semantics of GP is defined in the style of Plotkin’s structural operational semantics [Plo04] which facilitates a direct implementation of the language.

The language GP is extendable in various ways. These extensions might, however, influence the possibility of formal reasoning about GP programs as the semantics might become too complex. The GP language presented here should be seen as a core for further development. In this thesis, we present
the language GP$^+$ as one such extension of GP. To enhance structuring and
reuse of program parts, GP is extended with procedures and parameterised
rule-schema calls. Thus, GP$^+$ provides procedural abstraction.

Case studies in the domain of graph algorithms are used in this thesis to
demonstrate the practicality of GP and GP$^+$ on the one hand and formal
reasoning about programs on the other hand. The case studies presented
in this thesis are Dijkstra’s single-source shortest-path algorithm [Eve89,
CLR00, Jun02], Floyd-Warshall’s all-pairs shortest-path algorithm [Eve89,
CLR00, Jun02], a non-deterministic vertex-colouring algorithm [Jun02] and
Auslander-Parter’s planarity-test algorithm [AP61, Gol63].

We recall basic notions of graph transformation in Chapter 2. The basis
of GP in the form of conditional rule schemata is introduced in Chapter
3. The syntax and structural operational semantics of the language GP is
introduced in Chapter 4. Case studies for GP are introduced in Chapter
5. The GP language extension GP$^+$, and the planarity-test case study
are introduced in Chapters 6 and 7. We complete this thesis with our
conclusions in Chapter 8.
Chapter 2

Graph Transformation

In this chapter, we define the graph-transformation approach underlying GP. Graph-transformation rules in GP are so-called conditional double-pushout rules with relabelling. These are rules in the double-pushout approach with relabelling as introduced in [HP02b] together with conditions. Semantically, these conditions are predicates on graph morphisms.

We review basic notions of graphs in Section 2.1 and introduce graph transformation in Section 2.2. We conclude the chapter with a brief comparison to other approaches in Section 2.3.

2.1 Preliminaries

A label alphabet is an arbitrary but fixed set $\mathcal{L}$.

Definition 1 (Graph) A partially labelled graph over $\mathcal{L}$ is a system $G = (V_G, E_G, s_G, t_G, l_G, m_G)$, where $V_G$ and $E_G$ are finite sets of nodes and edges, $s_G, t_G : E_G \rightarrow V_G$ are source and target functions for edges, $l_G : V_G \rightarrow \mathcal{L}$ is the partial node labelling function and $m_G : E_G \rightarrow \mathcal{L}$ is the (total) edge labelling function.\(^1\)

\(^1\)Given a partial function $f : A \rightarrow B$, the set $\text{Dom}(f) = \{x \in A \mid f(x) \text{ is defined}\}$ is the domain of $f$. We write $f(x) = \perp$ if $f(x)$ is undefined.
A graph is \textit{totally labelled} if $I_G$ is a total function. We write $\mathcal{G}(L)$ for the set of partially labelled graphs (or \textit{graphs} for short), and $\mathcal{G}^t(L)$ for the set of totally labelled graphs over $L$. We introduce graphs with partially labelled nodes and totally labelled edges since we introduce graph transformation with node relabelling only (see Section 2.2).

Graphs in the above sense are \textit{directed} as each edge comes with a source and a target node. In pictures, nodes are depicted as circles and edges are depicted as arrows with the arrowhead pointing to the target node. Labels of nodes are written inside the circle and labels of edges next to the arrow. The empty graph is depicted by $\emptyset$. For example, Figure 2.1 depicts the graph $G$ over $L = \{1, 2, 3, a, b\}$ defined by $G = (\{v_1, v_2, v_3, v_4\}, \{e_1, e_2\}, (s_G(e_1) = v_1, s_G(e_2) = v_2), (t_G(e_1) = v_2, t_G(e_2) = v_3), (l_G(v_1) = 1, l_G(v_2) = 1, l_G(v_3) = 1, l_G(v_4) = 2), (m_G(e_1) = a, m_G(e_2) = a))$.

![Figure 2.1: A graph](image)

**Definition 2 (Premorphism and Graph Morphism)** A \textit{premorphism} $g: G \to H$ between two graphs $G$ and $H$ consists of two functions $g_V: V_G \to V_H$ and $g_E: E_G \to E_H$ that preserve source and targets: $s_H \circ g_E = g_V \circ s_G$ and $t_H \circ g_E = g_V \circ t_G$, where $\circ$ denotes function composition. If $g$ also preserves labels in the sense that $m_H \circ g_E = m_G$ and $l_H(g(v)) = l_G(v)$ for all $v$ in $\text{Dom}(l_G)$ then $g$ is a \textit{graph morphism}.

A premorphism $g$ is \textit{injective} (\textit{surjective}) if $g_V$ and $g_E$ are injective (surjective), and it is an \textit{inclusion} if $g(x) = x$ for all nodes and edges $x$ in $G$. A graph morphism $G \to H$ is an \textit{isomorphism} if it is injective, surjective and satisfies $l_H(g(v)) = \bot$ for all nodes $v$ with $l_V(v) = \bot$. In this case $G$ and $H$ are \textit{isomorphic}, which is denoted by $G \cong H$. 
For example, consider the graph $H = (\{v_1, v_2, v_3, v_4\}, \{e_1, e_2\}, (s_H(e_1) = v_2, s_H(e_2) = v_3), (t_H(e_1) = v_3, t_H(e_2) = v_1), (l_H(v_1) = 1, l_H(v_2) = 1, l_H(v_3) = 1, l_H(v_4) = 2), (m_H(e_1) = a, m_H(e_2) = a))$. Then the graphs $G$ depicted in Figure 2.1 and $H$ are isomorphic as one can easily find an isomorphism from $G$ to $H$ and indeed the pictures of $G$ and $H$ are identical. Note that pictures of graphs represent so-called abstract graphs, i.e. isomorphism classes of graphs, where graphs as defined above are concrete graphs.

### 2.2 Rules and Derivations

In this section, we explain the model of graph transformation underlying GP, the so-called double-pushout approach with relabelling [HP02b]. Graphs are transformed by applying graph transformation rules locally.

Traditionally, the double-pushout approach is being considered without relabelling. A rule in [Ehr79, CMR+97] consists of three totally labelled graphs $L, K$ and $R$. The idea is to match $L$ to a subgraph of an input graph $G$, delete from that subgraph all items in $L$ that are not in $K$ and add to the result all items in $R$ that are not in $K$. A rule may not be applied to a subgraph if it deletes nodes whose incident edges are not deleted by the rule (dangling edge condition).

As $L, K$ and $R$ in [Ehr79] are totally labelled, changing node labels by transformation steps is difficult in this approach. Nodes have to be deleted and reinserted to change their label. Deletion might not be possible in all matching situations because of dangling edges. Thus, double-pushout rules with relabelling were introduced in [HP02b], allowing changes of node labels by using rules with a partially labelled interface graph $K$.

We tailor this relabelling approach to GP by relabelling nodes only — it is always possible to delete and reinsert edges in an arbitrary context —, not allowing node-merging rules and defining $L$ and $R$ as totally labelled graphs.
**Definition 3 (Rule)** A rule \( r = (L \leftarrow K \rightarrow R) \) consists of two graph morphisms \( K \rightarrow L \) and \( K \rightarrow R \) that are inclusions, where \( L \) and \( R \) are totally labelled. We call \( L \) the *left-hand side*, \( R \) the *right-hand side* and \( K \) the *interface* of \( r \).

In pictures of rules, the graph morphisms are indicated by small numbers attached to nodes. By Definition 3, \( K \rightarrow L \) and \( K \rightarrow R \) are inclusions and thus all nodes in \( K \) are also in \( L \) and \( R \). With the convention that the interface \( K \) always consists of the numbered nodes in \( L \) and \( R \), we can omit \( K \) in such pictures.

Figure 2.2 shows a rule according to Definition 3. The rule in Figure 2.3 is the short version of the same rule. The rule relabels node 1 from label 1 to 2, deletes the unnumbered node, removes the edge labelled by "a" and inserts a loop at node 2 with label "b". Note that the numbers attached to nodes indicate the graph morphisms and are used as node identifiers for this purpose.

![Figure 2.2: A rule](image)

![Figure 2.3: Short version of the rule in Fig. 2.2](image)

Before we introduce rule application formally, we give an example of the application of the rule \( r \) in Figure 2.2 to the graph \( G \) in Figure 2.1, see Figure 2.4. The first step of rule application is to find a graph morphism from \( L \) to \( G \). In our example, a possible match is the highlighted subgraph in \( G \). Next, all items in \( L \) that are not in \( K \) are deleted in that subgraph.
resulting in a new graph $G'$ where the edge and the 2-labelled node are deleted and nodes 1 and 2 are made unlabelled (see middle graph at the bottom). Last, all items in $R$ that are not in $K$ are added to $G'$ resulting in the new graph (right graph at the bottom) obtained from $G'$ by labelling node 1 with 2, node 2 with 1 and inserting a b-labelled loop at node 2.

![Diagram](image)

**Figure 2.4:** Example rule application

Figure 2.4 shows that a rule application forms two diagrams of graph morphisms. These are so-called *pushout* diagrams which are now formally introduced.

**Definition 4 (Pushout and Pullback)** A diagram of graph morphisms as in Figure 2.5 is a *pushout* if (i) $K \rightarrow R \rightarrow H = K \rightarrow D \rightarrow H$ and (ii) for every pair of graph morphisms $\langle R \rightarrow H', D \rightarrow H' \rangle$ with $K \rightarrow R \rightarrow H' = K \rightarrow D \rightarrow H'$ there is a unique morphism $H \rightarrow H'$ such that $R \rightarrow H' = R \rightarrow H \rightarrow H'$ and $D \rightarrow H' = D \rightarrow H \rightarrow H'$. The diagram in Figure 2.5 is a *pullback* if property (i) holds and if for every pair of graph morphisms $\langle K' \rightarrow R, K' \rightarrow D \rangle$ with $K' \rightarrow R \rightarrow H = K' \rightarrow D \rightarrow H$, there is a unique morphism $K' \rightarrow K$ such that $K' \rightarrow R = K' \rightarrow K \rightarrow R$ and $K' \rightarrow D = K' \rightarrow K \rightarrow D$. A pushout is natural if it is simultaneously a pullback.

For the construction of pushouts on partially labelled graphs see [HP02b].
Figure 2.5: A pushout

Definition 5 (Direct Derivation) Let \( G \) and \( H \) be graphs in \( \mathcal{G}(L) \) and \( r = (L \leftarrow K \rightarrow R) \) a rule. A direct derivation from \( G \) to \( H \) by \( r \) consists of two natural pushouts as in Figure 2.6, where \( g: L \rightarrow G \) is injective.

Figure 2.6: A direct derivation

We write \( G \Rightarrow_{r,g} H \) or just \( G \Rightarrow_r H \) if there exists a direct derivation as in Definition 5. If \( \mathcal{R} \) is a set of rules, then \( G \Rightarrow_{\mathcal{R}} H \) means that there is some \( r \) in \( \mathcal{R} \) such that \( G \Rightarrow_r H \).

Definition 6 (Match) Given a rule \( r = (L \leftarrow K \rightarrow R) \) and a graph \( G \) in \( \mathcal{G}(L) \), an injective graph morphism \( g: L \rightarrow G \) is a match for \( r \) if it satisfies the dangling condition: no node in \( g(L) \) is incident to an edge in \( G - g(L) \).

As we allow node relabelling in our approach, the interface graph \( K \) in a rule may be partially labelled. Therefore, the usual category of graphs does not apply. The condition is required to ensure the existence and uniqueness of natural pushout complements for rules defined on partially labelled graphs.

In [HP02b] it is shown that, given a rule \( r \) and an injective morphism \( g: L \rightarrow G \), there exists a direct derivation as in Figure 2.6 if and only if \( g \) is a match for \( r \). Moreover, in this case \( D \) and \( H \) are determined uniquely.
up to isomorphism. The two natural pushouts (1) and (2) of Figure 2.6 are constructed from $G$ as follows:

1. Remove all nodes and edges in $g(L) - g(K)$, obtaining a subgraph $D'$. 

2. Turn $D'$ into $D$ by defining $l_D(g_V(v)) = \bot$ for each $v \in (V_K - \text{Dom}(l_K))$. 

3. Add disjointly to $D$ all nodes and edges from $R - K$, keeping their labels. For $e \in E_R - E_K$, $s_H(e)$ is $s_R(e)$ if $s_R(e) \in V_R - V_K$, otherwise $g_V(s_R(e))$. Targets are defined analogously. 

4. For each node $v$ in $K$ with $l_K(v) = \bot$, $l_H(g_V(v))$ becomes $l_R(v)$. The resulting graph is $H$.

An example of the two natural pushouts of a direct derivation is given in Figure 2.4. Note that finding a match in general is non-deterministic. Another possible match for the rule in Figure 2.2 in the graph $G$ of Figure 2.1 is the subgraph of $G$ that does not include the top-left node and its incident edge. Note also that the rule is applicable once more to the resulting graph of Figure 2.4.

In Chapter 3, we introduce conditional rule schemata as the basic components of GP. To interpret these schemata semantically, we equip rules with predicates that restrict sets of matches.

**Definition 7 (Conditional Rule)** A conditional rule $q = (r, P)$ consists of a rule $r$ and a predicate $P$ on graph morphisms that is invariant under isomorphic codomains: for a morphism $g: L \rightarrow G$ and an isomorphism $i: G \rightarrow G'$, we have $P(g)$ if and only if $P(i \circ g)$.

Given totally labelled graphs $G$ and $H$ and a match $g: L \rightarrow G$ for $r$, we write $G \Rightarrow_{q, g} H$ if $G \Rightarrow_{r, g} H$ and $P(g)$. As in the case of unconditional rules, $G \Rightarrow q H$ means that there exists a match $g$ such that $G \Rightarrow_{q, g} H$. For a set of conditional rules $\mathcal{R}$, we write $G \Rightarrow_{\mathcal{R}} H$ if there is some $q$ in $\mathcal{R}$ such that $G \Rightarrow q H$. 

An example of a conditional rule \((r, P)\) is shown in Figure 2.7, where \(r\) is the rule of Figure 2.2 and \(P\) is the predicate \(\text{not edge}(1, 1)\). Here, we anticipate a form of condition that is introduced for GP in Section 3.1. The predicate \(\text{not edge}(1, 1)\) holds true for a graph morphism \(g\) if and only if \(g\) is a graph morphism from \(L\) to \(G\) such that there is no edge from \(g_i(1)\) to \(g_i(1)\). \((r, P)\) is applicable as in Figure 2.4 but not to the result of Figure 2.4 as the predicate \(\text{not edge}(1, 1)\) does not hold true for the only possible match in that result.

\[
\begin{array}{c}
\begin{array}{c}
1 \quad 2 \\
\downarrow & \quad \downarrow \\
1 & 2
\end{array}
\quad \Longleftarrow \quad
\begin{array}{c}
\circ & \circ \\
\end{array}
\quad \longrightarrow \quad
\begin{array}{c}
\circ & \circ \\
\end{array}
\quad \begin{array}{c}
2 \\
\uparrow \\
1
\end{array}, \quad \text{not edge}(1, 1)
\end{array}
\]

**Figure 2.7:** A conditional rule

The nature of conditions will be developed in Section 3.1 (rule-schema conditions) and Section 3.2 (Definition 10 and Definition 11) with the introduction of conditional rule schemata. In this thesis, atomic conditions will be of two kinds. They either relate numeric labels or require or forbid the existence of an edge (see the example above).

### 2.3 Related Work

The chosen graph-transformation approach is one of the two well-known algebraic approaches in graph transformation, the other one being the so-called *single-pushout approach* \([\text{EHK}^+97]\). We have chosen the double-pushout approach as the formal basis of our language as most theoretic results were obtained for this approach \([\text{EPS73, Ehr79, CMR}^+97, \text{HMP}01]\). The single-pushout approach \([\text{EHK}^+97]\) differs from the double-pushout approach in that there is no interface graph \(K\), the morphism from \(L\) to \(R\) is partial and the dangling condition is omitted. Dangling edges in the single-pushout approach are deleted together with the deleted node. For a comparison of the double- and the single-pushout approach see \([\text{EHK}^+97]\).
Another graph-transformation approach is the algorithmic approach introduced in [Nag79].

Our relabelling approach restricts the approach presented in [HP02b]. We require totally labelled graphs in the left- and right-hand side of a rule. The use of edge-relabelling is not necessary as edges can be deleted and reinserted in an arbitrary context. Thus, we restrict relabelling to nodes. The concept of merging items in a rule is not used in any of our case studies. Therefore, we differ from [HP02b] and [PS04] in that we allow only injective rules.

Negative conditions on the application of rules are not naturally expressible by rules in the double-pushout approach. Especially expressing structural negative application conditions such as the non-existence of edges by general rule applications is cumbersome. Our concept of a conditional rule is similar to that of [HHT96] where rules are equipped with two sets of morphisms (representing positive and negative application conditions, respectively). Because [HHT96] is based on the single-pushout approach, admissible morphisms need not satisfy the dangling condition.
Chapter 3

GP Rule Schemata

In this chapter, we introduce the main building blocks of GP programs, namely conditional rule schemata [PS04]. Conditional rule schemata represent sets of conditional rules in the sense of Definition 7. Where the graphs in conditional rules are labelled with “constant” values, graphs in rule schemata are labelled with expressions over variables. Semantically, these are evaluated to values by assigning values to variables and evaluating the expressions.

This concept is crucial to express graph algorithms conveniently. Input graphs in this domain are typically labelled with numbers. Graph algorithms expressed by rules need to perform computations on labels. This is difficult to implement with conditional rules in the sense of Definition 7 as each such rule is labelled with constant values. To transform differently labelled subgraphs that share some property of labels, a (possibly) infinite set of rules each representing the same structure but with different labelling is needed. Our solution to this problem is to represent sets of conditional rules by conditional rule schemata.

In this thesis, we forbear from introducing general rule schemata that use arbitrary expressions in the left and right graph as presented in [PS04]. Instead, we introduce conditional rule schemata (see “deterministic rule schemata” in [PS04]) which admit an efficient implementation (see Section 3.3). Conditional rule schemata restrict the form of expressions in the left
graph and the use of variables. We prove in Section 3.3 that rule schemata of this form, for a given premorphism, induce at most one conditional rule such that the premorphism is a match satisfying the condition. For simplicity of presentation, we directly introduce the GP syntax for rule schemata instead of first introducing the general form given in [PS04].

Figure 3.1 shows an example of the graphical declaration of a conditional rule schema in GP. It consists of the identifier \texttt{Reduce} and a list of formal parameters of type integer, the left- and right-hand graphs of the schema which are labelled with expressions over the formal parameters, the node names 1 and 2 determining the graph morphisms between left-hand side, interface and right-hand side, and the condition in form of a boolean expression following the keyword \textit{where}.

\[
\text{Reduce}(x,y,z:\text{int}) = \\
\begin{tikzpicture}[scale=1]
  \node (x) at (0,0) {x};
  \node (y) at (1,0) {y};
  \node (z) at (2,0) {z};
  \draw (x) to (y);
  \node (x2) at (0,1) {x};
  \node (y2) at (1,1) {y};
  \node (z2) at (2,1) {x+y};
  \draw (x2) to (y2);
\end{tikzpicture}
\]

\textit{where } \(x + y < z\) \textit{ and not edge}(2,1)

\textbf{Figure 3.1: A conditional rule schema}

One of the conditional rules induced by the rule schema in Figure 3.1 is shown in Figure 3.2. Here, the integer constants 1, 2, 4 have replaced the parameters \(x, y, z\) of the rule schema and the predicate \textit{not edge}(2,1) holds true for a graph morphism \(g\) from the left-hand side to the input graph if there exists no edge from node \(g_V(2)\) to node \(g_V(1)\).

\[
\begin{pmatrix}
1 & 2 & 4 \\
1 & 2 & 3 \\
\end{pmatrix}
\Rightarrow
\begin{pmatrix}
1 & 2 & 3 \\
\text{, not edge}(2,1) \\
\end{pmatrix}
\]

\textbf{Figure 3.2: A conditional rule induced by the rule schema in Figure 3.1}

Rule schemata offer two main advantages over ordinary rules. First, a (possibly infinite) set of rules is represented by one rule schema and with
this, programming becomes simpler. Second, using expressions in labels makes computations on labels possible which is a crucial concept when implementing graph algorithms (see also Chapter 5).

In the following, we give a textual syntax for conditional rule schema declarations in Section 3.1 and define the semantics of conditional rule schemata in Section 3.2. In Section 3.3, we show that conditional rule schemata admit an efficient implementation. A brief comparison to other approaches is given in Section 3.4.

3.1 Syntax

To give a complete formal semantics for GP, we need to give a syntax definition of the language first. In this section, we give a syntax and context conditions for the textual components of ruleschema declarations, including node and edge labels. We also give a syntax for the textual definition of the involved graphs. However, for better readability, we use a visual representation of rule schemata — also offered by the GP environment — throughout this thesis.

The following grammar is written in a variant of extended Backus-Naur Form (EBNF). We use three extensions to Backus-Naur Form. These are:

- optional parts in the right-hand side denoted by square brackets [...],
- parts arbitrarily often repeated or left out denoted by curly brackets {...},
- terminal words consisting of more than one symbol are written in type- writer font; terminal symbols and mathematical operators consisting of more than one symbol are enclosed in single quotation marks.

Given a nonterminal $N$ in the following grammar, we also refer to $N$ — by abuse of notation — as the set of terminal strings derivable from $N$ and call this set the category $N$. 
We define context conditions (if necessary) together with the grammar for each of the components of a rule-schema declaration. Context conditions, also called static semantics (see [Plo04]), define conditions or restrictions on the program text that can be statically verified but not necessarily expressed in the context-free syntax description.

\section*{Rule-schema Declaration}

\begin{verbatim}
RuleSchemaDecl ::= RuleId ['(' ParSection {',' ParSection ')']} '='
                   Graphs [Cond]
ParSection ::= VarId {',' VarId} ':' Type
Type ::= int | string
VarId ::= Identifier
RuleId ::= Identifier
Identifier ::= Letter {Letter | Digit}
Letter ::= a | ... | z | A | ... | Z
Digit ::= 0 | ... | 9
\end{verbatim}

Context Conditions:

1. All variable identifiers (VarId) in the parameter section listed with the rule schema identifier (RuleId) must be distinct.

2. All identifiers must be distinct from the reserved words of GP. These are: \textit{int}, \textit{string}, \textit{where}, \textit{edge}, \textit{not}, \textit{and}, \textit{or}, \textit{impl}, \textit{main}, \textit{if}, \textit{then}, \textit{else}, \textit{try}, \textit{while}, \textit{do}, \textit{skip}, and \textit{fail}.

A variable is a placeholder for either an integer value or a character string. A rule schema may be defined without any parameters and then corresponds to an ordinary conditional rule.
3.1. Syntax

**Textual Graph Representation**

Graphs  ::=  '\{\{LeftGraph\}\} =⇒ \{\{RightGraph\}\}'
LeftGraph ::= LeftNodes ['|'] LeftEdges
RightGraph ::= RightNodes ['|'] RightEdges
LeftNodes ::= [Node '::' LeftLabel {',': Node '::' LeftLabel}]
LeftEdges ::= '(' Node '::', Node '::', LeftLabel ')' {',', '(' Node '::', Node '::', LeftLabel ')'}
RightNodes ::= [Node '::' RightLabel {',': Node '::' RightLabel}]
RightEdges ::= '(' Node '::', Node '::', RightLabel ')' {',', '(' Node '::', Node '::', RightLabel ')'}
Node ::= Num
Num ::= Digit {Digit}

Context Conditions:

1. The node numbers (Node) occurring in the list of left nodes (LeftNodes) must be pairwise distinct. The same applies to the node numbers in the list of right nodes (RightNodes).

2. The node numbers occurring in the declaration of edges of the left graph (LeftEdges) must occur in the list declaring left nodes (LeftNodes).

3. The node numbers occurring in the declaration of edges of the right graph (RightEdges) must occur in the list declaring right nodes (RightNodes).

LeftGraph and RightGraph define the representations of the left and right graph in a conditional rule schema. The number of nodes in the left and right graph is given by the length of the list LeftNodes or RightNodes, respectively. Each node is declared together with its label. The number of edges in the left and right graph is given by the length of the list LeftEdges or
RightEdges, respectively. Each entry of the lists LeftEdges and RightEdges defines the source node (first position), the target node (second position) and the label (third position) of an edge.

**Node and Edge Labels**

\[
\begin{align*}
\text{LeftLabel} & \ ::= \ \text{SimpleExp} \ ['_' \ \text{LeftLabel}] \\
\text{RightLabel} & \ ::= \ \text{Exp} \ ['_' \ \text{RightLabel}] \\
\text{SimpleExp} & \ ::= \ ['-' \ \text{Num} | \ \text{String} | \ \text{VarId} \\
\text{Exp} & \ ::= \ \text{SimpleExp} \ | \ \text{Exp} \ \text{ArithOp} \ \text{Exp} \ | \ '(' \ \text{Exp} \ ')'
\end{align*}
\]

\[
\begin{align*}
\text{ArithOp} & \ ::= \ '+' \ | \ '-' \ | \ '*' \ | \ '/'
\text{String} & \ ::= \ '""' \ \{\text{Char}\} '""
\text{Char} & \ ::= \ \text{Letter} \ | \ \text{Digit} \ | \ '.' \ | \ ';' \ | \ ':' \ | \ '?' \ | \ '?' \ | \ '(' \ | \ ')' \ | \ '{' \ | \ '}' \ | \ '[' \ | \ ']' \ | \ '{' \ | \ '}' \ | \ '[' \ | \ ']' \ | \ '=' \ | \ '+' \ | \ '-' \ | \ '#' \ | \ '@' \ | \ '$' \ | \ '$' \ | \ '\' \ | \ '<' \ | \ '>' \ | \ '<' \ | \ '~'
\end{align*}
\]

**Context Conditions:**

1. All variable identifiers occurring in expressions (Exp) must have been declared with the rule schema identifier.

2. Each variable identifier occurring in labels of category RightLabel must also occur in some label of category LeftLabel of the same rule-schema declaration.

3. Numerals ("-" Num) have type \textit{int}. Character strings (String) have type \texttt{string}. The type of a variable identifier is the type assigned to that identifier in the parameter section (ParSection) of the rule schema declaration. For an expression of the form \( e_1 \ \text{op} \ e_2 \) where \text{op} is an arithmetic operator (ArithOp), \( e_1 \) and \( e_2 \) must be expressions of type \textit{int}. The overall expression has type \textit{int}.

The categories LeftLabel and RightLabel specify what labels may occur in the left and right graph of a rule schema. Labels in the left graph must be
3.1 Syntax

simple expressions because their values at execution time are determined by graph matching. All variable identifiers in the right graph must also occur in the left graph so that all expressions in the right graph can be uniquely evaluated through graph matching.

Node and edge labels can be sequences of expressions. Figure 3.3 shows the tagged version of the rule schema in Figure 3.1. Graphs in the tagged version can be labelled with sequences of expressions delimited by the underscore symbol ‘_’. Note that the sequence \( l_{t_1\, t_2\, \ldots\, t_n} \) is a label, the expressions \( t_1, \ldots, t_n \) are called tags. By abuse of notation, we also call the first component \( l \) of \( l_{t_1\, t_2\, \ldots\, t_n} \) a label. In Figure 3.3, \( j, y, k \) are labels and \( x, z \) are tags. Tags provide a way to distinguish items (nodes and edges) with the same label and to store extra information in the labels. In Figure 3.3, the original labelling of the graph is preserved by tagging. Where each application of the rule schema in Figure 3.1 destroys an original node label \((j, k)\), an application of the rule schema in Figure 3.3 does only add extra information to the labels in form of tags. Examples of the application of the tagging concept are presented in Chapter 5.

\[
\text{Reduce}(j, k, x, y, z : \text{int}) =
\]

\[
\begin{array}{c}
\text{Figure 3.3: A rule schema with tags}
\end{array}
\]

\[
\text{where } x + y < z \text{ and not edge}(2, 1)
\]

One of the conditional rules induced by the rule schema in Figure 3.3 is shown in Figure 3.4. Here, the integer constants 1, 4, 1, 2, 4 have replaced the parameters \( j, k, x, y, z \) of the rule schema and the predicate \( \text{not edge}(2, 1) \) holds true for a graph morphism \( g \) from the left-hand side to the input graph if there exists no edge from node \( g_\nu(2) \) to node \( g_\nu(1) \). Note that the original node label 4 of the node numbered 2 is preserved by the induced conditional rule.
\[
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
1 \quad 2
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
4 \quad 4
\end{array}
\end{array}
\end{array}
\end{array}
\Rightarrow
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
1 \quad 2
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
4 \quad 3
\end{array}
\end{array}
\end{array}, \text{ not } \text{edge}(2,1)
\end{array}
\]

Figure 3.4: A conditional rule induced by the rule schema in Figure 3.3

Rule-schema Condition

\[
\begin{align*}
\text{Cond} & \ ::= \ \text{where } \text{BoolExp} \\
\text{BoolExp} & \ ::= \ \text{edge} \ (\text{Node}, \text{Node})' \\
& \quad | \ \text{Exp RelOp Exp} \\
& \quad | \ \text{not BoolExp} \\
& \quad | \ \text{BoolExp RelOp BoolExp} \\
& \quad | \ '('. \text{BoolExp} ')'
\end{align*}
\]

\[
\begin{align*}
\text{RelOp} & \ ::= \ '=' | '\neq' | '>' | '<' | '>=' | '<= '
\end{align*}
\]

\[
\begin{align*}
\text{BoolOp} & \ ::= \ \text{and} | \ \text{or} | \ \text{impl}
\end{align*}
\]

Context Conditions:

1. Each variable identifier occurring in a condition (Cond) must also occur in some label of category LeftLabel of the same rule-schema declaration.

2. For a boolean expression (BoolExp) of the form \(e_1 \text{ op } e_2\), where \(\text{op}\) is a relational operator (RelOp), \(e_1\) and \(e_2\) must be expressions of type \(\text{int}\).

The predicate \text{edge} provides a way to require the (non-)existence of an edge between two nodes in the graph to which the rule schema is applied. For example, the expression \text{not edge}(2,1) in the condition of Figure 3.1 forbids an edge from node 2 to node 1 when the left graph is matched. Boolean expressions over variable identifiers of the left graph restrict possible matches. For example, the condition \(x + y < z\) in Figure 3.1 restricts the possible
matches to those where the label of node 1 plus the label of the edge is smaller than the label of node 2.

**Operator Precedence**

The evaluation order of operators in ambiguous expressions is specified by the following four-level precedence, going from highest to lowest:

1. not

2. 
   
   3. 
   
   4. 

Operators on the same level associate to the left.

Examples of operator precedences are the following:

1 + 2 * 3 is equivalent to 1 + (2 * 3).

not $x = 2$ or edge(1, 2) is equivalent to (not $x = 2$) or edge(1, 2).

$x = 2$ or $x = 3$ impl $x \leq y$ is equivalent to ($x = 2$ or $x = 3$) impl $x \leq y$.

**Example 1 (Textual Rule Schema Declaration)** The textual representation of the conditional rule schema declaration in Figure 3.1 is the following:

\[
\text{Reduce}(x, y, z: \text{int}) = \{1 : x, 2 : z \mid (1, 2, y)\} \Rightarrow \{1 : x, 2 : x + y \mid (1, 2, y)\}
\]

where $x + y < z$ and not edge(1, 2)

Here, $x$ and $z$ are the labels of nodes 1 and 2 in the left graph of Figure 3.1, (1, 2, y) defines the edge with label y from node 1 to node 2, $x$ and $x + y$ are the labels of nodes 1 and 2 in the right graph and (1, 2, y) defines the edge with label y from node 1 to node 2 in the right graph. □
3.2 Semantics

The semantics of a conditional rule schema declaration is a set of conditional rules in the sense of Definition 7. Conditional rules are obtained from the rule schema declaration by interpreting the graph parts of the declaration as graphs in the sense of Definition 1, assigning values to all variables, evaluating all expressions and interpreting the condition as a predicate on graph morphisms. We next formally define this semantics.

For simplicity of notation, we assume the following.

Assumption 1 For the rest of this thesis, let $D$ be an arbitrary but fixed declaration of a conditional rule-schema. Let $L$ and $R$ be the components of category LeftGraph and RightGraph, and $c$ the condition (of category Condl) occurring in $D$.

Syntactic declarations of graphs are semantically interpreted as graphs according to Definition 1. We next define the semantics of the graph declarations $L$ and $R$.

Definition 8 (Semantics of $L$ and $R$) The semantics $\text{Graph}(L)$ of $L$ is the following graph over $\text{Exp}$:

- Let $n_1:a_1,\ldots,n_k:a_k$ be the list in $L$ declaring the nodes (LeftNodes).
  Then $V_{\text{Graph}(L)} = \{n_1,\ldots,n_k\}$ and $I_{\text{Graph}(L)}:V_{\text{Graph}(L)} \rightarrow \text{Exp}$ is defined by $I_{\text{Graph}(L)}(n_i) = a_i$ for $i = 1,\ldots,k$.

- Let $(u_1,v_1,b_1),\ldots,(u_p,v_p,b_p)$ be the list in $L$ declaring the edges (LeftEdges). Then $E_{\text{Graph}(L)} = \{1,\ldots,p\}$ and $m_{\text{Graph}(L)}:E_{\text{Graph}(L)} \rightarrow \text{Exp}$ and $s_{\text{Graph}(L)}:t_{\text{Graph}(L)}:E_{\text{Graph}(L)} \rightarrow V_{\text{Graph}(L)}$ are defined by $m_{\text{Graph}(L)}(i) = b_i$, $s_{\text{Graph}(L)}(i) = u_i$ and $t_{\text{Graph}(L)}(i) = v_i$, for $i = 1,\ldots,p$.

The semantics $\text{Graph}(R)$ of $R$ is obtained analogously.

We write $\text{Var}(D), \text{Var}(L), \text{Var}(R), \text{Var}(c)$ for the set of identifiers of category VarId occurring in $D, L, R$ and $c$, respectively. Given $x$ in $\text{Var}(X)$ where
$X \in \{D, L, R, c\}$, type($x$) denotes the unique type (of category Type) that is associated with $x$ (as determined by the typing rules in context conditions).

Let $\mathbb{Z}$ be the set of integers. We fix the label alphabet $\mathcal{L} = (\mathbb{Z} \cup \text{Char}^*)^+$ consisting of all nonempty sequences made up from integers and character strings. (The reason for using sequences is the possibility of tagging, see Section 3.1.) An assignment is a mapping $\alpha: \text{Var}(D) \to (\mathbb{Z} \cup \text{Char}^*)$ such that for each $x$ in $\text{Var}(D)$, type($x$) = int implies $\alpha(x) \in \mathbb{Z}$, and type($x$) = string implies $\alpha(x) \in \text{Char}^*$.

**Definition 9 (Evaluation of Labels)** Let $\alpha$ be an assignment. For every label $l \in \text{RightLabel}$ occurring in $D$, its value $l^\alpha$ is inductively defined as follows:

- For $n \in \text{Num}$, $n^\alpha$ and $(-n)^\alpha$ are the integers represented by $n$ and $-n$.
- For $c_1, \ldots, c_n \in \text{Char}$, $n \geq 0$, $(c_1 \ldots c_n)^\alpha = c_1 \ldots c_n$.
- For $l \in \text{VarId}$, $l^\alpha = \alpha(l)$.
- For $e_1, e_2 \in \text{Exp}$ and $\otimes \in \text{ArithOp}$, $(e_1 \otimes e_2)^\alpha = e_1^\alpha \otimes_{\mathbb{Z}} e_2^\alpha$ where $\otimes_{\mathbb{Z}}$ is the integer operation represented by $\otimes$.
- For $e \in \text{Exp}$, $((e))^\alpha = e^\alpha$.
- For $e \in \text{Exp}$ and $m \in \text{RightLabel}$, $(e \_ m)^\alpha = e^\alpha m^\alpha$.

Note that this definition covers all labels since $\text{LeftLabel} \subseteq \text{RightLabel}$. Different from the evaluation of labels, the value of the condition $c$ depends not only on an assignment but also on a graph morphism if $c$ contains the predicate edge.

**Definition 10 (Evaluation of Boolean Expressions)** Let $\alpha$ be an assignment, let $\text{Graph}(L)^\alpha$ be obtained from the left graph $\text{Graph}(L)$ by replacing each label $l$ in $\text{Graph}(L)$ with $l^\alpha$, and let $g: \text{Graph}(L)^\alpha \to G$ be a graph morphism with $G \in \mathcal{G}^5(\mathcal{L})$. Then for every boolean expression $b \in \text{BoolExp}$ occurring in $c$, its value $b^\alpha \_ g$ in $\mathbb{B} = \{tt, ff\}$ is inductively defined as follows:
• For \( v, w \in \text{Node} \),
\[
\text{edge}(v, w)^{\alpha, g} = \begin{cases} 
\text{tt} & \text{if there is an edge from } g_V(v) \text{ to } g_V(w), \\
\text{ff} & \text{otherwise.} 
\end{cases}
\]

• For \( e_1, e_2 \in \text{Exp} \) and \( \otimes \in \text{RelOp} \), \((e_1 \otimes e_2)^{\alpha, g} = e_1^\alpha \otimes_Z e_2^\alpha \) where \( \otimes_Z \) is the relation on integers represented by \( \otimes \).

• For \( b_1 \in \text{BoolExp} \), \((b_1)^{\alpha, g} = b_1^\alpha \).

• For \( b_1 \in \text{BoolExp} \), \((\text{not } b_1)^{\alpha, g} \) is the negation of \( b_1^\alpha \).

• For \( b_1, b_2 \in \text{BoolExp} \) and \( \otimes \in \text{BoolOp} \), \((b_1 \otimes b_2)^{\alpha, g} = b_1^\alpha \otimes_B b_2^\alpha \)
where \( \otimes_B \) is the operation on \( \mathbb{B} \) represented by \( \otimes \).

We now define the semantics of the rule-schema declaration \( D \): the declaration represents the set of all conditional rules that are obtained by evaluating the graph labels and the condition according to some assignment.

**Definition 11 (Induced Conditional Rule)** Given an assignment \( \alpha \),
the conditional rule \((\text{Graph}(L)^{\alpha} \leftarrow K \rightarrow \text{Graph}(R)^{\alpha}, P^\alpha)\) induced by \( D \) and \( \alpha \) is given as follows:

• \( \text{Graph}(L) \) and \( \text{Graph}(R) \) are obtained from \( L \) and \( R \) according to
Definition 8.

• \( \text{Graph}(L)^{\alpha} \) and \( \text{Graph}(R)^{\alpha} \) are obtained from \( \text{Graph}(L) \) and
\( \text{Graph}(R) \) by replacing each label \( l \) with \( l^\alpha \).

• \( K \) is the discrete graph with \( V_K = V_{\text{Graph}(L)} \cap V_{\text{Graph}(R)} \). All nodes
are unlabelled.

• \( K \rightarrow \text{Graph}(L)^{\alpha} \) and \( K \rightarrow \text{Graph}(R)^{\alpha} \) are the inclusions of
\( K \) in \( \text{Graph}(L)^{\alpha} \) and \( \text{Graph}(R)^{\alpha} \), respectively.

• \( P^\alpha \) is defined by: \( P^\alpha(g) \) if and only if \( g \) is a graph morphism
\( \text{Graph}(L)^{\alpha} \rightarrow G \) such that \( G \in \mathcal{G}^\alpha(\mathcal{L}) \) and \( c^{\alpha, g} = \text{tt} \).

Let \( r \) be the rule-schema identifier associated with the declaration \( D \). Then
we write \( r^\alpha \) for the conditional rule induced by \( D \) and an assignment \( \alpha \).
Given graphs $G, H \in \mathcal{G}^2(\mathcal{L})$, we write $G \Rightarrow_r H$ if there is an assignment $\alpha$ such that $G \Rightarrow_{r, \alpha} H$.

Note that $V_K$ is defined as the intersection of the node sets of $\text{Graph}(L)$ and $\text{Graph}(R)$. Thus, to define the nodes in $K$, the same number of category Node has to be given to these nodes in categories LeftNodes and RightNodes in the rule-schema declaration.

**Example 2 (Induced Conditional Rule)** Consider the input graph in Figure 3.5. Then Figure 3.6 depicts the only two conditional rules induced by the rule schema in Figure 3.1 that are applicable to that graph. Here $P^\alpha$ and $P^\beta$ hold true for a graph morphism $g: \text{Graph}(L)^\alpha \rightarrow G$ respectively $g: \text{Graph}(L)^\beta \rightarrow G$ if and only if there is no edge from $g_V(2)$ to $g_V(1)$. Note that there are other assignments $\gamma$ forming matches $\text{Graph}(L)^\gamma \rightarrow G$ but for these the condition does not hold true. 

\[ \begin{align*}
\text{Figure 3.5: Input graph for the conditional rule schema in Fig. 3.1} \\
\text{Figure 3.6: Induced conditional rules}
\end{align*} \]
3.3 Uniqueness of Induced Rules

For an implementation of GP, it is prohibitive to enumerate all conditional rules induced by a conditional rule schema declaration in order to find a rule that turns a premorphism into a match. Conditional rule schema declarations according to Section 3.1 are called deterministic in [PS04]. This is because having found a premorphism \( g : \text{Graph}(L) \rightarrow G, D \) induces at most one conditional rule \( \tau^\alpha \) such that \( g' : \text{Graph}(L)^\alpha \rightarrow G \) is a match with \( (g'_v, g'_E) = (g_v, g_E) \). Therefore, conditional rule schemata admit an efficient implementation.

![Diagram](image)

**Figure 3.7:** Application of a conditional rule schema

Figure 3.7 illustrates the application of a conditional rule schema to a graph \( G \) in \( G^\tau(L) \) schematically. By a *conditional rule schema*, we refer here to \( (\text{Graph}(L) \leftarrow K \rightarrow \text{Graph}(R), c) \) the components of which are defined as in Definition 8 and Definition 11. We first need to find an injective premorphism \( g \) from \( \text{Graph}(L) \) to \( G \) such that the dangling condition is satisfied. We then need to find an assignment \( \alpha \) such that \( g \) extends to the graph morphism \( g' : \text{Graph}(L)^\alpha \rightarrow G \). If \( P^\alpha(g') \) holds, the induced
3.3. Uniqueness of Induced Rules

rule \((\text{Graph}^\alpha(L) \leftarrow K \rightarrow \text{Graph}^\alpha(R))\) is applied to \(G\) by constructing the natural double-pushouts of Definition 4 using \(g'\).

How \(\alpha\) is determined is explained by Proposition 2 below. If the sequence of labels of a node or edge \(n\) in \(\text{Graph}(L)\) has the same length as the sequence of labels in \(g(n)\), then \(\alpha\) replaces every variable of type \(t\) in the label of \(n\) with its value in the set corresponding to \(t\) at the same position in the sequence of labels in \(g(n)\). The same variable in different labels in \(\text{Graph}(L)\) is always mapped to the same value. Integer numerals and strings of character symbols in labels in \(\text{Graph}(L)\) are mapped to their respective values in \(\mathbb{Z}\) or \(\text{Char}^*\).

In Proposition 1 below we show that there is at most one \(\alpha\) that extends a given premorphism to a graph morphism. This is due to the fact that GP allows only simple expressions (of category SimpleExp) in the label sequences of the left-hand side of a (conditional) rule schema declaration. A label or tag can therefore be an integer numeral, a string of character symbols or a single variable identifier. As integer numerals and strings of character symbols are mapped to their respective values in \(\mathbb{Z}\) or \(\text{Char}^*\) and different occurrences of a variable are always mapped to the same value, there cannot be more than one assignment that extends a given premorphism to a graph morphism. By the process of determining an assignment \(\alpha\) as described above, several conditions must hold for \(\alpha\) to extend a premorphism to a graph morphism. The length of the label sequences and the type of labels must match, same variables in \(\text{Graph}(L)\) indicate the same values in \(G\) etc. This is made precise in Proposition 2 below. Because of these conditions, it is possible that there is no assignment such that a given premorphism extends to a graph morphism.

In [PS04], we allow a less restricted form of expressions in the left-hand side. This, in general, results in an infinite number of possible assignments to extend a premorphism to a graph morphism. We restrain from introducing general rule schemata [PS04] in this thesis as we want to extend premorphisms to morphisms in at most one way to admit an efficient implementation of GP.
We next show that for a given premorphism \( \text{Graph}(L) \to G \), \( D \) induces at most one conditional rule \( r^\alpha \) such that the premorphism extends to a match for \( r^\alpha \).

**Proposition 1** Let \( g: \text{Graph}(L) \to G \) be a premorphism with \( G \in \mathcal{G}^*(L) \). Then there is at most one assignment \( \alpha \) such that \( g \) extends to a graph morphism \( \text{Graph}(L)^\alpha \to G \).

**Proof.** Let \( \alpha \) and \( \beta \) be assignments such that \( g \) extends to graph morphisms \( \text{Graph}(L)^\alpha \to G \) and \( \text{Graph}(L)^\beta \to G \). We show that \( \alpha = \beta \). Consider some \( x \) in \( \text{Var}(D) \). By the context conditions in Section 3.1 we have \( \text{Var}(R) \subseteq \text{Var}(L) \) and \( \text{Var}(c) \subseteq \text{Var}(L) \). Hence, \( x \in \text{Var}(L) \). Thus, by the definition of the categories LeftLabel and SimpleExp in Section 3.1, there is some node or edge label \( e_1 \_ e_2 \_ \ldots \_ e_n \) in \( L \) such that \( e_1, \ldots, e_n \in \text{SimpleExp} \) and \( x = e_k \) for some \( k \) with \( 1 \leq k \leq n \). Without loss of generality, let \( v \) be a node in \( \text{Graph}(L) \) such that \( l_{\text{Graph}(L)}(v) = e_1 \_ \ldots \_ e_n \). Then, by Definition 9, \( l_{\text{Graph}(L)}(v) = e_1^\alpha \_ \ldots \_ e_n^\alpha \) and \( l_{\text{Graph}(L)}(v) = e_1^\beta \_ \ldots \_ e_n^\beta \). Since \( \text{Graph}(L)^\alpha \to G \) and \( \text{Graph}(L)^\beta \to G \) are label-preserving and map \( v \) to the same node in \( G \), it follows \( e_1^\alpha \_ \ldots \_ e_n^\alpha = e_1^\beta \_ \ldots \_ e_n^\beta \). Thus, in particular, \( x^\alpha = e_k^\alpha = e_k^\beta = x^\beta \). We conclude that \( \alpha = \beta \). \( \square \)

Proposition 1 ensures that conditional rule schemata cannot induce more than one conditional rule applicable according to a given premorphism. The next proposition gives a necessary and sufficient condition (informally described above) for such an induction to take place. The condition makes precise how to find the assignment \( \alpha \), if it exists, such that the premorphism becomes a graph morphism.

**Proposition 2** Let \( g: \text{Graph}(L) \to G \) be a premorphism with \( G \in \mathcal{G}^*(L) \). Then, there is an assignment \( \alpha \) such that \( g \) extends to a graph morphism \( \text{Graph}(L)^\alpha \to G \) if and only if the following conditions hold:
(1) For all nodes and edges $a$ in Graph$(L)$ such that $a$ is labelled with $e_1 \ldots e_k$, $g(a)$ is labelled with a sequence $u_1 \ldots u_k$ over $(\mathbb{Z} \cup \text{Char}^*)$ such that for $i = 1, \ldots, k$,

- if $e_i = n$ or $e_i = -n$ with $n \in \text{Num}$, then $u_i$ is the integer value in $\mathbb{Z}$ represented by $n$ respectively $-n$,
- if $e_i = "c_1 \ldots c_k"$ with $c_1, \ldots, c_k \in \text{Char}$, then $u_i = c_1 \ldots c_k$,
- if $e_i \in \text{VarId}$ with type$(e_i) = \text{int}$, then $u_i \in \mathbb{Z}$,
- if $e_i \in \text{VarId}$ with type$(e_i) = \text{string}$, then $u_i \in \text{Char}^*$.

(2) For all nodes and edges $a, a'$ in Graph$(L)$ such that $a$ and $a'$ are labelled with $e_1 \ldots e_p$ and $e'_1 \ldots e'_q$, respectively, and $g(a)$ and $g(a')$ are labelled with sequences $u_1 \ldots u_p$ and $u'_1 \ldots u'_q$ over $(\mathbb{Z} \cup \text{Char}^*)$, respectively: for $i = 1, \ldots, p$ and $j = 1, \ldots, q$, $e_i = e'_j \in \text{VarId}$ implies $u_i = u'_j$.

**Proof.** Suppose first that $g$ extends to a graph morphism Graph$(L)^a \rightarrow G$. If a node or edge $a$ in Graph$(L)$ is labelled with $e_1 \ldots e_k$, then by Definition 9, $a$’s label in Graph$(L)^a$ is the sequence $e_1^a \ldots e_k^a$. Since $g$ is label preserving, $g(a)$ is labelled with $e_1^a \ldots e_k^a$, too. Moreover, for $i = 1, \ldots, k$, $e_i^a$ satisfies conditions (1a) to (1d) by Definition 9. Let now $a$ and $a'$ be nodes or edges in Graph$(L)$ such that $a$ is labelled with $e_1 \ldots e_p$, $a'$ is labelled with $e'_1 \ldots e'_q$, and $a$ and $a'$ in Graph$(L)^a$ are labelled with $e_1^a \ldots e_p^a$ and $e'_1^a \ldots e'_q^a$, respectively. If $e_i = e'_j \in \text{VarId}$ for some $i \in \{1, \ldots, p\}$ and $j \in \{1, \ldots, q\}$, then both $a$ and $a'$ are labelled with $e_i^a = e'_j^a$ in Graph$(L)^a$. Let the labels of $g(a)$ and $g(a')$ be $u_1 \ldots u_p$ and $u'_1 \ldots u'_q$, respectively. Then, since $g$ is label preserving, $u_i = e_i^a = e'_j^a = u'_j$.

Conversely suppose that conditions (1) and (2) are satisfied. Consider any variable $x$ in Var$(D)$. Since Var$(R)$ and Var$(c)$ are contained in Var$(L)$, we have $x \in \text{Var}(L)$. Let $a$ be a node or edge in Graph$(L)$ with label $e_1 \ldots e_k$ such that $x$ occurs in some $e_i$, $1 \leq i \leq k$. Since $e_1, \ldots, e_k \in \text{SimpleExp}$, we have $x = e_i$. By (1), the label of $g(a)$ is a sequence $u_1 \ldots u_k$ over $(\mathbb{Z} \cup \text{Char}^*)$. We define $\alpha(x) = u_i$. By (1c), (1d) and (2), this yields a well-defined assignment $\alpha : \text{Var}(D) \rightarrow (\mathbb{Z} \cup \text{Char}^*)$. Consider now any node
or edge $a$ in $\text{Graph}(L)$ and let $l$ be the label of $a$. Then $l$ has the form $e_1 \ldots e_k$ for some $k \geq 1$, where $e_i \in \text{SimpleExp}$ for $i = 1, \ldots, k$. Hence by (1a), (1b) and the definition of $\alpha$, $\alpha(a)$ is labelled with $P$. Thus $g$ maps each item in $\text{Graph}(L)^\alpha$ to an item with the same label in $G$. That is, $g$ extends to a graph morphism $\text{Graph}(L)^\alpha \to G$. 

Operationally, the application of the conditional rule schema declaration $D$ to a graph $G$ in $G^\alpha(L)$ amounts to the following steps:

1. Transform $D$ into the conditional rule schema $(\text{Graph}(L) \leftarrow K \rightarrow \text{Graph}(R), c)$.

2. Find an injective premorphism $g$: $\text{Graph}(L) \to G$ satisfying the dangling condition.

3. Find an assignment $\alpha$ such that $g$ extends to a graph morphism $g': \text{Graph}^\alpha(L) \to G$ according to Proposition 2.

4. Check whether $c^{\alpha,g'} = \text{tt}$.

5. Construct for $(\text{Graph}(L)^\alpha \leftarrow K \rightarrow \text{Graph}(R)^\alpha)$ and the extension $g': \text{Graph}(L)^\alpha \to G$ of $g$ the natural pushouts of Definition 4 (according to [HP02b]).

### 3.4 Related Work

In this chapter, we introduced the building blocks of GP programs in form of conditional rule schemata. Rule schemata are not the only way to extend graph transformation with calculations on labels. An alternative is to use one of the approaches to attributed graph transformation that have been proposed in the literature. The recent papers [HKT02, EPT04], for example, merge graphs and algebras so that attributed graphs are usually infinite. We rather prefer to work with finite graphs in which “attributes” are ordinary labels.
Our method of working with rule schemata and their induced rules is close to Schied’s approach to double-pushout transformations on graphs labelled with algebra elements [Sch92]. (A single-pushout version of this approach is outlined in [LKW93].) Roughly, his double-pushout diagrams can be decomposed into our double-pushout diagrams with rule schema instantiation by assignments on top of them. A major difference between our approach and [Sch92] is that our rules can relabel nodes whereas the rules in [Sch92] are label-preserving. In this thesis, we chose to use only injective right-hand graph morphisms in rules whereas in [PS04], we introduced rule schemata with the possibility to merge items. Rules in [Sch92] are always injective. Schied also introduced conditions for rules, in the form of propositional formulas over term equations, but he did not consider built-in predicates on the graph structure such as our edge predicate. Another difference is that rule schemata are deterministic in the sense explained in Section 3.3 (that is, they induce unique conditional rules for given premorphisms) whereas [Sch92] allows graphs in the left-hand side of a rule to be labelled with any expressions in category Exp. This concept is also introduced in [PS04] but is omitted in this thesis as it does not admit an efficient implementation of GP.
Chapter 4

The Language GP

In this chapter, we introduce the graph programming language GP (for Graph Programs) [PS04]. GP is a declarative, rule-based graph transformation language. Graph transformation languages apply graph transformation rules, defined in one of the different approaches, to non-deterministically chosen matches in the input graph. Rule application is guided by several control concepts [Sch99].

GP is based on the application of sets of conditional rule schemata (according to Chapter 3) to an input graph. Rule schemata in a set are non-deterministically chosen and then non-deterministically applied to one of the possible matches in the graph. The application of sets of conditional rule schemata is directed by several control constructs. These define how often and in which order sets of rule schemata are applied. GP’s syntax and semantics is based on four core constructs. The semantics of a further four derived constructs is mapped to the semantics of the core.

The aim is to provide a simple language with a complete formal semantics to facilitate formal reasoning. In addition, the language must be practical. In this thesis, we demonstrate GP’s practicality for the implementation of graph algorithms by several case studies in Chapter 5.

We define GP’s semantics in structural operational style [Plo04]. Structural operational semantics focuses on single steps of a transformation and is therefore ideal as a formal basis for an implementation of GP.
We introduce the syntax of GP in Section 4.1. The structural operational semantics for GP is defined in Section 4.2. We give a short comparison to other approaches in Section 4.3.

4.1 Syntax

In this section, we give an abstract context-free syntax for GP programs together with context conditions. The syntax is abstract in that ambiguities in command sequences are not resolved. Note that the focus of this thesis is on a complete formal semantics for GP and not on a concrete syntax. In examples, we use parenthesis to disambiguate command sequences if necessary.

GP programs use control mechanisms to direct the application of sets of conditional rule schemata as introduced in Chapter 3. The main control constructs defined in GP are sequential composition (‘;’), non-deterministic single-step application of a set of conditional rule schemata (RuleSetCall), two branching constructs (if-then-else and try-then-else), and two looping constructs (while-do and as-long-as-possible(‘!’)). In the following, control constructs except sequential composition are called commands. Intuitively, every GP program is a sequence of commands. In addition, to structure programs and reuse program parts, macros and macro calls are introduced.

Example 3 (MakeConnected) Before we formally introduce the syntax of GP programs, we discuss a short example program that adds edges to any integer-labelled input graph until the graph is connected. A graph is connected if there is a path\(^1\) between each two nodes in the underlying undirected graph\(^2\) (see also [Jun02]).

\(^1\)A path from node \(v\) to node \(w\) in a graph \(G\) is a sequence \(e_1, \ldots, e_n\) of edges in \(E_G\) such that \(s_G(e_1) = v\), \(t_G(e_n) = w\) and \(t_G(e_i) = s_G(e_{i+1})\) for \(i = 1, \ldots, n - 1\).

\(^2\)Given a graph \(G\), the underlying undirected graph of \(G\) is obtained from \(G\) by adding for each edge \(e \in E_G\) an edge \(e'\) such that \(s_G(e) = t_G(e')\), \(t_G(e) = s_G(e')\) and \(m_G(e) = m_G(e')\).
main = while Disconnected do Add.
Disconnected = Select; Tag!: Check.
Tag = \{T1, T2\}.

Select(x:int) =
\[
\begin{array}{c}
\text{x} \\
\text{1}
\end{array}
\Rightarrow
\begin{array}{c}
x_0 \\
\text{1}
\end{array}
\]

T1(x, y, z:int) =
\[
\begin{array}{c}
\text{x}_0 \\
\text{1}
\end{array}
\xrightarrow{y}
\begin{array}{c}
z \\
\text{2}
\end{array}
\Rightarrow
\begin{array}{c}
x_0 \\
\text{1}
\end{array}
\xrightarrow{y}
\begin{array}{c}
z_0 \\
\text{2}
\end{array}
\]

T2(x, y, z:int) =
\[
\begin{array}{c}
\text{x}_0 \\
\text{1}
\end{array}
\xrightarrow{y}
\begin{array}{c}
z \\
\text{2}
\end{array}
\Rightarrow
\begin{array}{c}
x_0 \\
\text{1}
\end{array}
\xrightarrow{y}
\begin{array}{c}
z_0 \\
\text{2}
\end{array}
\]

Check(x:int) =
\[
\begin{array}{c}
x \\
\text{1}
\end{array}
\Rightarrow
\begin{array}{c}
x \\
\text{1}
\end{array}
\]

Add(x, y:int) =
\[
\begin{array}{c}
x \\
\text{1}
\end{array}
\begin{array}{c}
y \\
\text{2}
\end{array}
\Rightarrow
\begin{array}{c}
x \\
\text{1}
\end{array}
\xrightarrow{1}
\begin{array}{c}
y \\
\text{2}
\end{array}
\]

where not edge(1, 2) and not edge(2, 1)

Figure 4.1: Example program MakeConnected
The program `MakeConnected` in Figure 4.1 consists of five rule schema declarations (see Section 3.1), one of which is conditional, the macros `Disconnected` and `Tag` and the main command sequence following the keyword `main`. Given an integer-labelled input graph, the program continuously tests whether the input graph is disconnected. If this is the case, `Disconnected` succeeds and a new 1-labelled edge is added to the original input by `Add`, whose condition forbids inserting edges between two nodes that are directly connected. For the test, the macro `Disconnected` first selects a single node by adding a 0-tag (`Select`) and then adds tags to all untagged adjacent nodes of 0-tagged nodes as long as this is possible (`Tag`). If there is an untagged node left, `Check` is applicable and the graph is not connected. In this case, the computation proceeds with applying `Add` to the (untagged) graph to which `Disconnected` has been applied. The program `MakeConnected` terminates if the subprogram `Disconnected` fails for all possible executions because `Check` is not applicable (input is connected).

Figures 4.2a and 4.2b show an example run of the program `MakeConnected`. The symbol `Δ` in Figure 4.2b is a placeholder for all possible remaining executions of (`Select; Tag!; Check`) gained from non-deterministically applying `Select` and `Tag`. All of these executions fail like the two example executions shown in Figure 4.2b because `Check` is not applicable to any of the graphs resulting from a `Select; Tag!` execution. Therefore, the condition of `while` evaluates to false and the program execution terminates. The result of the run in Figure 4.2b is one out of the possible 18 results presented in Figure 4.3. Note that the rule schema `Add` is applied non-deterministically and therefore may also insert edges between already connected nodes.

The example program in Figure 4.1 is an artificial example chosen here to demonstrate the use of some GP constructs. The program `MakeConnected` is inefficient because edges may be inserted between nodes that are already connected and the graph is repeatedly tagged. The program in Figure 4.4 is an improved version of `MakeConnected` that only inserts edges between un-connected nodes and does not repeatedly tag the graph. The program first tags a node with 0 and then repeatedly tags connected nodes of 0-tagged
Figure 4.2a: An example run of MakeConnected
Figure 4.2b: An example run of MakeConnected (continued). Here, \( \triangle \) stands for all possible remaining executions of (Select; Tag; Check) on the top left graph.
Figure 4.3: Possible results for MakeConnected
main = Select; (Tag!; AddNew)!; Untag!.

Tag = \{T1, T2\}.

Select(x:int) =
\[
\begin{array}{cc}
\text{x} & \Rightarrow \text{x}_0
\end{array}
\]

T1(x, y, z:int) =
\[
\begin{array}{cc}
\text{x}_0 & \text{y} & \Rightarrow \\
\text{z} & & \\
\text{1} & & \text{2}
\end{array}
\]

T2(x, y, z:int) =
\[
\begin{array}{cc}
\text{x}_0 & \text{y} & \Rightarrow \\
\text{z} & & \\
\text{1} & & \text{2}
\end{array}
\]

AddNew(x, y:int) =
\[
\begin{array}{cc}
\text{x}_0 & \Rightarrow \\
\text{y} & \\
\text{1} & \text{2}
\end{array}
\]

Untag(x:int) =
\[
\begin{array}{cc}
\text{x}_0 & \Rightarrow \\
\text{} & \\
\text{1} & 1
\end{array}
\]

**Figure 4.4:** Improved program of MakeConnected
nodes as long as possible and inserts an edge between an untagged and a tagged node. The execution of \((\text{Tag}!; \text{AddNew})\) terminates if \text{AddNew} is not applicable because there is no untagged node left. Then, all 0-tags are removed by \text{Untag}.

In some GP examples, the use of iteration as-long-as-possible (‘!’) is less intuitive than using \text{while} because the termination condition of the iteration is hidden in the enclosed program text whereas in \text{while}, the termination condition is formulated directly in the condition. This hiding makes it sometimes difficult to see in which cases the program execution terminates.

We next introduce the abstract syntax for GP programs.

**Programs**

\[
\begin{align*}
\text{Prog} & \ ::= \ \text{Decl} \{ \text{Decl} \} \\
\text{Decl} & \ ::= \ \text{RuleSchemaDecl} \mid \text{MacroDecl} \mid \text{MainDecl} \\
\text{MacroDecl} & \ ::= \ \text{MacroId} \ ' = ' \ \text{ComSeq} \ ' . ' \\
\text{MainDecl} & \ ::= \ \text{main} \ ' = ' \ \text{ComSeq} \ ' . ' \\
\text{MacroId} & \ ::= \ \text{Identifier}
\end{align*}
\]

Context Conditions:

1. There is exactly one declaration of category \text{MainDecl} in \text{Prog}.

2. The command sequence following a macro identifier \(m\) in a macro declaration does not contain a macro call for \(m\) (recursive program calls are forbidden). Mutual recursion of macros is also forbidden, that is the call structure is acyclic.

3. All macro identifiers (\text{MacroId}) declared in the same program are distinct.

4. All rule-schema identifiers (\text{RuleId}) are distinct and distinct with the macro identifiers (\text{MacroId}) declared in the same program (\text{Prog}).
A program consists of a number of declarations of conditional rule schemata and macros, and exactly one declaration of a main command sequence. The context conditions require the uniqueness of identifiers in a program and that macros are not called recursively.

**Command Sequences**

\[
\begin{align*}
\text{ComSeq} & ::= \text{Com} \{?\} \text{Com} \\
\text{Com} & ::= \text{RuleSetCall} | \text{MacroCall} \\
& \quad | \text{if ComSeq then ComSeq [else ComSeq]} \\
& \quad | \text{try ComSeq then ComSeq [else ComSeq]} \\
& \quad | \text{while ComSeq do ComSeq} \\
& \quad | \text{ComSeq '!' } \\
& \quad | \text{skip | fail} \\
\text{RuleSetCall} & ::= \text{RuleId} | '{' [\text{RuleId} '{', \text{RuleId}] '}' \\
\text{MacroCall} & ::= \text{MacroId}
\end{align*}
\]

**Context Conditions:**

1. A rule-schema identifier (RuleId) in a call in RuleSetCall refers to a declaration of a conditional rule schema (RuleSchemaDecl) in the same program.

2. A macro identifier (MacroId) in a call in MacroCall refers to a declaration of a macro (MacroDecl) in the same program.

Macros are simply a means to structure programs and thereby to make them more readable. Every program can be transformed into an equivalent macro-free program by replacing macro calls with their associated command sequences (recursive macros are not allowed by the context conditions).
Assumption 2 (Macro-free Programs) In this chapter and Chapter 5, we use the notions program and command sequence synonymously, assuming that macro calls have been replaced with their associated command sequences and that every rule-schema identifier in RuleId comes with a unique declaration. Hence, every program consists of a single, macro-free command sequence together with the declarations of all rule-schema identifiers occurring in the command sequence.

4.2 Structural Operational Semantics

In this section, we present a semantics for GP in the style of Plotkin’s structural operational semantics (short SOS) [Pl04, NN07]. This type of semantics is often called small-step semantics as it describes how individual steps of a computation take place. The focus is not on the overall result of a computation as would be the case when using denotational style semantics [All86, Ten91, Win93, NN07] but on the steps leading to a result. Thus, structural operational semantics sets the right perspective for an implementation of the language.

4.2.1 Configurations and Inference Rules

The SOS semantics is given in form of inference rules which define a small-step transition relation “$\rightarrow$” on configurations. Configurations are either intermediate or terminal. In imperative languages, intermediate configurations typically consist of a statement of the language together with a (system) state that represents the memory, and terminal configurations consist of only a system state. Statements in imperative languages correspond to command sequences in GP. System states in imperative languages correspond to graphs in GP. However, to model the case that a set of rules in GP is not applicable, we need a failure state. This additional state is denoted by “fail” in GP’s semantics.
Consequently, the transition relation for GP's semantics has the form:

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

A configuration is an element of the set $(\text{ComSeq} \times G^v(\mathcal{L})) \cup G^v(\mathcal{L}) \cup \{\text{fail}\}$. A configuration $\gamma$ is terminal if there is no configuration $\delta$ such that $\gamma \rightarrow \delta$.

Inference rules in GP are of the general form

\[
\begin{array}{c|c}
\text{Name} & \text{Premises} \rightarrow \text{Conclusion} \\
\hline
\end{array}
\]

where Name is the name of the inference rule, Premises is a set of premises and Conclusion a transition of the form $\langle P, G \rangle \rightarrow \gamma$ with $P$ in category ComSeq, $G$ in $G^v(\mathcal{L})$ and $\gamma$ is a configuration. If all premises are satisfied, the conclusion is implied and the inference rule is applicable. Premises in GP can be transitions, expressions over transitions or expressions over the applicability or non-applicability of sets of rule schemata. For example, the expressions "$G \Rightarrow_R \mathcal{H}$" and "$G \not\subseteq \text{Dom}(\Rightarrow_R)$" with $G, \mathcal{H}$ in $G^v(\mathcal{L})$ and $\mathcal{R}$ in RuleSetCall, that define applicability or non-applicability conditions on sets of rule schemata, may be premises for inference rules in GP.

If an inference rule has an empty set of premises, we omit the horizontal bar and only write the conclusion. In this case, the conclusion is called an axiom. Inference rules use meta-variables for command sequences, ruleset calls and graphs, e.g. $P, \mathcal{R}$ and $G$ as above, that are to be considered universally quantified.

Each rule-schema identifier (RuleId) in a call of category RuleSetCall stands for the set of all conditional rules induced by the declaration for that identifier. A call of the form $\{r_1, \ldots, r_n\}$ stands for the union of the rule sets induced by the declarations for $r_1$ to $r_n$.

Inference rules in GP use meta-variables $C, P, P', Q$ for command sequences in category ComSeq, $G$ and $H$ for graphs in $G^v(\mathcal{L})$ and $\mathcal{R}$ for a call in category RuleSetCall. We write $G \Rightarrow_\mathcal{R} \mathcal{H}$ if there is a rule-schema identifier
4.2 Structural Operational Semantics

In $\mathcal{R}$ such that $G \Rightarrow_r H$ (confer the notation introduced below Definition 11). For example, the rule

$$[\text{Call}_1] \quad \frac{G \Rightarrow_r H}{(\mathcal{R}, G) \rightarrow H}$$

in Definition 12 should be read as: “For all $\mathcal{R} \in \text{RuleSetCall}$ and all $G, H \in \mathcal{G}^*(\mathcal{L})$, $G \Rightarrow_r H$ implies $\langle \mathcal{R}, G \rangle \rightarrow H$”.

4.2.2 Definition of the Semantics

We next define the structural operational semantics for GP. Semantically, the commands defined by the grammar in Section 4.1 are dependent on each other. In the following, we define four core constructs before defining the other constructs through the core constructs. These core constructs are the call of a set of conditional rule schemata, sequential composition (’;’), the if-then-else statement and the while-do statement.

Definition 12 (SOS inference rules) Figure 4.5 shows the inference rules for the structural operational semantics of GP, where $\rightarrow^+$ is the transitive closure of $\rightarrow$ and where a command sequence $C$ finitely fails on a graph $G \in \mathcal{G}^*(\mathcal{L})$ if (1) there does not exist an infinite sequence $\langle C, G \rangle \rightarrow \langle C_1, G_1 \rangle \rightarrow \ldots$ and (2) for each terminal configuration $\gamma$ such that $\langle C, G \rangle \rightarrow^+ \gamma$, $\gamma = \text{fail}$.

Note that $C$ finitely fails on $G$ if and only if each computation starting from the configuration $\langle C, G \rangle$ eventually ends in the fail configuration. The latter can happen only if $C$ contains a call of a rule set that is not applicable to any of the graphs in the intermediate configurations. This is because the only rule that generates fail is $[\text{Call}_2]$ whereas all other rules either propagate failure ($[\text{Seq}_3]$) or use it to verify finite failure ($[\text{If}_2]$, $[\text{While}_2]$).

Let $E$ be an execution $\langle P, G \rangle \rightarrow \langle P', G' \rangle \rightarrow \ldots \rightarrow \gamma$. We say $E$ terminates with a graph if $\gamma \in \mathcal{G}^*(\mathcal{L})$, $E$ terminates with failure if $\gamma = \text{fail}$ and $E$ gets stuck if $\gamma \in (\text{ComSeq} \times \mathcal{G}^*(\mathcal{L}))$. An infinite execution $\langle P, G \rangle \rightarrow \langle P', G' \rangle \rightarrow \ldots$ is said to be non-terminating.
[Call$_1$] \[ G \Rightarrow^R H \]
\[ \langle R, G \rangle \rightarrow H \]

[Call$_2$] \[ G \not\in \text{Dom}(\Rightarrow^R) \]
\[ \langle R, G \rangle \rightarrow \text{fail} \]

[Seq$_1$] \[ \langle P, G \rangle \rightarrow \langle P', H \rangle \]
\[ \langle P; Q, G \rangle \rightarrow \langle P'; Q, H \rangle \]

[Seq$_2$] \[ \langle P, G \rangle \rightarrow H \]
\[ \langle P; Q, G \rangle \rightarrow \langle Q, H \rangle \]

[Seq$_3$] \[ \langle P, G \rangle \rightarrow \text{fail} \]
\[ \langle P; Q, G \rangle \rightarrow \text{fail} \]

[If$_1$] \[ \langle C, G \rangle \rightarrow^+ H \]
\[ \langle \text{if } C \text{ then } P \text{ else } Q, G \rangle \rightarrow \langle P, G \rangle \]

[If$_2$] \[ C \text{ finitely fails on } G \]
\[ \langle \text{if } C \text{ then } P \text{ else } Q, G \rangle \rightarrow \langle Q, G \rangle \]

[While$_1$] \[ \langle C, G \rangle \rightarrow^+ H \]
\[ \langle \text{while } C \text{ do } P, G \rangle \rightarrow \langle P; \text{while } C \text{ do } P, G \rangle \]

[While$_2$] \[ C \text{ finitely fails on } G \]
\[ \langle \text{while } C \text{ do } P, G \rangle \rightarrow G \]

**Figure 4.5:** The SOS inference rules
The inference rules [Seq1] and [Seq2] express that to execute \( P; Q \) on the input graph \( G \), \( P \) is executed first and if that execution terminates with a graph \( H \), \( Q \) is executed on \( H \) in the next step. If the execution of \( P \) resulted in an intermediate configuration \( \langle P', G' \rangle \), then [Seq1] expresses that we first have to complete the execution of \( P' \) on \( G' \) before continuing with the execution of \( Q \).

The rules [If1], [If2], [While1] and [While2] first execute the condition \( C \) and then execute the respective program on the original input graph of the command depending on the result of the execution of \( C \). If there is at least one execution of \( C \) on a graph \( G \) that terminates with a graph, the premise of rules [If1] and [While1] is satisfied. If \( C \) finitely fails on \( G \), the premise of rules [If2] and [While2] is satisfied.

Configurations of the form \( \langle \text{while} \ C \ \text{do} \ P, \ G \rangle \) and \( \langle \text{if} \ C \ \text{then} \ P \ \text{else} \ Q, \ G \rangle \) are stuck\(^3\) if the execution of the condition \( C \) on input \( G \) neither terminates with a graph nor finitely fails on \( G \), as then no inference rule is applicable. This is the case if there exists an infinite sequence \( \langle C, \ G \rangle \rightarrow \langle C', \ G' \rangle \rightarrow \ldots \), or a stuck configuration of the form \( \langle C', \ G' \rangle \) such that \( \langle C, \ G \rangle \rightarrow^* \langle C', \ G' \rangle \), or both.

Note that the only GP program executions that may be non-terminating are the ones where the program contains a \texttt{while-do} or an iteration as-long-as-possible ('!') command. The latter is formally defined through \texttt{while-do} by the inference rule in Figure 4.6. Only \texttt{while-do} executions may be non-terminating because \texttt{while-do} is the only command that may be applied repeatedly. For example, the execution of \texttt{while \ C \ do \ P} is non-terminating if \( C \) and \( P \) always terminate with a graph because in this case, [While1] is always applicable. If an execution of \( P \) is non-terminating, the result for that execution of \texttt{while} is undefined as [Seq1] is always applicable. In case an execution of \( P \) terminates with failure, that execution of \texttt{while \ C \ do \ P} terminates with failure since [Seq3] is applicable.

\(^3\)A configuration \( \langle P, G \rangle \) is stuck if there is no configuration \( \gamma \) such that \( \langle P, G \rangle \rightarrow \gamma \).
The semantics of the remaining commands given by the grammar in Section
4.1 is defined by the semantics of the core constructs of Definition 12. Figure
4.6 shows the inference rules for the derived constructs in GP.

[Alap] \[\langle P!, G \rangle \rightarrow \langle \text{while } P \text{ do } P, G \rangle\]

[Skip] \[\langle \text{skip}, G \rangle \rightarrow \langle r_{\text{skip}}, G \rangle\]

where \( r_{\text{skip}} \) is a rule-schema identifier whose declaration
induces the rule \( \{\emptyset \leftarrow \emptyset \rightarrow \emptyset\} \)

[Fail] \[\langle \text{fail}, G \rangle \rightarrow \langle \{\}, G \rangle\]

[Try\textsubscript{1}] \[\langle \text{try } C \text{ then } P \text{ else } Q, G \rangle \rightarrow \langle \text{if } C \text{ then } C; P \text{ else } Q, G \rangle\]

[Try\textsubscript{2}] \[\langle \text{try } C \text{ then } P, G \rangle \rightarrow \langle \text{try } C \text{ then } P \text{ else skip}, G \rangle\]

[I\textsubscript{3}] \[\langle \text{if } C \text{ then } P, G \rangle \rightarrow \langle \text{if } C \text{ then } P \text{ else skip}, G \rangle\]

**Figure 4.6: Inference rules for derived constructs**

The command \( P! \) means that \( P \) is executed as long as possible, that is \( P \)
is executed consecutively \( k \geq 0 \) times until the next execution of \( P \) finitely
fails on its input. A special case exists if there is at least one execution of
\( P \) that terminates with a graph and other executions of \( P \) terminate with
failure or get stuck. In this case, the execution of \( P! \) can either terminate
with a graph or with failure or get stuck. \( P! \) never finitely fails as this would
imply that on the one hand there is an execution of \( P \) that terminates with
a graph (condition evaluation) and on the other hand that all executions of
\( P \) terminate with failure (\( P \) execution). \text{skip} is mapped to a rule schema
inducing the rule \( \{\emptyset \leftarrow \emptyset \rightarrow \emptyset\} \) which is applicable to every graph. \text{fail}
is mapped to the empty set of rule schemata the execution of which will
always terminate with fail. \text{try} executes the condition \( C \) on the input
graph if there is at least one execution of \( C \) that terminates with a graph. A special case exists if there is an execution of \( C \) that terminates with a graph and other executions of \( C \) that terminate with failure or get stuck. In this case, the execution of \texttt{try} can also terminate with a graph or with failure or get stuck. If \( C \) finitely fails on the input graph, the program \( Q \) is executed on the input graph. \texttt{try-then} and \texttt{if-then} are short forms for \texttt{try-then-else} and \texttt{if-then-else} in which the \texttt{else} part contains a \texttt{skip} application.

The concept of finite failure stems from logic programming where it is used to define negation [Cla78]. In the context of GP it is a powerful tool to “hide” destructive tests.

**Example 4 (Undirected)** Consider the program in Figure 4.7 that tests whether an integer-labelled, loop-free\(^4\) input graph can be considered to be undirected. A graph \( G \) in \( \mathcal{G}^+(\mathcal{L}) \) can be considered \textit{undirected} if the set of edges \( E_G \) can be partitioned into pairs \((e, e')\) such that for all pairs \((e, e') : s_G(e) = t_G(e')\), \( t_G(e) = s_G(e')\), \( m_G(e) = m_G(e')\) and \( e \) and \( e' \) are not in any other pair. The subprogram \texttt{NotUndirected} removes as long as possible pairs of edges between two nodes pointing in opposite directions and having the same label. It then tries to apply the rule schema \texttt{Test} that checks whether there are edges left in the remaining graph. If this is the case, the input graph cannot be considered undirected. The \texttt{then}-branch is executed on the original input graph and the rule schema \texttt{No} produces a node labelled with a negative answer. If \texttt{Test} finitely fails on its input (no edges left), the \texttt{else}-branch is executed on the original input graph and \texttt{Yes} produces a positive answer.

The execution of the subprogram \texttt{NotUndirected} is destructive as it destroys the input graph. Without a concept of returning to the original input graph, the programmer would have to copy the graph first or insert tags to reestablish the original graph after executing a destructive test. Otherwise the information about the input graph is lost. Destructive programs in the \texttt{if-then-else} or the \texttt{while-do} condition have no lasting effect on the

---

\(^4\)A graph \( G \in \mathcal{G}^+(\mathcal{L}) \) is \textit{loop-free} if there is no edge \( e \) such that \( s_G(e) = t_G(e) \).
main = if NotUndirected then No else Yes.

NotUndirected = Remove!; Test.

Remove(x,y,z:int) =

\[
\begin{array}{c}
\text{x} \\
1
\end{array}
\quad \frac{y}{\Rightarrow}
\quad \begin{array}{c}
\text{z} \\
2
\end{array}
\quad \frac{y}{\Rightarrow}
\quad \begin{array}{c}
\text{x} \\
1
\end{array}
\quad \begin{array}{c}
\text{z} \\
2
\end{array}
\]

Test(x,y,z:int) =

\[
\begin{array}{c}
\text{x} \\
1
\end{array}
\quad \frac{y}{\Rightarrow}
\quad \begin{array}{c}
\text{z} \\
2
\end{array}
\quad \frac{y}{\Rightarrow}
\quad \begin{array}{c}
\text{x} \\
1
\end{array}
\quad \begin{array}{c}
\text{z} \\
2
\end{array}
\]

No =

\[
\emptyset \quad \Rightarrow \quad "\text{No}"\]

Yes =

\[
\emptyset \quad \Rightarrow \quad "\text{Yes}"\]

Figure 4.7: Example program Undirected
input graph as the branches of the constructs are executed on the original input graph. By offering this form of test in GP, programming with graph transformation becomes simpler. However, one sometimes explicitly wants to keep the changes done by the test program if the test succeeds. For this purpose, the try-then-else command may be used.

Figure 4.8 shows an example run of the program in Figure 4.7 to an input graph that can be considered undirected and Figure 4.9 to an input graph that cannot be considered undirected. Note that the resulting graphs are unique, that is, input/output behaviour of the program is deterministic.

![Diagram](image)

**Figure 4.8: A “Yes”-run of Undirected**

![Diagram](image)

**Figure 4.9: A “No”-run of Undirected**

The SOS semantics defines all program executions for a given GP program and input graph. It thus gives us all results of a computation. We are mostly interested in the set of results in $G^e(L)$ since GP is designed for the transformation of graphs. An interesting special case is given if all resulting graphs are isomorphic, meaning that the result of a computation is deterministic (see for example Figures 4.8 and 4.9). However, we are
also interested in other results such as finite failure of the program. To
define these result sets we are interested in, we define the following semantic
function:

**Definition 13 (Semantic Function \( S_{sos} \))** The function

\[
S_{sos} : \text{ComSeq} \to (\mathcal{G}^+(\mathcal{L}) \to (2^{\mathcal{G}^+(\mathcal{L})} \cup \{\text{fail}\} \cup \{\bot\})) \text{ is defined by}^{5}
\]

\[
S_{sos}[P]G = \begin{cases} 
\{ H \in \mathcal{G}^+(\mathcal{L}) \mid \langle P, G \rangle \to^+ H \} & \text{if there is } H \in \mathcal{G}^+(\mathcal{L}) \text{ with} \\
\langle P, G \rangle \to^+ H, & \text{fail if } P \text{ finitely fails on } G, \\
\bot & \text{otherwise.}
\end{cases}
\]

The function \( S_{sos} \) yields the set of all graphs that are derivable from \( G \) with
the SOS inference rules (provided there exists at least one), it returns \text{fail}
if \( P \) finitely fails on \( G \) and it produces \( \bot \) for all other cases. The result \( \bot \)
indicates that there exists an infinite sequence \( \langle P, G \rangle \to \langle P_1, G_1 \rangle \to \ldots \),
or a stuck configuration of the form \( \langle P', G' \rangle \) such that \( \langle P, G \rangle \to^* \langle P', G' \rangle \),
or both, and no graph in \( \mathcal{G}^+(\mathcal{L}) \) is derivable.

For example, the set of graphs depicted in Figure 4.3 is the set
\( S_{sos}[\text{while Select; \{T1,T2\}; Check do Add}]G \) where \( G \) is the input
graph in the example run in Figure 4.2a. As an extreme example, the result
of \( S_{sos}[\text{skip!}] \) is \( \bot \) as \text{skip!} never terminates. The result of \( S_{sos}[\text{fail}]G \) is
obviously \text{fail}. The result of \( S_{sos}[\text{while skip! do skip}] \) is \( \bot \) as \text{skip!} never
terminates and thus, the configuration \( \langle \text{while skip! do skip}, G \rangle \) is stuck
for all graphs \( G \) in \( \mathcal{G}^+(\mathcal{L}) \) and no graph in \( \mathcal{G}^+(\mathcal{L}) \) is derivable.

Note that the number of different output graphs for a program need not be
finite. For example, consider the program in Figure 4.10.

Executed on a graph \( G \) containing a single node labelled 1, \( S_{sos}[\{r_1,r_2\}]G \)
is the following infinite set:

\[
\left\{ \emptyset, 2, 2, 2, 2, 2, 2, \ldots \right\}
\]

---

5 Given \( P \in \text{ComSeq} \) and \( G \in \mathcal{G}^+(\mathcal{L}) \), we write \( S_{sos}[P]G \) for the result of applying the
function \( S_{sos}(P) \) to \( G \).
main = \{r_1, r_2\}!

r_1 =
\begin{align*}
1 & \implies 1 \quad 2
\end{align*}

r_2 =
\begin{align*}
1 & \implies 0
\end{align*}

\textbf{Figure 4.10: Example program with infinite set of output graphs}

Program equivalence is important in the context of program transformation, for example when searching for more intuitive or more efficient versions of a program in existence [Rey98]. To discuss two general program equivalences formally in Section 4.2.3, we next define semantic equivalence.

\textbf{Definition 14 (Semantic Equivalence)} Two GP programs \(P\) and \(Q\) are \textit{semantically equivalent}, denoted by \(P \equiv Q\), if \(S_{\text{so}1}[P] = S_{\text{so}1}[Q]\).

\subsection{4.2.3 Some Program Equivalences}

In this section, we prove two semantic program equivalences according to Definition 14. The first is an equivalence between \texttt{if} and \texttt{while}.

\textbf{Proposition 3 (while Unravelling)} Let \(C\) and \(P\) be GP programs. Then \(\text{while } C \text{ do } P \equiv \text{if } C \text{ then } (P; \text{while } C \text{ do } P)\).

\textbf{Proof.} We prove the equivalence by showing that \(S_{\text{so}1}[\text{while } C \text{ do } P]G = S_{\text{so}1}[\text{if } C \text{ then } (P; \text{while } C \text{ do } P)]G\) for any \(G \in \mathcal{G}^\Sigma(\mathcal{L})\). There are three cases to consider. \textit{Case 1: } \(\langle C, G \rangle \rightarrow^* H\) for some \(H \in \mathcal{G}^\Sigma(\mathcal{L})\). Then, with \([\text{While}]\), which is the only rule applicable to \(\langle \text{while } C \text{ do } P, G \rangle\), we get \(\langle \text{while } C \text{ do } P, G \rangle \rightarrow \langle P; \text{while } C \text{ do } P, G \rangle\). Also, with \([\text{If}_3]\) and \([\text{If}_1]\), where these are the only rules applicable to \(\langle \text{if } C \text{ then } (P; \text{while } C \text{ do } P), G \rangle\), we get
\(\langle \text{if } C \text{ then } (P; \text{while } C \text{ do } P), \ G \rangle \rightarrow^+ \langle P; \text{while } C \text{ do } P, \ G \rangle\) which proves the equivalence.  

**Case 2:** \(C\) finitely fails on \(G\). Then with \([\text{While}_2]\), which is the only rule applicable to \(\langle \text{while } C \text{ do } P, \ G \rangle\), we get \(\langle \text{while } C \text{ do } P, \ G \rangle \rightarrow G\). Also, with \([\text{If}_3],[\text{If}_2],[\text{Skip}]\) and \([\text{Call}_1]\), where these are the only rules applicable to \(\langle \text{if } C \text{ then } (P; \text{while } C \text{ do } P), \ G \rangle\), we get \(\langle \text{if } C \text{ then } (P; \text{while } C \text{ do } P), \ G \rangle \rightarrow^+ G\) which proves the equivalence.

**Case 3:** There is no \(H \in \mathcal{G}^\ast(\mathcal{L})\) such that \(\langle C, \ G \rangle \rightarrow^+ H\), and \(C\) does not finitely fail on \(G\). Then for both initial configurations, \(\langle \text{while } C \text{ do } P, \ G \rangle\) and \(\langle \text{if } C \text{ then } (P; \text{while } C \text{ do } P), \ G \rangle\), there is no rule applicable so that both configurations are stuck. Thus, \(S_{\text{sos}}[\text{while } C \text{ do } P]G = \bot = S_{\text{sos}}[\text{if } C \text{ then } (P; \text{while } C \text{ do } P)]G\).

\[\square\]

We next prove an equivalence between two of the derived constructs.

**Proposition 4 (! Unravelling)**  
Let \(P\) be a GP program. Then \(P! \equiv \text{if } P \text{ then } (P!)\).

**Proof.** We prove the equivalence by showing that \(S_{\text{sos}}[P!]G = S_{\text{sos}}[\text{if } P \text{ then } (P!)]G\) for any \(G \in \mathcal{G}^\ast(\mathcal{L})\). There are three cases to consider.

**Case 1:** \(\langle P, \ G \rangle \rightarrow^+ H\) for some \(H \in \mathcal{G}^\ast(\mathcal{L})\). Then, with \([\text{If}_3]\) and \([\text{If}_1]\), where these are the only rules applicable to \(\langle \text{if } P \text{ then } (P!), \ G \rangle\), we get \(\langle \text{if } P \text{ then } (P!), \ G \rangle \rightarrow^+ \langle P!, \ G \rangle\) which proves the equivalence.

**Case 2:** \(P\) finitely fails on \(G\). Then, with \([\text{Alap}]\) and \([\text{While}_2]\), where these are the only rules applicable to \(\langle P!, \ G \rangle\), we get \(\langle P!, \ G \rangle \rightarrow^+ G\). Also, with \([\text{If}_3],[\text{If}_2],[\text{Skip}]\) and \([\text{Call}_1]\), where these are the only rules applicable to \(\langle \text{if } P \text{ then } (P!), \ G \rangle\), we get \(\langle \text{if } P \text{ then } (P!), \ G \rangle \rightarrow^+ G\) which proves the equivalence.  

**Case 3:** There is no \(H \in \mathcal{G}^\ast(\mathcal{L})\) such that \(\langle P, \ G \rangle \rightarrow^+ H\), and \(P\) does not finitely fail on \(G\). Then, with \([\text{Alap}]\), which is the only rule applicable to \(\langle P!, \ G \rangle\), we get \(\langle P!, \ G \rangle \rightarrow \langle \text{while } P \text{ do } P, \ G \rangle\). Then, for both configurations, \(\langle \text{while } P \text{ do } P, \ G \rangle\) and \(\langle \text{if } P \text{ then } (P!), \ G \rangle\), there is no rule applicable so that both configurations are stuck. Thus, \(S_{\text{sos}}[P!]G = \bot = S_{\text{sos}}[\text{if } P \text{ then } (P!)]G\).

\[\square\]
4.3 Related Work

The GP language is based on the programming constructs introduced in [HP01, HP]. The language in [HP01] consists of just three control constructs, namely non-deterministic application of a set of ordinary double-pushout rules either in a single step or iterated as long as possible, and sequential composition. The language is proven to be computationally complete and minimal in that omitting either sequential composition or iteration results in a computationally incomplete language.

The language in [HP01] is extended in this thesis to provide a practical yet simple language with a formal semantics to facilitate formal reasoning. The first extension is the change of the basis from ordinary double-pushout rules to conditional rule schemata (see [PS04] and Chapter 3) to facilitate operations on labels and more programming comfort. The second extension is the introduction of a branching concept (if-then-else) and a loop concept (while-do) to provide more programming comfort (see also Chapter 5).

In the following, we compare GP with other rule-based languages in Section 4.3.1 and with graph-transformation languages in Section 4.3.2.

4.3.1 Rule-based Languages

The rule-based language Gamma [BFL01] is based on the transformation of multisets. Gamma programs in form of conditional rewrite rules are applied as long as possible and may be composed using sequential composition. Thus, Gamma programs compare to the sequential composition of rule schemata iterated as long as possible (\(\mathcal{R}!\)) in GP.

There are several extensions to basic Gamma in the literature [BFL01]. The extensions closest to GP commands are extensions with composition operators. Apart from a sequential composition operator, Gamma was also extended with a parallel application operator. The latter applies Gamma programs in any possible order until none can proceed further. Parallel
application of GP programs is not implemented yet. A careful prior analysis of the advantages of such extension will have to take place.

Another popular extension of Gamma is called Structured Gamma [FL98]. In [FL98], the Gamma language is extended with structured multisets to express structuring of data. Graph grammars on structured multisets define types. A comparable approach for graphs is proposed in [BPR04, DP06] for the definition and verification of so-called shapes by graph reduction systems. The latter approach could be used in GP to define shapes of input graphs and prove statically that a GP program preserves the shape of the input graph through transformation.

The language ELAN [BKKM02] uses term-rewriting rules with conditions similar to those of rule schemata. In ELAN, rule application is controlled by functions, called strategies. The input to a strategy is a term and the output is a multiset of results. Thus, strategies compare to command sequences in GP. A strategy fails if the resulting multiset is empty. The list of strategy constructors presented in [BKKM02] includes the application of a labelled rule (Call), sequential composition, repeat as long as possible, id (Skip), fail and several constructors to choose a strategy out of a set of strategies. The latter strategies typically choose a strategy that does not fail in the sense of failure in ELAN. Such choice constructors could be implemented in GP by defining sets of GP programs out of which one program is chosen for execution that does not finitely fail on the input. However, general choice constructors are not implemented in GP for the sake of semantic simplicity.

Gamma and ELAN are defined by formal semantics using rewriting logic [BFL90, BKKM02]. As we explain in Section 4.3.2, existing graph-transformation languages often lack such a complete formal definition.

### 4.3.2 Graph Transformation Languages

A guiding principle in our design of GP is syntactic and semantic simplicity, which distinguishes GP from the complex PROGRES language [SWZ99]. Our approach also differs from a language such as AGG [ERT99] in that we
insist on a formal semantics. We want GP to be semantics-based since we consider the ability to formally reason on programs as a key feature.

Except for the language AGG [ERT99] that implements only very limited control (application of a set of rules with the same priority (layered rules) as long as possible), other graph-transformation languages such as PROGRES and Fujaba [FNTZ98] and model transformation languages based on graph-transformation such as GReAT [KASS03] and VTCL [BV06] offer sequential composition, rule calls, branching and iteration. For a comparison of the three best-known graph-transformation languages PROGRES, Fujaba and AGG see also [FMRS07]. How the syntax and semantics of the above concepts is defined differs in these languages.

Our concepts of iteration (while-do) and branching (if-then-else) that use finite failure for the condition evaluation are closest to the iteration construct (loop) and the branching construct using the definedness operator def in PROGRES [Sch90, SWZ99]. The latter returns true if the application of a program results in a graph, false if the program cannot be successfully applied and the result is undefined if the program is not executable in finite time. The application of a conditional statement in PROGRES has no effect on the input graph. The unsuccessful case is similar to our concept of finite failure. PROGRES uses backtracking to try all executions of a program and fails only if all the executions terminate but none of them terminates with a graph. In contrast to PROGRES and GP, VTCL allows only iteration as long as possible of a set of rules comparable to the command /R! in GP.

The non-deterministic choice operator for PROGRES programs is similar to the choice operator in the Gamma language (see Section 4.3.1). This concept is not implemented in the successor of PROGRES, called Fujaba [FNTZ98]. A special concept in PROGRES and Fujaba in comparison to other graph transformation languages are parameterised calls. These are comparable to GP procedures and procedure calls introduced in Chapter 6.

Implementing graph algorithms in PROGRES is tedious as edges in the graph model may not carry attributes and therefore the use of edge weights
often needed in graph algorithms (see Chapter 5 for examples) is not directly implementable. An edge in an edge-weighted graph has to be simulated by a node with one ingoing and one outgoing edge that carries the weight attribute. That makes programs difficult to comprehend.

The drawback of the PROGRES language is that it is unsuitable for verification. The language description in [Sch90] contains 300 to 350 rules defining the static and dynamic semantics of the language which is too much to support verification. The semantics of PROGRES programs is given by control diagrams that are similar to control-flow diagrams. Thus, the semantics of PROGRES is defined in a form of operational semantics.

The UML and Java development platform Fujaba [FNTZ98] is a direct successor of PROGRES. Fujaba is simpler than PROGRES because some of the constructs of PROGRES are not implemented in Fujaba (e.g. non-deterministic choice and backtracking). However, Fujaba lacks a complete formal semantics. The control flow is directed by transitions in so-called Story Diagrams. These are graphically defined and similar to control diagrams in PROGRES.

A different concept of introducing control to graph-transformation languages is introduced in [GdL07]. Here, the control in form of recursive rule-application is added to the basis of double-pushout rules. Rules are equipped with a base and a recursion condition. A rule is applied as long as the recursion conditions are satisfied. Matches are passed between successive recursion steps. Recursive rules can also be equipped with negative application conditions. Typically the recursive transformation or search of graphs is realised by adding additional control elements to the input graph and applying rules using these elements recursively. Such artificial recursion control could be replaced by introducing recursive rules to the language. As this type of recursion is added to ordinary double-pushout rules it is imaginable to extend GP with such rules and transfer the concept to rule schemata.
AGG, PROGRES and Fujaba implement type graphs (predefined types for nodes and edges including arity of ingoing and outgoing edges) as a means to control the shape of a graph during transformation. GP uses a very limited form of typing. Labels in graphs in GP need to match the predefined types — string or int — of variables and the number of possible tags given by the labelling in a conditional rule schema in order to be matched. However, a more sophisticated type system for graphs in GP might be preferable in future.

VTCL, PROGRES and Fujaba allow variable declarations, variable updates and function definitions. As introducing such a concept on the one hand makes the language more complex and might therefore not be suitable for verification anymore and on the other hand leads away from a pure graph-transformation language to a more imperative language, GP does not implement these concepts.

AGG uses single-pushout rules with negative application conditions [HHT96]. Conditions on the labels in form of expressions in the Java language are attached to rules. Thus, the basis of AGG is similar to the basis of GP. However, we prefer using expressions defined within the language and having a clear semantics for the evaluation of conditions.

PROGRES and AGG both have some desirable features that are not implemented in GP apart from the noted ones. PROGRES offers, for example, the possibility to match sets of nodes of the same structure in the left-hand side of a rule and the possibility of matching paths. The AGG system, for example, offers a critical pair analysis for AGG rules [HKT02]. The term critical pairs stems from term rewriting where this technique is used to verify that a system is confluent. Critical pair analysis has been generalised to graph rewriting [Plu93, Plu99] and is realised in AGG. These concepts are not realised in GP.

The researchers involved in work on the model-transformation languages GReAT and VTCL (VIATRA2 framework) aim at giving a complete formal semantics for their languages. The authors of [KASS03] claim that the
semantics of GReAT, defined in the Object-Z notation, is formally specified. However, some mappings in [KASS03] are for brevity reasons described in English. The semantics of VTCL is described by programs over Abstract State Machines (ASM).

A different language approach is taken in [Kus00] with the introduction of so-called transformation units (short TUnits). TUnits are not a language but a structuring principle as the concept abstracts from graph-transformation approaches. Units are similar to procedures in GP (introduced in Chapter 6) where TUnits include their own rule declarations and an initial and terminal graph specification. TUnits may be called in a commands section within another TUnit. In [Kus00], the semantics for a system consisting of several TUnits with an arbitrary call structure (recursive calls are allowed) is given in denotational-style semantics (see [NN07]). To prove the well-definedness of the language it was assumed that the control constructs used are continuous. Continuity of commands could not be proven in [Kus00] as the set of control constructs is not fixed in this approach.
Chapter 5

Small Case Studies

In this chapter, we present three case studies in the domain of graph algorithms. Case studies in our sense include a description of the problem, a solution to the problem in the form of a GP program and a formal discussion of that program. The formal discussion may include proofs of termination, correctness and time-complexity of the respective program. Where possible, we compare theoretic complexity results with results from program runs on the GP system [MP06, MP07]. The focus of this chapter, however, is not on the GP environment, but on the GP programs and formal reasoning about them.

The three case studies we present are Dijkstra’s single-source shortest path algorithm [Eve89, CLR00, Jun02], Floyd-Warshall’s all-pairs shortest-path algorithm [Eve89, CLR00, Jun02] and a vertex-colouring algorithm [Jun02]. Where the first two produce unique results, the third algorithm is non-deterministic in the results. With these case studies, we show that GP is practical and demonstrate that GP is suitable for verification. We refrain from introducing an axiomatic system or an assertion language for GP as the focus of this thesis is primarily on the language GP and not on formal verification.

We review basic notions of graph algorithms and case studies in Section 5.1. The GP environment is briefly introduced in Section 5.2. Dijkstra’s shortest path algorithm is discussed in Section 5.3, Floyd-Warshall’s algorithm is
discussed in Section 5.4 and the vertex-colouring algorithm is discussed in Section 5.5. We conclude the chapter with a brief comparison to case studies in other languages in Section 5.6.

## 5.1 Preliminaries

In the following, we recall some notions related to graph algorithms. A graph $G$ is called *simple* if for all nodes $u, v \in V_G$ there is at most one edge $e$ in $E_G$ such that $s_G(e) = u$ and $t_G(e) = v$. Given a graph $G$ and nodes $v$ and $w$ in $V_G$, a *path* from $v$ to $w$ is a sequence $e_1, \ldots, e_n$ of edges in $E_G$ such that $s_G(e_1) = v$, $t_G(e_n) = w$ and $t_G(e_i) = s_G(e_{i+1})$ for $i = 1, \ldots, n-1$. In such path $e_1, \ldots, e_n$, the nodes $v_2, \ldots, v_{n-1}$ with $v_i = t_G(e_i)$ for $1 < i < n$ are called *inner nodes*. If all edges in $G$ are labelled with numbers, the *distance* of such a path in $G$ is the sum of its edge labels. A *shortest path* between two nodes is a path of minimal distance. A node $v$ is called *reachable* from a node $w$ if there exists a path from $w$ to $v$. We call a path from $v$ to $w$ a *cycle* if $v = w$. A *cycle of negative length* is a cycle for which the sum of all edge labels in the cycle is negative. The distance of a shortest path from $u$ to $v$ with inner nodes $I \subseteq V_G$ is denoted by $sd(u, v, I)$. Note that if $I = V_G$, $sd(u, v, I)$ denotes the distance of a shortest path from $u$ to $v$ in $G$.

A graph $G$ is called *transitive* if there is a path from $u$ to $v$ with $u \neq v$ in $G$ if and only if there is an edge from $u$ to $v$ in $G$. Let $G$ and $H$ be two graphs whose edges are labelled with numbers such that $V_G = V_H$. Then $H$ is called *distance preserving* if for all edges $e$ from $u$ to $v$ in $H$ there is a path from $u$ to $v$ of distance $m_H(e)$ in $G$.

We next define notions closely related to case studies in the domain of graph algorithms. In general, a program execution in GP may either terminate with a graph or with failure (see Chapter 4). To define termination over all possible program executions, we next define program termination. A GP program $P$ *terminates* if all executions of $P$ terminate either with a graph or with failure.
In general, all executions of GP programs that implement graph algorithms should terminate with a graph as the result of the algorithm is encoded in the graph structure of the output graph. Thus, an execution with failure would not provide any information. GP programs in this chapter will indeed never terminate with failure. In this chapter, all of the programs only use iteration as-long-as-possible ("!") applied to a set of rule schemata. An execution of such a command can never terminate with failure because the set is either applicable or not and thus the iteration will always terminate with a graph (confer the GP semantics in Chapter 4). The conditions in while commands used in this chapter are single rule-schemata. These are either applicable or not. Thus, an execution of a while command never gets stuck. Single-step applications of rule schemata in the GP programs in this chapter are always applicable because the input graphs for these always contain the structure matched by the left graph. As no other commands are used in this chapter, we may assume that all GP programs in this chapter terminate with a graph if they terminate at all. Hence, for the rest of this chapter we use the term termination whenever we mean termination with a graph.

The semantics of a GP program is the semantics of consecutive rule schema applications induced by the programs command sequence (see SOS semantics in Definition 12). Thus, we define the time-complexity of a GP program by the number of rule-schema applications in a derivation sequence.\footnote{The implementation of conditional rule-schema applications and their complexity are outside the scope of this thesis.} We say a GP program $P$ has (time-)complexity $m$ if $P$ terminates after $m$ rule schema applications. We consider only best- and worst-case time complexity and use the Big O notation\footnote{If $g$ and $f$ are functions of $x$, we say $f(x)$ is $O(g(x))$ if for some constants $k$ and $j$, $|f(x)| \leq k|g(x)| + j$ for all $x$.} to give a measure.

A graph-class specification defines labelling and structure of a set of graphs in general terms. The following definition, for example, is a graph class specification: Let $G$ be a directed, loop-free graph in $G^x(\mathcal{L})$ in which all nodes and edges are labelled with untagged integers. We will use this general form
of graph-class specification to define the relationship between input and expected output graphs for a given program to argue program correctness.

A GP program $P$ is called \textit{partially correct} if the result of any terminating execution of $P$ (on a graph that conforms to a given input graph-class specification) satisfies a given output (graph-class) specification. $P$ is called \textit{totally correct} or \textit{correct} if $P$ terminates on all input graphs (that conform to a given input graph-class specification) and the result always satisfies a given output (graph-class) specification.

A \textit{case study} consists of a description of the problem and solution to the problem at hand and a formal discussion of properties such as termination, total correctness and (possibly) time-complexity of the GP program implementing the solution. Note that a case study in our sense is not just a GP program for a given problem but rather a formal analysis of the program. Also note that a formal discussion in general may cover more properties, e.g. uniqueness of result. In this thesis we focus on the above only.

In the following discussions, we often give names to programs. These names are a synonym for the command sequence representing the program.

\section{5.2 The GP Environment}

In this section, we briefly introduce the GP environment \cite{MP06, MP07} to be able to compare theoretic time-complexity with time-complexity computed by the system. We measure time-complexity in the GP system by the number of graph changes in an execution. Graph changes directly correspond to rule applications as each rule application results in a fixed number of graph changes.

Figure 5.1 shows the general design of the GP system. It comprises an editor, called GPE, and an abstract machine, called YAM (York Abstract Machine). The editor consists of a graphical editor for rule schemata and programs and an additional graphical editor for input graphs. Programs are written as shown in the examples in this thesis (see Chapter 4). GPE
Currently offers all the general features of graphical editors including easy insertion, deletion and labelling of graph elements, insertion of new rule schemata, storing and loading of programs and graphs, etc. There is no public release of the GP environment, yet. Thus, we use the version dated the 6th of September 2007 in this thesis.

![Diagram of the GP environment](image)

**Figure 5.1: The GP environment**

To run a GP program, the graphical program is converted into a textual form similar to that introduced in Section 4.1. This textual program is then compiled into YAM bytecode. The YAM bytecode is a language independent format. The bytecode is then executed on the YAM machine. The result of the execution is converted into a graph that is then again represented in the graph editor in GPE.

The YAM system also offers a backtracking mode. In this mode, all possible results (answers) of a computation are computed by executing all possible derivation sequences. Since graph matching and selection of rules out of sets are non-deterministic choices, the possible derivation sequences are produced by backtracking through these decisions and choosing another match or rule if possible.

The system does not distinguish isomorphic graphs in the set of results. The set of results, computed by the system when using the backtracking mode, is available for graphical representation in the editor. Hence, the system or editor may be used for testing program correctness.
Using the backtracking mode, the number of graph changes in the different
derivation sequences can be computed. Thus, we get the number of graph
changes for all possible execution sequences resulting from different non-
deterministic choices. In this thesis, we are only interested in best and
worst case time-complexity. Therefore, we need a program that produces
all results in backtracking mode, counting the number of graph changes
in each derivation sequence and returning the number of graph changes in
the best and worst case execution. Complexity is always dependant on the
size $n$ of the input graph. Thus, a randomised input graph of size $n$ would
be the preferred input for such program. However, the GP system cannot
produce such random graphs. Hence, we assume that the user provides the
necessary random input graphs for time-complexity computations.

The computation of graph changes is done by the program FastWorstCase.
FastWorstCase (YAM) (SETUPPROG) (TESTPROG) (INPUTSIZE) ex-
ppects a YAM-compiled textual version$^3$ of a GP program (SETUPPROG)
that transforms a single node with label INPUTSIZE into an input graph
for the test GP program (TESTPROG) representing the solution to the
problem at hand. The output of FastWorstCase is the overall number of
answers (to get an estimate on the size of the set of results) and the mini-
umum and maximum number of graph changes of the program TESTPROG
executed on the graph generated by SETUPPROG. We use this command
in the following sections to confirm our theoretic time-complexity results.

We also use the GP system in Chapter 7 for testing.

Example 5 Consider the following textual version (stored in gen-
erate,yamcil) of the GP setup program Generate. The program gets as input
a single node with label $k$ and produces $k$ nodes with label $10$.

---

$^3$The textual version of a visual GP program is output in a terminal window by
pressing the button “Write” in the editor. The textual version of a program, stored in a
file ending with “.yamcil”, is compiled into a yam machine program using the command
“YAMCompile -m -b name.yamcil name.yam”.
main = Tag; Add!; Delete.
Tag(x:int) = \{1 : x\} => \{1 : x - 1\}
Add(x:int) = \{1 : x - 1\} => \{1 : x - 1 - 1, 2 : 10\} where x > 0
Delete(x:int) = \{1 : x - 1\} => \{

The generate.yamcil program is then compiled into yam code using the command “YAMCompile -m -b generate.yamcil generate.yam”.

Assume now the test program is already compiled and called test.yam. Then we get the minimum and maximum number of graph changes in all executions of test.yam on a graph with three nodes labelled with 10 using the command “FastWorstCase YAM generate.yam test.yam 3”.

5.3 Dijkstra’s Single-Source Shortest Path Algorithm

The so-called single-source shortest path algorithm by Dijkstra [CLR00, Jun02] computes the distances from a given start node to all other nodes in a graph whose edges are labelled with nonnegative numbers. Dijkstra’s algorithm stores the distance from the start node to a node \( v \) in a variable \( d(v) \). Initially, the start node’s distance is set to the value 0 and the distance of every other node in the graph is set to \( \infty \). Nodes for which the shortest distance has been computed are added to a set \( S \), which is empty in the beginning. In each step of the algorithm, first a node \( w \) from \( V_G - S \) is added to \( S \), where \( d(w) \) is minimal. Then for each edge \( e \) outgoing from \( w \), \( d(t_G(e)) \) is changed to \( \min(d(t_G(e)), d(w) + m_G(e)) \). After execution of the algorithm, \( S = V_G \) and \( d(v) \) is the shortest distance from the start node to \( v \in V_G \). If \( d(v) = \infty \), there is no path from the start node to \( v \).

In the following, we discuss two GP programs implementing Dijkstra’s algorithm in Sections 5.3.1 and 5.3.2. In each of these case studies, we discuss termination, correctness and complexity of the program.
5.3.1 The GP Program Simple_Dijkstra

The first program for Dijkstra’s single-source shortest-path algorithm, Simple_Dijkstra, is given in Figure 5.2. We implement the algorithm in a non-destructive way. In general, we are not interested in the original node labels and their type in the input graph. For reasons of simplicity, we assume the labels to be integers. Original node labels are not altered by any of the rules. The distance value for each node is stored in the nodes tag. If a node in the output graph is untagged, that implies that this node is not reachable from the start node. Thus, an untagged node in the output graph corresponds to a distance value of $\infty$ in the original version of the algorithm.

\[ \text{main} = \text{S_InitTags!; S_Reduce!} \]

\[ \text{S_InitTags}(x,y,z,k: \text{int}) = \]

\[ \begin{array}{c}
\circ x_k \quad y \\
\hline
1 \\
\end{array} z \quad \implies \quad \begin{array}{c}
\circ x_k \quad y \\
\hline
1 \\
\end{array} z_{k+y} \]

\[ \text{S_Reduce}(x,y,z,k,j: \text{int}) = \]

\[ \begin{array}{c}
\circ x_k \quad y \\
\hline
1 \\
\end{array} z_j \quad \implies \quad \begin{array}{c}
\circ x_k \quad y \\
\hline
1 \\
\end{array} z_{k+y} \]

where $(k + y) < j$

**Figure 5.2:** The program Simple_Dijkstra

The program Simple_Dijkstra first applies S_InitTags iteratively as-long-as-possible and then S_Reduce iteratively as-long-as-possible to the

---

4 Otherwise identical rule schemata for each type in category Type would have to be written.
result of \texttt{S\_InitTags}! The rule schema \texttt{S\_InitTags} tags every untagged node \( v \) with the distance value of a path from the start node to \( v \). \texttt{S\_Reduce} changes a stored distance value to a smaller one whenever a shorter path has been found.

We expect the input graph to be loop-free and untagged except for the start node \( s \) that carries a 0-tag to distinguish it from other nodes. In standard implementations of this algorithm (see for example [Sed02]), the start node is chosen by the programmer indicated in GP by tagging it. We assume that the graph is loop-free for simplicity of writing reasons (otherwise additional rules have to be written to deal with this case).

**Proposition 5 (Correctness of Simple Dijkstra)** Let \( G \) be an untagged, loop-free graph in \( G^\ast (L) \) containing a unique start node \( s \) with tag 0, whose nodes are labelled with integers and whose edges are labelled with nonnegative integers. When started from \( G \), \texttt{Simple Dijkstra} terminates and produces a unique graph \( H \) that is obtained from \( G \) by tagging each node \( v \) that is reachable from the start node \( s \) with the shortest distance from \( s \) to \( v \).

**Proof.** Termination of \texttt{Simple Dijkstra} follows from the fact that every application of \texttt{S\_InitTags} reduces the number of untagged nodes, and that every application of \texttt{S\_Reduce} reduces the sum of all node tags in a graph.

Let now \( H \) be a graph such that \( \langle \texttt{Simple Dijkstra}, G \rangle \rightarrow^+ H \) where we use the name \texttt{Simple Dijkstra} as a synonym for the program’s command sequence. Since there are no rule schemata for adding or deleting nodes and edges, altering edges or changing node labels, it is clear that \( H \) can be obtained from \( G \) by tagging nodes. Thus, \( H \) is uniquely determined if each node \( v \) reachable from \( s \) is tagged with the shortest distance from \( s \) to \( v \) and that all other nodes (if any) are untagged. To show the former, we need the following invariance property.

**Invariant 1** Let \( \langle \texttt{S\_InitTags!}, G \rangle \rightarrow^+ H_0 \) and \( \langle \texttt{S\_Reduce!}, H_0 \rangle \rightarrow^+ H' \). Then for each node \( v \) in \( H' \), either \( l_{H'}(v) = x \cdot d \) where \( d \) is the distance of a path from \( s \) to \( v \) or \( l_{H'}(v) = x \) (untagged) where \( x \) is \( v \)'s original node label.
Proof. The proposition holds for $H_0$, because $s$ is tagged with 0 and every other node is either tagged with the distance of a path from $s$ to $v$ or untagged as S_InitTags only tags connected nodes where the source node of the connecting edge is already tagged. Moreover, it is easy to see that every application of S_Reduce preserves the claimed property. □

Suppose now that there is a node $v$ in $H$ such that $l_H(v) = x \_d$ and $d$ is not the shortest distance from $s$ to $v$. We distinguish two cases.

Case 1: $v = s$. Since $v$ is tagged with 0 in $G$ and tags are not deleted by any of the rule schemata, and $l_H(v) = x \_d$ with $d \neq 0$, there must be an application of S_Reduce that changes $v'$s label to a negative number. But this contradicts the above claim.

Case 2: $v \neq s$. By the above claim, there is a path from $s$ to $v$ (as otherwise $l_H(v) \neq x$). Let $e_1, \ldots, e_n$ be a shortest path from $s$ to $v$. Let $v_0 = s$ and $v_i = t_H(e_i)$ for $i = 1, \ldots, n$. By Case 1, $l_H(v_0) = x \_0$. Hence, there is some $k$, $1 \leq k \leq n$, such that $l_H(v_k) = x_k \_d_k$ and $d_k$ is not the shortest distance from $v_0$ to $v_k$ and for $i = 0, \ldots, k - 1$, $l_H(v_i) = x_i \_d_i$ and $d_i$ is the shortest distance from $v_0$ to $v_i$. Now since $e_1, \ldots, e_n$ is a shortest path to $v_n$ it follows that $e_1, \ldots, e_k$ is a shortest path to $v_k$ and that $e_1, \ldots, e_{k-1}$ is a shortest path to $v_{k-1}$. So the shortest distance from $v_0$ to $v_k$ is $\sum_{i=1}^{k-1} m_H(e_i) + m_H(e_k) = d_{k-1} + m_H(e_k)$. As this sum is smaller than $d_k$, S_Reduce is applicable to $e_k$. But this contradicts the fact that the application of S_Reduce to $H$ has failed. □

The correctness of Simple_Dijkstra was easy to show, however the program can be expensive in the number of applications of the rule schema S_Reduce. For example, the right-hand derivation sequence in Figure 5.3 contains 40 applications of S_Reduce and represents the worst-case program run of the given input graph of 5 nodes. In contrast, Dijkstra’s algorithm (as sketched at the beginning of this section) changes distances only 10 times when applied to the same graph. Although Simple_Dijkstra needs only 4 applications of S_Reduce in the best case, there is no guarantee that
Figure 5.3: An example run of Simple_Dijkstra and Dijkstra
it does not choose the worst case. In fact, the next proposition shows that there exist input graphs on which the number of $\text{S\_Reduce}$ applications grows exponentially.

![Diagram of graph with labels $l_{n+1}$, $l_n$, $l_{n-1}$, $l_1$, $l_0$ and edges with labels $2^n$, $2^{n-1}$, $2^0$.]

**Figure 5.4:** Input permitting an exponential run of Simple Dijkstra

**Proposition 6 (Simple Dijkstra may use exponential time)** Let $G$ be a graph of the form shown in Figure 5.4, where $n \geq 0$. Then there are graphs $G', H$, and an integer $k > \sum_{i=0}^{n} 2^i$ such that $G \Rightarrow_{\text{S\_InitTags}}^{p} G' \Rightarrow_{\text{S\_Reduce}}^{q} H$ with $p + q = k$.

**Proof.** Call a graph of the form

![Diagram with labels $a_{x_{n+1}}$, $b_{x_n}$, $c_{x_{n-1}}$, $d_{x_1}$, $e_{x_0}$ and edges with labels $2^n$, $2^{n-1}$, $2^0$.]

a chain if $x_i - x_{i+1} \in \{0, 2^i, 2^{i+1}\}$ for $i = 0, \ldots, n$. We first show that chains are closed under a certain “rightmost” strategy of $\text{S\_Reduce}$ applications.

**Claim 1** Let $C$ be a chain and $C \Rightarrow C'$ an application of $\text{S\_Reduce}$ to the edge $e$ such that

- all nodes reachable from $t_C(e)$ have the same tag and
- $m_C(e) = 2^i$ if $x_i - x_{i+1} = 2^{i+1}$, where $y_{i-1}x_i = t_C(t_C(e))$ and $y_{i+1}x_{i+1} = t_C(s_C(e))$.

Then $C'$ is a chain.
**Proof.** We have \( x_i - x_{i+1} \in \{2^i, 2^{i+1}\} \) as otherwise \texttt{S\_Reduce} could not be applied to \( e \).

Case 1: \( x_i - x_{i+1} = 2^i \). Then \( m_C(e) = 0 \) and \( C' \) is obtained from \( C \) by replacing \( x_i \) with \( x_{i+1} \). Hence, in \( C' \), \( x_i - x_{i+1} = 0 \) and, if \( i \geq 1 \), \( x_{i-1} - x_i = 2^i \) since \( x_{i-1} = x_i \) in \( C \).

Case 2: \( x_i - x_{i+1} = 2^{i+1} \). Then \( m_C(e) = 2^i \) and \( C' \) is obtained from \( C \) by replacing \( x_i \) with \( x_{i+1} + 2^i \). Hence, in \( C' \), \( x_i - x_{i+1} = 2^i \) and, if \( i \geq 1 \), \( x_{i-1} - x_i = 2^{i+1} - 2^i = 2^i \) since \( x_{i-1} = x_i \) in \( C \).

Thus, in both cases \( C' \) is a chain. \( \square \)

Let now \( G' \) be the graph such that \( \langle \texttt{S\_InitTags!}, G \rangle \to^+ G' \) where \texttt{S\_InitTags} is applied to the edges with labels \( 2^n, \ldots, 2^0 \) (in that order). Then \( G' \) is the chain \( C \) where \( x_{n+1} = 0 \) and for \( i = 0, \ldots, n \), \( x_i = \sum_{j=i}^{n} 2^j \). Hence there is a derivation \( C \Rightarrow^q \texttt{S\_Reduce} H \) where \( H \) is the chain with \( x_i = 0 \) for \( i = 0, \ldots, n+1 \) and where each application of \texttt{S\_Reduce} satisfies the conditions of the above claim. This is because every chain in which not all nodes have label 0 contains an edge satisfying the conditions, and every application of \texttt{S\_Reduce} reduces the sum of the node labels in a chain. Since \( C \Rightarrow^q \texttt{S\_Reduce} H \) reduces \( x_0 \) from \( \sum_{j=0}^{n} 2^j \) to 0 and, by the above claim, each application of \texttt{S\_Reduce} either preserves \( x_0 \) or decrements it by 1, it follows \( k \geq \sum_{j=0}^{n} 2^j \). Adding the initial steps \( G \Rightarrow^q \texttt{S\_InitTags} C \) therefore gives \( k > \sum_{j=0}^{n} 2^j \). \( \square \)

Note that \( \langle \texttt{S\_Reduce!}, G \rangle \to^+ H \) is equivalent to writing \( G \Rightarrow^q \texttt{S\_Reduce} H \) as we have shown in Proposition 5 that \texttt{S\_Reduce!} terminates and the semantics of GP in Section 4.2.2 gives that \( \langle \texttt{S\_Reduce!}, G \rangle \to \langle \text{while} \texttt{S\_Reduce} \text{ do} \texttt{S\_Reduce}, G \rangle \to^i G \Rightarrow \texttt{S\_Reduce} G^1 \Rightarrow \texttt{S\_Reduce} G^2 \ldots G^k \Rightarrow \texttt{S\_Reduce} H = G \Rightarrow^k \texttt{S\_Reduce} H \) where \( k \geq 0 \). In the following, we use the form \( G \Rightarrow^k \texttt{R!} H \) for \( \langle \texttt{R!}, G \rangle \to^+ H \) whenever we have shown that \( \texttt{R!} \) terminates.

We have shown that \texttt{Simple\_Dijkstra} can be expensive in the number of rule applications. This is due to the non-determinism in the selection
of matches by \texttt{S\_Reduce}. Therefore, we next introduce a more efficient version of Dijkstra’s algorithm.

### 5.3.2 The GP Program \texttt{Dijkstra}

The program \texttt{Dijkstra} of Figure 5.5 uses a while-loop to repeatedly select a node of minimal distance and to update only the distances of nodes reachable from that node by a direct edge. Nodes that have not yet been selected are tagged by 1. Removing the 0 tag from a node by \texttt{Next} corresponds to adding that node to the set \( S \) of the original algorithm. Note that \texttt{Dijkstra} is essentially deterministic: \texttt{Min!} always determines a node of minimal distance among all tagged nodes, and \texttt{Reduce} is applied only to edges outgoing from this node.

The program \texttt{Dijkstra} is destructive in that we store the distance values in the labels of nodes and thus destroy original node labels. This is done for simplicity reasons. However, a non-destructive program similar to \texttt{Simple\_Dijkstra} can easily be written, where two additional tags are inserted, the first one storing the distances and the second one marking a node as selected or not selected.

To distinguish nodes with computed distance from those not reachable from the start node, \texttt{Mark!} tags all nodes not reachable from the start node with 2. As 2-tags are not altered by any of the rules, 2-tagged nodes in the output graph mark nodes not reachable from the start node.

**Proposition 7 (Correctness of \texttt{Dijkstra})** Let \( G \) be an untagged, loop-free graph in \( G^*(L) \) containing a unique start node \( s \) with tag 0 whose nodes are labelled with integers and whose edges are labelled with non-negative integers. When started from \( G \), \texttt{Dijkstra} terminates and produces a unique graph \( H \) which is obtained from \( G \) by labelling each node \( v \) reachable from the start node \( s \) with the shortest distance from \( s \) to \( v \) and tagging all unreachable nodes with 2.
main = Prepare;

while Tagged do (Min!; Reduce!; Next); CleanUp.

Tagged(x:int) =

$\begin{array}{c}
\circ x_{-1} \\
1
\end{array} \quad \Rightarrow \quad \begin{array}{c}
\circ x_{-1} \\
1
\end{array}$

Min(x,y:int) =

$\begin{array}{c}
\circ x_{0} \\
1
\end{array} \quad \begin{array}{c}
\circ y_{1} \\
2
\end{array} \quad \Rightarrow \quad \begin{array}{c}
\circ x_{1} \\
1
\end{array} \quad \begin{array}{c}
\circ y_{0} \\
2
\end{array}$

where $y < x$

Reduce(x,y,z:int) =

$\begin{array}{c}
\circ x_{0} \\
1
\end{array} \quad y \quad \begin{array}{c}
\circ z_{1} \\
2
\end{array} \quad \Rightarrow \quad \begin{array}{c}
\circ x_{0} \\
1
\end{array} \quad y \quad \begin{array}{c}
\circ y_{1} \\
2
\end{array}$

where $(x + y) < z$

Next(x,y:int) =

$\begin{array}{c}
\circ x_{0} \\
1
\end{array} \quad \begin{array}{c}
\circ y_{1} \\
2
\end{array} \quad \Rightarrow \quad \begin{array}{c}
\circ x \\
1
\end{array} \quad \begin{array}{c}
\circ y_{0} \\
2
\end{array}$

CleanUp(x:int) =

$\begin{array}{c}
\circ x_{0} \\
1
\end{array} \quad \Rightarrow \quad \begin{array}{c}
\circ x \\
1
\end{array}$

Figure 5.5: The program Dijkstra
Prepare = Start; InitTags!; Mark!

Start(x:int) =
\[
\begin{array}{ccc}
  x_0 & \Rightarrow & 0_0 \\
  1 & & 1
\end{array}
\]

InitTags(x,y,z,j:int) =
\[
\begin{array}{ccc}
x_j & y & \Rightarrow & x_j & y_{j+1} \\
  1 & z & & 1 & z
\end{array}
\]

Mark(x:int) =
\[
\begin{array}{ccc}
x & \Rightarrow & x_2 \\
  1 & & 1
\end{array}
\]

**Figure 5.6** The macro Prepare

**Proof.** We first show that Dijkstra terminates on input G. The initialisation (Start; InitTags!; Mark!) terminates because every application of InitTags and Mark reduces the number of nodes without tag. Therefore, the execution of InitTags! and Mark! finitely fails after a finite number of rule applications. The body of the while loop terminates since every application of Min reduces the sum of the integer labels of all nodes tagged with 0 (of which there is actually exactly one), and since every application of Reduce reduces the sum of all node labels. The while loop terminates as every execution of its body reduces the number of nodes tagged with 1 and therefore the application of Tagged finitely fails after a finite number of while iterations.

Let now \( \langle Dijkstra, G \rangle \rightarrow^+ H \). It is clear that H can be obtained from G by relabelling G’s nodes because the rule schemata have an effect on node
labels only. To show that all untagged nodes \( v \) in \( H \) are labelled with the shortest distance from \( s \) to \( v \), we first prove the following invariant.

**Invariant 2** Let \( \langle \text{Start}; \ InitTags!; \ Mark!, \ G \rangle \rightarrow^+ G_0 \) and \( \langle \text{while Tagged do (Min!; Reduce!; Next), G}_0 \rangle \rightarrow^+ H \). Then for each node \( v \) in \( H \), either \( l_H(v) = d \) or \( l_H(v) = d - i \) where \( d \) is the distance of a path from \( s \) to \( v \) and \( i \in \{0,1\} \) or \( l_H(v) = x - 2 \) where \( x \) is \( v \)'s original node label.

**Proof.** The proposition holds for \( G_0 \) because \( s \) is tagged with 0 and thus relabelled to 0 by \text{Start} and every other node \( v \) is either relabelled with the distance of a path from \( s \) to \( v \) by \text{InitTags} where \text{InitTags} adds an additional 1-tag to \( v \) or is tagged with 2 by \text{Mark} where the original node label is not altered. As 2-tagged nodes are not altered by any of the rules, the proposition holds for 2-tagged nodes. It is easy to see that \text{InitTags} indeed relabels nodes \( v \) with the distance of a path from \( s \) to \( v \) as only those nodes \( v \) are relabelled that are target of an edge connecting \( v \) with an already tagged node for which a distance has been computed. Moreover, it is easy to see that the body of \text{while} preserves the claimed property because every application of \text{Min} only alters tags and every application of \text{Reduce} preserves the claimed property. Note that the condition of \text{while} has no effect on the input graph. \( \square \)

Suppose now that there is an untagged node \( v \) in \( H \) such that \( l_H(v) \) is not the shortest distance from \( s \) to \( v \) in \( G \).

**Case 1:** \( v = s \). Since \( v \) is labelled with 0 after application of \text{Start} and this label is not changed by any of the other rules as \( v \)'s distance is the smallest one in the beginning and \( l_H(v) \neq 0 \), there must be a rule that changes \( v \)'s label to a negative number which is a contradiction.

**Case 2:** \( v \neq s \). By Invariant 2, there is a path from \( s \) to \( v \) of length \( l_H(v) = d \). Let \( e_1, \ldots, e_n \) be a shortest path from \( s \) to \( v \). Let \( v_0 = s \) and \( v_i = t_H(e_i) \) for \( i = 1, \ldots, n \). By Case 1, \( l_H(v_0) = 0 \). Hence, there is some \( k, 1 \leq k \leq n \), such that \( v_k = v \) and \( l_H(v_k) \) is not the shortest distance from \( v_0 \) to \( v_k \) and for \( i = 0, \ldots, k - 1 \), \( l_H(v_i) \) is the shortest distance from \( v_0 \) to \( v_i \). Now since
\(e_1, \ldots, e_n\) is a shortest path to \(v_n\) it follows that \(e_1, \ldots, e_k\) is a shortest path to \(v_k\) and that \(e_1, \ldots, e_{k-1}\) is a shortest path to \(v_{k-1}\). So the shortest distance from \(v_0\) to \(v_k\) is \(\sum_{i=1}^{k-1} m_H(e_i) + m_H(e_k) = l_H(v_{k-1}) + m_H(e_k)\).

Since \(l_H(v) \neq x-2\), there must be a node \(u\) such that \(l_H(v_k) = l_H(u) + m_H(e)\) where \(s_H(e) = u\) and \(t_H(e) = v_k\). We have to distinguish two cases.

**Case a:** \(l_H(v_{k-1}) < l_H(u)\). In this case \(v_{k-1}\) is chosen by \(\text{Min}\) in an iteration before \(u\) is chosen and thus \(l_H(v_k) = l_H(v_{k-1}) + m_H(e_k)\) by application of \(\text{Reduce}\) which is a contradiction to the fact that \(v_k\) is not labelled with the shortest distance.

**Case b:** \(l_H(v_{k-1}) > l_H(u)\). In this case \(v_{k-1}\) is chosen by \(\text{Min}\) in an iteration after \(u\) is chosen but since \(l_H(u) + m_H(e) > l_H(v_{k-1}) + m_H(e_k)\) (otherwise \(l_H(v_{k-1}) + m_H(e_k)\) would not be the shortest distance from \(s\) to \(v_k\)), \(l_H(v_k)\) is changed to \(l_H(v_{k-1}) + m_H(e_k)\) by \(\text{Reduce}\) in this later application contradicting the fact that \(v_k\) is not labelled with the shortest distance from \(s\) to \(v\).

The left-hand side of Figure 5.3 represents a worst-case run of \(\text{Dijkstra}\), containing 26 rule-schema applications. Among these are only 10 applications of \(\text{Reduce}\), which correspond to the 10 distance changes performed by the original algorithm.

The next proposition establishes the worst-case complexity of \(\text{Dijkstra}\), where we assume that input graphs satisfy the precondition of Proposition 7.

**Proposition 8 (Complexity of Dijkstra)** When started from a graph containing \(n\) nodes and \(e\) edges, \(\text{Dijkstra}\) terminates after \(O(n^2 + e)\) rule-schema applications.

**Proof.** The initialisation phase \(\text{Prepare}\) uses \(n\) rule-schema applications because the start node is relabelled once and then every other node is tagged. The body of the while-loop is executed \((n-1)\)-times because initially there are at most \(n-1\) 1-tagged nodes, and each execution of the body reduces this number by one. So the overall number of \(\text{Next}\)-applications is \(n - 1\), too.
Each execution of Min! takes at most \( n - 1 \) steps because there is only one 0-tagged node. Hence, there are at most \( (n - 1)^2 \) applications of Min overall. The total number of Reduce-applications is at most \( e \) since Reduce cannot be applied twice to the same edge. This is because Reduce is applied only to edges outgoing from the 0-tagged node, and this tag is removed by Next. Thus, a bound for the overall number of rule-schema applications is \( n + (n - 1) + (n - 1)^2 + e \), which is in \( O(n^2 + e) \). \[ \square \]

Note that if we forbid parallel edges in input graphs, then \( e \) is bounded by \( n^2 \) and hence the complexity of Dijkstra is \( O(n^2) \). The quadratic complexity of Dijkstra means a dramatic improvement on the exponential time-complexity of Simple_Dijkstra. Note that the running time of Dijkstra for the graph in Figure 5.4 is actually linear.

### 5.3.3 GP System Runs

We next demonstrate that our complexity results are indeed equivalent to the ones computed by the GP system using the program FastWorstCase introduced in Section 5.2. We compare the minimum and maximum number of graph changes performed in all program runs of Simple_Dijkstra and Dijkstra (see Figures 5.2 and 5.5) on the example graphs in Figures 5.4 and 5.3 with the number of rule-schema applications established in Propositions 6 and 8. Therefore, we first write two GP programs that generate the respective input graphs.

The GP program GenerateComplex in Figure 5.7 generates the input graph of Figure 5.4 for Simple_Dijkstra and Dijkstra. The input graph for GenerateComplex contains a single node labelled with a positive integer number \( k \) and produces an output graph with \( k + 1 \) nodes in the shape of the graph in Figure 5.4.

The GP program GenerateFiveNodes consists of a single rule that deletes the input node and inserts the input graph consisting of five nodes depicted in Figure 5.3.
main = Add; Count; Delete.

Add(x:int) =
\[ x \]
1
\[ \Rightarrow \]
\[ x_{-1} \]
1
\[ 0 \] 0

Count(x,y:int) =
\[ x_{-1} \]
1
\[ \Rightarrow \]
\[ x_{-1} \]
1
\[ 0 \] 0 0 0
\[ y \] 2
\[ 2y \] 2

where \( x > 0 \)

Delete(x:int) =
\[ x_{-1} \]
\[ \Rightarrow \]
\[ \emptyset \]

**Figure 5.7: The program GenerateComplex**

Running `Simple_Dijkstra` and `Dijkstra` on input graphs generated by `GenerateComplex` with the program `FastWorstCase` produces the results shown in Table 5.1. Here, the left table shows the results for `Simple_Dijkstra` and the right table the results for `Dijkstra`.

The number of answers for both programs grows at least exponentially and with it the runtime of the program. Thus, a run of `Simple_Dijkstra` on a graph with 6 nodes takes too long to present the result in Table 5.1. However, the number of graph changes, and the time-complexity classes resulting from these, directly compare to the ones given in Propositions 6 and 8. In the worst case, the number of graph changes performed by applying
**Simple_Dijkstra** to a graph generated by **GenerateComplex** grows exponentially and the number of graph changes performed by **Dijkstra** grows linearly.

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Answers</th>
<th>BC</th>
<th>WC</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>328</td>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>3603878</td>
<td>4</td>
<td>30</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Answers</th>
<th>BC</th>
<th>WC</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>24</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>924</td>
<td>14</td>
<td>25</td>
</tr>
<tr>
<td>5</td>
<td>89280</td>
<td>18</td>
<td>37</td>
</tr>
<tr>
<td>6</td>
<td>19810560</td>
<td>22</td>
<td>51</td>
</tr>
</tbody>
</table>

**Table 5.1:** Results of a system run of Simple_Dijkstra (left) and Dijkstra (right) on graphs generated by GenerateComplex. “Nodes” stands for the number of nodes in the graph, “Answers” for the number of generated results using the backtracking mode, “BC” for Best Case and “WC” for Worst Case, where entries in BC and WC are the number of graph changes in a best and worst-case execution, respectively.

The BC and WC values for **Dijkstra** are relatively high. This is because **Dijkstra**’s rule schemata perform many simple changes such as relabelling. If applied, the rule schema **Min**, for example, performs two graph changes by relabelling both nodes.

On the five node graph generated by **GenerateFiveNodes** and depicted in Figure 5.3, **Dijkstra** produces 13943808 graphs and needs 18 graph changes in the best case and 39 in the worst case. In comparison to our worst case run with 26 rule schema applications (5 times **Prepare**, 4 times **Tagged**, 0 + 3 + 2 + 1 times **Min**, 3 + 3 + 2 + 1 times **Reduce**, 4 times **Next** and 1 time **CleanUp**), 39 graph changes result because **Start**, **Reduce** and **CleanUp** each perform 1, **InitTags,** **Min** and **Next** each perform 2 and **Tagged** performs no graph change. Therefore, we get an overall number of $1 + 2 \times 4 + 0 + 2 \times 6 + 9 + 2 \times 4 + 1 = 39$ graph changes.

The results for a system run of **Simple_Dijkstra** on the five node graph generated by **GenerateFiveNodes** are missing. The high non-determinism
in the program results in an extremely large set of possible results (> 1.5
Millard) that cannot be computed in less than a few weeks time.

A result from this comparison of calculated rule-schema applications with
computed graph changes is that we may rely on the system to produce
exact measures. Therefore, we may use the GP system for these calcu-
lations, if needed. However, a major drawback of this approach is the
(at least) exponentially growing number of answers for complex graphs or
highly non-deterministic programs. This growth makes it impossible to run
FastWorstCase on graphs with more than 5 nodes in less than a week on
a 2.3 GHz Athlon machine with 1 GB main memory. Thus, in this case,
FastWorstCase is unfeasible.

5.4 Floyd-Warshall’s All-Pairs Shortest Path
Algorithm

The so-called all-pairs shortest path algorithm by Floyd and Warshall
[CLR00, Jun02] computes the shortest distances between each two nodes in
a graph whose edges are labelled with integers and which does not contain
cycles of negative lengths.5

Floyd-Warshall’s algorithm stores the shortest distances of a graph G in
the fields of a $|V_G| \times |V_G|$ matrix, where $d(u,v)$ with $u,v \in V_G$ denotes the
distance of a path from $u$ to $v$ in $G$. Initially, the matrix entry $d(u,v)$ is set
to $m_G(e)$ if there exists an edge $e \in E_G$ such that $s_G(e) = u$ and $t_G(e) = v$.
If $u = v$ the entry is set to 0 and all other entries are set to $\infty$. For each pair
of nodes $(u,v)$ and each other node $w$ in $G$, the algorithm checks if a shorter
distance via the node $w$ exists, that is if $d(u,v) > d(u,w) + d(w,v)$. If this
is the case, the entry is changed to $d(u,w) + d(w,v)$. After termination,
the matrix entries denote the values of the shortest distances between each
pair of nodes, i.e. $sd(u,v,V_G)$ for each pair $u, v \in V_G$.

---

5There is no shortest distance for paths through cycles of negative lengths.
5.4.1 The GP Program Simple_Floyd_Warshall

The resulting \(|V_G| \times |V_G|\) matrix of the Floyd-Warshall algorithm represents a complete graph in which edges are labelled with the shortest distance of all paths from their source node to their target node. If an entry in the resulting matrix is \(\infty\), there is no path from source to target in \(G\). Thus, the graph program will store computed distances directly as edge labels. If there is no edge between two nodes in the resulting graph, that corresponds to a matrix entry of \(\infty\) in the original algorithm.

The program is destructive because smaller distances may overwrite original edge labels. However, this problem can easily be solved by either copying edge labels into edge tags first and working on edge tags only (see, for example, the program Simple_Dijkstra in Section 5.3.1) or copying the graph and working on the copy (for an example of a copy program see [HP02a]). If the graph contains parallel edges, only the latter solution will work with our GP program as parallel edges are deleted.

Our first program for Floyd-Warshall’s algorithm, called Simple_Floyd_Warshall, is given in Figure 5.8. We assume that the program is started from an untagged, loop-free graph in \(G^\alpha(\mathcal{L})\) whose nodes and edges are labelled with integers and which does not contain cycles of negative length. Note that we are not interested in the particular labels of nodes or their types in the input graph as the algorithm is only concerned with edges. For reasons of simplicity, we assume the labels to be integers.

The command \texttt{Delete!} deletes one of two parallel edges as long as possible, keeping the edge with smaller label. Therefore, the result of \texttt{Delete!} is a simple graph. The rule schema \texttt{Add} inserts an edge \(e\) from node \(u\) to \(v\) with label \(z\) whenever no such edge exists and there exists a direct path from \(u\) to \(v\) with distance \(z\). Hence, \texttt{Add} computes a transitive graph. The rule schema \texttt{Reduce} reduces an edge label whenever a shorter distance is found.
main = Delete!; \{Add,Reduce\}!.

Delete(x,y,j,k:int) =

\[
\begin{array}{c}
\text{k} \\
1
\end{array} \xrightarrow{x} \begin{array}{c}
\text{j} \\
2
\end{array} \quad \Rightarrow \quad \begin{array}{c}
\text{k} \\
1
\end{array} \xrightarrow{x} \begin{array}{c}
\text{j} \\
2
\end{array}
\]

where not (y < x)

Add(x,y,j,k,l:int) =

\[
\begin{array}{c}
\text{k} \\
1
\end{array} \xrightarrow{x} \begin{array}{c}
\text{j} \\
2
\end{array} \xrightarrow{y} \begin{array}{c}
\text{l} \\
3
\end{array} \quad \Rightarrow \quad \begin{array}{c}
\text{k} \\
1
\end{array} \xrightarrow{x} \begin{array}{c}
\text{j} \\
2
\end{array} \xrightarrow{y} \begin{array}{c}
\text{l} \\
3
\end{array}
\]

where not edge(2,3)

Reduce(x,y,z,j,k,l:int) =

\[
\begin{array}{c}
\text{k} \\
1
\end{array} \xrightarrow{x} \begin{array}{c}
\text{j} \\
2
\end{array} \xrightarrow{y} \begin{array}{c}
\text{l} \\
3
\end{array} \quad \Rightarrow \quad \begin{array}{c}
\text{k} \\
1
\end{array} \xrightarrow{x} \begin{array}{c}
\text{j} \\
2
\end{array} \xrightarrow{y} \begin{array}{c}
\text{l} \\
3
\end{array}
\]

where z > (x + y)

Figure 5.8: The program Simple_Floyd_Warshall
**Proposition 9 (Termination of Simple_Floyd_Warshall)** Let $G$ be an untagged, loop-free graph in $\mathcal{G}^i(\mathcal{L})$ whose nodes and edges are labelled with integers and which does not contain cycles of negative length. When started from $G$, Simple_Floyd_Warshall terminates.

**Proof.** Delete! terminates because it reduces the number of parallel edges in each application. Each application of Add reduces the number of nodes not connected by a direct edge. As Reduce does not delete edges, Add terminates after a finite number of applications. Thus, there is a $k$ such that only Reduce is applicable to graphs $G_i$ with $i \geq k$. Because we do not allow cycles of negative length, there exists a shortest distance $sd(u,v, V_G)$ from $u$ to $v$ in $G$. Thus, the reduction of edge labels by Reduce is bounded. For every $j \geq k$ and every edge $e$ in $E_{G_j}$, $m_{G_j}(e) \geq sd(s_G(e), t_G(e), V_G)$ and for every step $G_j \Rightarrow_{\text{Reduce}} G_{j+1}$, $\#G_j \geq \#G_{j+1}$, where $\#G_i = \sum_{e \in G_i} m_{G_i}(e)$. Hence, Simple_Floyd_Warshall terminates. \hfill $\Box$

To prove correctness of Simple_Floyd_Warshall, we first prove the following claims. The first claim states that Delete preserves shortest distances.

**Claim 2** Let $G$ be an untagged, loop-free graph in $\mathcal{G}^i(\mathcal{L})$ whose nodes and edges are labelled with integers and let $G \Rightarrow^k_{\text{Delete}} H$. Then for each pair of nodes $u$ and $v$ in $G$ $sd(u,v, V_G) = sd(u,v, V_H)$.

**Proof.** By induction on $k$. For $k = 0$ the property obviously holds because $H \cong G$. Let us now assume that the property holds for a derivation of length $k$ and let $G \Rightarrow^k_{\text{Delete}} H' \Rightarrow_{\text{Delete}} H$ be a derivation of length $k + 1$. Since Delete does not alter nodes, and by induction hypothesis we get $sd(u,v, V_G) = sd(u,v, V_{H'})$ for all $u,v \in V_G = V_{H'}$. Let us now assume that there are nodes $u,v$ in $H$ such that $sd(u,v, V_{H'}) \neq sd(u,v, V_H)$. We have to distinguish two cases: Case 1: $sd(u,v, V_{H'}) > sd(u,v, V_H)$. In this case Delete has to reduce edge weights which is not the case. Case 2: $sd(u,v, V_{H'}) < sd(u,v, V_H)$. Then there are edges $e_1$ and $e_2$ in $H'$ where we assume that $e_1$ is the edge deleted by Delete in derivation step $k + 1$. To satisfy the assumption $sd(u,v, V_{H'}) < sd(u,v, V_H)$, $e_1$ is an edge on the path from $u$ to $v$ in $G$ whose distance is $sd(u,v, V_{H'})$, and $m_{H'}(e_1) < m_{H'}(e_2)$. 


But $m_{H'}(e_1) < m_{H'}(e_2)$ contradicts the fact that the rule schema condition 
not $(m_{H'}(e_2) > m_{H'}(e_1))$ is satisfied. Hence, $sd(u, v, V_G) = sd(u, v, V'_{H'}) = 
\text{sd}(u, v, V_H).$ 
\hfill \Box

\textbf{Claim 3} Let $G$ be an untagged, loop-free graph in $\mathcal{G}^+(\mathcal{L})$ whose nodes and 
edges are labelled with integers and let $(\text{Simple\_Floyd\_Warshall}, G) \rightarrow^+ H$. Then $H$ is simple.

\textbf{Proof.} Obviously, all parallel edges are removed by \textsf{Delete!} and the condition of \textsf{Add} guarantees that \textsf{Add} does not insert parallel edges. As \textsf{Reduce} 
only relabels edges, the property holds for $H$. 
\hfill \Box

\textbf{Claim 4} Let $G$ be an untagged, loop-free graph in $\mathcal{G}^+(\mathcal{L})$ whose nodes and 
edges are labelled with integers and let $(\text{Simple\_Floyd\_Warshall}, G) \rightarrow^+ H$. Then $H$ is transitive.

\textbf{Proof.} It is sufficient to show that $H$ is transitive for paths of length 2 
because we can easily prove by induction on path lengths that the result is 
equivalent for arbitrary path lengths. Let us now assume there is a path 
e_1, e_2 of length 2 in $H$ but there is no edge $e$ in $H$ such that $s_H(e) = s_H(e_1)$ 
and $t_H(e) = t_H(e_2)$. Then \textsf{Add} is applicable which is a contradiction to the 
fact that \{\textsf{Add,Reduce}\}! terminated. Assume there is an edge $e$ in $H$ but 
no path from $s_H(e)$ to $t_H(e)$. This is a contradiction in itself because $e$ is a 
path of length 1 from $s_H(e)$ to $t_H(e)$. 
\hfill \Box

We next prove the correctness of \textsf{Simple\_Floyd\_Warshall}.

\textbf{Proposition 10 (Correctness of Simple\_Floyd\_Warshall)} Let $G$ be 
an untagged, loop-free graph in $\mathcal{G}^+(\mathcal{L})$ whose nodes and edges are labelled 
with integers and which does not contain cycles of negative length. When 
started from $G$, \textsf{Simple\_Floyd\_Warshall} produces a unique graph $H$ which 
is obtained from $G$ by deleting all edges and inserting an edge between each 
two nodes $u$ and $v$ for which a path from $u$ to $v$ in $G$ exists and labelling 
each such edge with the distance of the shortest such path.
Proof. As none of the rules alters nodes, we may assume that $V_G = V_H$.
By Claims 3 and 4, $H$ is simple and transitive. Thus, there is exactly one
dge from a node $u$ to a node $v$ in $H$ and whenever there is a path from
$u$ to $v$ in $H$, there is such an edge from $u$ to $v$. By Claim 2, \textbf{Delete}! is
shortest distance preserving, i.e. for a graph $G'$ and nodes $u, v \in V_{G'}$ with
$\langle \texttt{Delete}, G \rangle \rightarrow^{+} G'$, $sd(u, v, V_G) = sd(u, v, V_G')$. We now have to show
that \textbf{Add} and \textbf{Reduce} also keep these shortest distances, i.e. $sd(u, v, V_{G'}) =
\text{sd}(u, v, V_H)$, and that all edges in $H$ are labelled with the shortest distance
from source to target node in $G$. The former is easily observed as neither
\textbf{Add} nor \textbf{Reduce} delete any edges and existing edge labels are only reduced
if a smaller path of length 2 is found.

To prove the latter, we first prove the following invariant property.

**Invariant 3** Let $\langle \texttt{Simple\_Floyd\_Warshall}, G \rangle \rightarrow^{+} H$, then for all edges
$e$ in $H$ with $m_H(e) = n$ there is a path from $s_G(e)$ to $t_G(e)$ of length $n$ in
$G$.

Proof. For all edges in $G$ and in $G'$ with $\langle \texttt{Delete}, G \rangle \rightarrow^{+} G'$, the state-
ment is trivially satisfied. By hypothesis, we may assume that edge labels
in the left-hand side of \textbf{Add} and \textbf{Reduce} are lengths of paths from $s_G(e)$
to $t_G(e)$ in $G$. Then \textbf{Reduce} only changes a distance to a smaller one in
$G$. Thus, every application of \textbf{Reduce} preserves the property. We therefore
only have to consider the application of \textbf{Add} that inserts edges.

Assume the property holds for $k$ inserted edges and let $e$ be the $k+1$ edge
added to $G'$ by \textbf{Add}. Let furthermore $e_1$ and $e_2$ be the edges matched in
the left hand side of \textbf{Add}.

\textit{Case 1:} $e_1, e_2 \in E_G$. Then obviously, \textbf{Add} inserts $e$ with a label $n$ where $n$
is the length of a path from $s_G(e)$ to $t_G(e)$ in $G$.

\textit{Case 2:} $e_1, e_2$ are two of the $k$ inserted edges. By hypothesis, the statement
holds for $e_1$ and $e_2$ and the property is obviously preserved by \textbf{Add}.

Because the other cases are just mixed cases of \textit{Case 1} and \textit{Case 2}. Hence,
the property holds for every edge $e$ in $H$. \hfill \box
Assume now \( n \) is not the length of a shortest path from \( s_G(e) \) to \( t_G(e) \) in \( G \). Since \textbf{Simple\_Floyd\_Warshall} has terminated and \( H \) is transitive and simple, there must be edges \( e_1 \) and \( e_2 \) such that \( s_H(e_1) = s_H(e), \ t_H(e_1) = s_H(e_2), \ t_H(e_2) = t_H(e) \) and \( m_H(e_1) + m_H(e_2) < m_H(e) \) where we assume without loss of generality that \( e \) is the edge in the smallest such triangular situation (otherwise \textbf{Add} is applicable or \( e_1 \) or \( e_2 \) are not labelled with the shortest paths). But then \textbf{Reduce} is applicable which is a contradiction to the fact that \textbf{Simple\_Floyd\_Warshall} terminated. \( \square \)

Although the graph program \textbf{Simple\_Floyd\_Warshall} is simple and produces a unique result, it is also highly non-deterministic. This possibly results in a high number of \textbf{Reduce} rule schema applications in the worst case. We conjecture the time-complexity of \textbf{Simple\_Floyd\_Warshall} to be at least polynomial if not exponential. However, we could not find an input graph that would confirm our conjecture. Input graphs need to have a particular labelling and structure such that \textbf{Reduce} is applicable an exponential number of times to the same edge as in the example of \textbf{Simple\_Dijkstra} in Section 5.3.1. We were unable to find such graph.

As time-complexity for \textbf{Simple\_Floyd\_Warshall} is difficult to estimate, we model the GP program closer to the original algorithm in the next section.

### 5.4.2 The GP Program \textbf{Floyd\_Warshall}

We next introduce the GP program \textbf{Floyd\_Warshall} that represents a more deterministic version of the program \textbf{Simple\_Floyd\_Warshall}. We model the program close to the original algorithm introduced in Section 5.4. \textbf{Floyd\_Warshall} is also close to a program based on Transformation Units (see [Kus00]) presented in [KK99].

The program \textbf{Floyd\_Warshall} is shown in Figure 5.9. First, the input graph is transformed into a simple, transitive graph \( G' \) where all nodes are tagged with 0 by \( \langle \text{Delete!}; \text{Add!}; \text{Tag!}, \ G \rangle \rightarrow^+ G' \). Then, instead of applying \textbf{Reduce} non-deterministically to any matching position in the graph
— as done in Simple_Floyd_Warshall —, distance values are reduced in a controlled way using the while loop. In each iteration of while in Floyd_Warshall, Pick selects a unique node $w$ with tag 0 and relabels its tag with 1. Then, Reduce checks if the distance of a path from $u$ to $v$ through $w$ is shorter than the distance stored in the label of the direct edge $e$ from $u$ to $v$. If this is the case, $m_{G'}(e)$ is changed to this new distance. Thus, the set of inner nodes on a path is extended by one in each iteration of the while loop until all nodes are in the set. This is achieved by changing a node’s tag from 0 to 2 by the application of while.

The rule schema Delete is equivalent to the rule schema Delete in Simple_Floyd_Warshall. Thus, we may assume by Claim 2 that Delete in Floyd_Warshall also preserves shortest distances.

Figure 5.11 shows a worst-case run of Floyd_Warshall. Delete first deletes two parallel edges where the edges with smaller distance are preserved respectively. Then, Add adds 4 edges to construct a transitive graph. These edges are labelled with the length of some path from their source to their target node. Note that Add is applied non-deterministically. In this example, Add choose the longer paths of length 5. Then, Tag is applied 4 times tagging all nodes with 0. After this preparation, the second part of the program is started. In each iteration of the while loop, the 0-tag of one node is changed to 2 and new shortest distances are computed. Note that the number of rule applications during the execution of while in this example is not dependent on the sequence in which the nodes are chosen by Pick. However, the non-determinism of Pick can be relevant for the actual number of rule applications in other examples.

Running Floyd_Warshall on the input graph in Figure 5.11 with the GP system results in 780337152 answers, 30 graph changes in the best case and 34 graph changes in the worst case. In each application of Delete and Add two graph changes are performed and every other rule application corresponds to one graph change. In a worst case run, Reduce is applied 6 times. Once to each of the four 5-labelled edges inserted by Add and once
main = PrepareFW;

while Tagged do (Pick; Reduce!; Next); Untag! .

Tagged(x:int) =

\[ x_0 \quad \Rightarrow \quad x_0 \]

Pick(x:int) =

\[ x_0 \quad \Rightarrow \quad x_1 \]

Reduce(x,y,z,j,k,l,r,s:int) =

\[ \begin{align*}
&x \quad \downarrow \quad y \\
&\quad \downarrow \quad z \\
&\quad \downarrow \quad l_s \\
\end{align*} \quad \Rightarrow \quad \begin{align*}
&x+y \quad \downarrow \quad y \\
&\quad \downarrow \quad l_s \\
\end{align*} \]

where \( z > (x + y) \)

Next(x:int) =

\[ x_1 \quad \Rightarrow \quad x_2 \]

Untag(x:int) =

\[ x_2 \quad \Rightarrow \quad x \]

Figure 5.9: The program Floyd_Warshall
PrepareFW = Delete!; Add!; Tag!.

\[\text{Delete}(x, y, j, k: \text{int}) =\]

\[
\begin{array}{c}
\begin{array}{c}
\k
\end{array} \\
1
\end{array}
\quad
\begin{array}{c}
\begin{array}{c}
\j
\end{array} \\
2
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\k
\end{array} \\
1
\end{array}
\quad
\begin{array}{c}
\begin{array}{c}
\j
\end{array} \\
2
\end{array}
\]
\]

where not \((y < x)\)

\[\text{Add}(x, y, j, k, l: \text{int}) =\]

\[
\begin{array}{c}
\begin{array}{c}
\k
\end{array} \\
1
\end{array}
\quad
\begin{array}{c}
\begin{array}{c}
\j
\end{array} \\
2
\end{array}
\quad
\begin{array}{c}
\begin{array}{c}
\l
\end{array} \\
3
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\k
\end{array} \\
1
\end{array}
\quad
\begin{array}{c}
\begin{array}{c}
\j
\end{array} \\
2
\end{array}
\quad
\begin{array}{c}
\begin{array}{c}
\l
\end{array} \\
3
\end{array}
\]
\]

where not \(\text{edge}(2, 3)\)

\[\text{Tag}(x: \text{int}) =\]

\[
\begin{array}{c}
\begin{array}{c}
x
\end{array} \\
1
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
_0
\end{array} \\
1
\end{array}
\]

\[\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\q
and to the 4-labelled edges of the input graph. In a best case run, \texttt{Reduce} is applied 2 times since \texttt{Add} inserts 2-labelled edges.

![Diagram](image)

**Figure 5.11:** A worst-case run of \texttt{Floyd Warshall}

The next proposition establishes the worst-case complexity of \texttt{Floyd Warshall}, where we assume that input graphs satisfy the precondition of Proposition 10.

**Proposition 11 (Complexity of Floyd Warshall)** When started from an untagged, loop-free graph in $G^x(L)$ whose nodes and edges are labelled with integers, which does not contain cycles of negative length, that contains $n$ nodes and $e$ edges, \texttt{Floyd Warshall} terminates after $O(e + n^3)$ rule-schema applications.
Proof. Delete is applied at most \(e - 1\) times because this is the maximum number of parallel edges in a graph (all \(e\) edges between two nodes) and Delete deletes all parallel edges. Add is applied at most \(n \times (n - 1)\) times because Add adds edges where no such edge exists and \(n \times (n - 1)\) is the maximum number of edges in a simple (complete) graph. Tag is applied once to every node and thus is applied \(n\) times overall. The body of the while-loop is executed \(n\)-times because initially there are \(n\) 0-tagged nodes, and each execution of the body reduces this number by one by first applying Pick and then Next. So the overall number of Pick- and Next-applications respectively is \(n\). Each execution of Reduce takes at most \((n - 1)(n - 2)\) steps because there is at most one 1-tagged node connected to at most \(n - 1\) other nodes which can be connected to \(n - 2\) other nodes to form the matched triangular situation and Reduce can only be applied once to each such situation because of the rule-schema condition. Hence, there are at most \(n(n - 1)(n - 2)\) applications of Reduce overall. Untag is applied once to every node and therefore is applied \(n\) times. Thus, an upper bound for the overall number of rule-schema applications is \(e - 1 + n \times (n - 1) + n + n(n - 1)(n - 2) + 2n + n = e + n^3 - 2n^2 + 5n - 1\), which is in \(O(e + n^3)\). □

To prove the correctness of Floyd_Warshall, we first show some properties. The following observation states that shortest distances can be presented by the sum of two shortest distances splitting the original path at an inner node.

Observation 5.4.1 Let \(p\) be a shortest path from \(u\) to \(v\) in a graph \(G\) in \(\mathcal{G}^s(\mathcal{L})\) with inner nodes in \(I \subseteq V_G\) and let \(w\) be a node on \(p\) with \(u \neq w \neq v\). Then \(\text{sd}(u, v, I) = \text{sd}(u, w, I - \{w\}) + \text{sd}(w, v, I - \{w\})\).

Claim 5 Let \(G\) be an untagged, loop-free graph in \(\mathcal{G}^s(\mathcal{L})\) whose nodes and edges are labelled with integers and that does not contain cycles of negative length and let \((\text{Floyd_Warshall}, G) \rightarrow^+ H\). Then \(H\) is simple.

Proof. The graph \(G'\) with \((\text{Delete!}, G) \rightarrow^+ G'\) is simple because Delete deletes all parallel edges. As no other rule adds edges except for Add and Add does not add parallel edges because of its condition, \(H\) is simple. □
Claim 6 Let \( G \) be an untagged, loop-free graph in \( \mathcal{G}(\mathcal{L}) \) whose nodes and edges are labelled with integers and that does not contain cycles of negative length and let \( \langle \text{Floyd-Warshall}, G \rangle \to^+ H \). Then \( H \) is transitive.

Proof. As none of the rules except Delete deletes edges, Delete only deletes parallel edges and Add is the only rule to insert edges, \( H \) is transitive if Add produces a transitive graph. The rule schema Add is equivalent to the rule schema Add in Simple Floyd-Warshall. Therefore, the proof is analogous to the proof of Claim 4 and hence, \( H \) is transitive. \( \square \)

Invariant 4 Let \( G \) be an untagged, loop-free graph in \( \mathcal{G}(\mathcal{L}) \) whose nodes and edges are labelled with integers and that does not contain cycles of negative length. Let \( G \Rightarrow^k_{\text{add}} H \). Then \( H \) is distance preserving.

Proof. By induction on \( k \). For \( k = 0 \), \( H \) is obviously distance preserving because \( G \cong H \). Let us now assume \( H \) is distance preserving for a derivation of length \( k \) and let \( G \Rightarrow^k_{\text{add}} H' \Rightarrow_{\text{add}} H \) be a derivation of length \( k + 1 \). Then the left hand side of Add is applied to two edges \( e_1, e_2 \in E_{H'} \) where \( m_{H'}(e_1) = x, m_{H'}(e_2) = y, \) and \( t_{H'}(e_1) = s_{H'}(e_2) \) and \( H \) contains the edge \( e \) with \( s_H(e) = s_H(e_1), t_H(e) = t_H(e_2), \) and \( m_H(e) = x + y \). By induction hypothesis, \( H' \) is distance preserving. Thus, there is a path from \( s_H(e_1) \) to \( t_H(e_1) \) of length \( x \) and a path from \( s_H(e_2) \) to \( t_H(e_2) \) of length \( y \) in \( G \). Hence, there is a path from \( s_H(e) = s_H(e_1) \) to \( t_H(e) = t_H(e_2) \) of length \( x + y \) in \( G \) and therefore \( H \) is distance preserving. \( \square \)

Invariant 5 Let \( H_0 \) be a loop-free, simple and transitive graph in \( \mathcal{G}(\mathcal{L}) \) whose nodes and edges are labelled with integers, whose nodes are tagged with 0 and that does not contain cycles of negative length. Let furthermore \( I_k \) denote the \( k \)-th iteration of the body Pick; Reduce!; Next of the while loop and \( H_k \) the resulting graph of \( I_k \). Then for every edge \( e \) in \( H_k, m_{H_k}(e) = \text{sd}(s_{H_k}(e), t_{H_k}(e), \text{U}(H_k)) \) where \( \text{U}(H_k) \) is the set of 2-tagged nodes in \( H_k \).

Proof. By induction on \( k \). The property clearly holds for \( H_0 \) because \( H_0 \) is simple, \( \text{U}(H_0) = \emptyset \), and every edge label in \( H_0 \) represents the shortest distance of paths of length 1. Thus,
\(m_{H_0}(e) = \text{sd}(s_{H_0}(e), t_{H_0}(e), \emptyset)\) for all \(e \in E_{H_0}\). Assume now \(m_{H_k}(e) = \text{sd}(s_{H_k}(e), t_{H_k}(e), U(H_k))\) for some \(k\) and all \(e\) in \(E_{H_k}\) and let \(w\) be the node selected by \text{Pick} in iteration \(k + 1\), i.e. \(w \notin U(H_k)\) and \(U(H_{k+1}) = U(H_k) \cup \{w\}\). For every edge \(e\) in \(H_{k+1}\) we have to distinguish two cases:

\textbf{Case 1:} \(m_{H_k}(e) = m_{H_{k+1}}(e)\). The only rule altering edges in each iteration is \textbf{Reduce}. Thus, \textbf{Reduce} was not applied to the triangular pattern depicted in the left hand side of \textbf{Reduce} where \(e\) is the edge labelled by \(z\). We distinguish two cases: \textbf{Case a:} The triangular pattern \(T\) does not exist. Then either edge \(e_1\) in \(T\) with \(s_{H_k}(e_1) = s_{H_k}(e), t_{H_k}(e_1) = w\), and \(m_{H_k}(e_1) = x\) or edge \(e_2\) in \(T\) with \(s_{H_k}(e_2) = w, t_{H_k}(e_2) = t_{H_k}(e)\), and \(m_{H_k}(e_2) = y\) does not exist or both do not exist. Note that the edges \(e_1, e_2\) are unique because \(H_k\) is simple (\(H_0\) is simple and none of the rules inserts edges) and \(w\) is unique in each iteration. Let now \(e_1\) be the missing edge. Since \(H_0\) is transitive, \(H_k\) is transitive (\(H_0\) is transitive and none of the rules deletes edges). Thus, there is no path from \(s_{H_k}(e)\) to \(w\) in \(H_k\) and therefore no path from \(s_{H_k}(e)\) to \(t_{H_k}(e)\) with inner node \(w\) in \(H_k\). Thus, \(m_{H_k}(e) = \text{sd}(s_{H_k}(e), t_{H_k}(e), U(H_k)) = \text{sd}(s_{H_{k+1}}(e), t_{H_{k+1}}(e), U(H_{k+1})) = m_{H_{k+1}}(e)\). \textbf{Case b:} The triangular pattern \(T\) exists but the condition is not satisfied. In this case, we get the following equation:

\[
x + y = \text{def.} e_1, e_2 \quad m_{H_k}(e_1) + m_{H_k}(e_2) \\
= \text{hyp.} \quad \text{sd}(s_{H_k}(e_1), t_{H_k}(e_1), U(H_k)) \\
+ \text{sd}(s_{H_k}(e_2), t_{H_k}(e_2), U(H_k)) \\
= \text{def.} w \quad \text{sd}(s_{H_k}(e_1), w, U(H_k)) + \text{sd}(w, t_{H_k}(e_2), U(H_k)) \\
= \text{def.} e \quad \text{sd}(s_{H_k}(e), w, U(H_k)) + \text{sd}(w, t_{H_k}(e), U(H_k)) \\
= \text{obs.} 5.4.1 \quad \text{sd}(s_{H_{k+1}}(e), t_{H_{k+1}}(e), U(H_{k+1})) \geq \text{Cond} = \text{def.} z \\
= \text{def.} e \quad m_{H_k}(e) \\
= \text{hyp.} \quad \text{sd}(s_{H_k}(e), t_{H_k}(e), U(H_k))
\]

Thus, the original label of \(e\) has to be kept which is guaranteed by non-applicability of \textbf{Reduce}. 
**Case 2**: \( m_{H_k}(e) \neq m_{H_{k+1}}(e) \). Since **Reduce** is the only rule which alters edges, **Reduce** was applied to \( e \). Let \( e_1, e_2 \) be the unique edges with \( s_{H_k}(e_1) = s_{H_k}(e), t_{H_k}(e_1) = w, s_{H_k}(e_2) = w, t_{H_k}(e_2) = t_{H_k}(e), m_{H_k}(e_1) = x, \) and \( m_{H_k}(e_2) = y \). By the **Reduce** condition and the induction hypothesis, we get the following equation:

\[
x + y = \text{def.} e_1, e_2 \quad m_{H_k}(e_1) + m_{H_k}(e_2)
\]

\[
= \text{hyp.} \quad \text{sd}(s_{H_k}(e_1), t_{H_k}(e_1), U(H_k)) \\
+ \text{sd}(s_{H_k}(e_2), t_{H_k}(e_2), U(H_k))
\]

\[
= \text{def.} w \quad \text{sd}(s_{H_k}(e_1), w, U(H_k)) + \text{sd}(w, t_{H_k}(e_2), U(H_k))
\]

\[
= \text{def.} e \quad \text{sd}(s_{H_k}(e), w, U(H_k)) + \text{sd}(w, t_{H_k}(e), U(H_k))
\]

\[
= \text{obs.} 5.4.1 \quad \text{sd}(s_{H_{k+1}}(e), t_{H_{k+1}}(e), U(H_{k+1}))
\]

\[
< \text{cond.} \quad z
\]

\[
= \text{def.} e \quad m_{H_k}(e)
\]

\[
= \text{hyp.} \quad \text{sd}(s_{H_k}(e), t_{H_k}(e), U(H_k))
\]

Since \( m_{H_k}(e) \) is changed to \( \text{sd}(s_{H_{k+1}}(e), t_{H_{k+1}}(e), U(H_{k+1})) \) by **Reduce**, \( m_{H_{k+1}}(e) = \text{sd}(s_{H_{k+1}}(e), t_{H_{k+1}}(e), U(H_{k+1})) \).

\[\square\]

**Proposition 12 (Correctness of Floyd_Warshall)** Let \( G \) be an un-tagged, loop-free graph in \( G^*(\mathcal{L}) \) whose nodes and edges are labelled with integers and which does not contain cycles of negative length. \( \langle \text{Floyd_Warshall}, G \rangle \to^+ H \) produces a unique graph \( H \) which is obtained from \( G \) by deleting all edges and inserting edges if there is a path from \( s_G(e) \) to \( t_G(e) \) in \( G \) and \( m_H(e) \) is the distance of the shortest such path.

**Proof.** By Claims 5 and 6 we get that \( H \) is simple and transitive. Thus, for all nodes \( u \) and \( v \) in \( H \) there is a simple edge \( e \) from \( u \) to \( v \) if there is a path from \( u \) to \( v \) in \( H \). It remains to show that for every edge \( e \) with \( m_H(e) \) is the distance of a shortest path from \( s_G(e) \) to \( t_G(e) \) in \( G \). Let now \( \langle \text{Delete!}; \text{Add!}; \text{:Tag!}, G \rangle \to^+ G' \) and \( \langle \text{while Tagged do Pick; Reduce!; Next; Untag} \rangle \to^+ H \). Combining Claim 2 and Invariant 4 we get that for every two nodes \( u \) and \( v \), \( \text{sd}(u, v, V_G) = \text{sd}(u, v, V_{G'}) \) (Invariant 2 and \textbf{Add} preserves the property because \textbf{Add} does
not alter existing edges) and every edge in $G'$ is labelled with the length of a path in $G$ (Invariant 4 and Tag does not alter edges). Tag! tags all nodes by 0. Thus $G'$ is a proper input graph for Invariant 5. By Invariant 5 we get that for every edge $e$ in $H$, $m_H(e) = sd(s_H(e), t_H(e), V_H)$. By the above observation that shortest distances in $G$ are preserved and still existent in $G'$, every edge in $G'$ is labelled with the distance of a path in $G$. By Invariant 5, we then get that all edges in $H$ are labelled with the distance of a shortest path in $G$.

The introduced graph program Floyd_Warshall is similar to the one in [KK99]. The main difference to [KK99] is that we introduce a clearer partition of program parts and tasks in our approach. Where we use Delete and Add to construct a simple and transitive graph as a preparation for the actual program, Kreowski and Kuske mix these concepts with the concept of the rule schema Reduce as we have done in Simple_Floyd_Warshall. The Transformation Unit (see [Kus00]) sums inserts an edge with label $x + y$ to form the triangular situation if no such edge with label $x + y$ exists yet. Thus, parallel edges are inserted which might be labelled with a greater distance value than existing ones. These are then merged to a new edge labelled with the minimum of all parallel edges by the Unit minimum. By splitting this concept of intertwined Units into clearly formulated rule schemata, our approach is simpler and yields less rule applications in general.6 In the example in [KK99] the Unit shortestpath needs at most 6 sum and 6 minimum applications whereas Floyd_Warshall needs at most 1 Delete, 1 Add and 3 Reduce applications. Regarding the example in Figure 5.11, shortestpath needs at most 22 minimum and 24 sum applications whereas Floyd_Warshall needs at most 2 Delete, 4 Add and 6 Reduce applications. Thus, by splitting the program into preparation and distance reduction phases, we are able to reduce the actual number of rule applications and gain a better comprehension of the execution.

6Note that the worst case class for Floyd_Warshall is identical to the one given in [KK99].
Although Simple_Floyd_Warshall is a simple GP program, time-complexity is difficult to measure as a program run is heavily dependent on the structure and labelling of the input graph. We tried to get an indication of the complexity class for Simple_Floyd_Warshall by running Simple_Floyd_Warshall on a graph with $n$ nodes of the shape exemplified in Figure 5.11 (pairwise opposite edges labelled in one direction with 1 and in the other alternating with 1 and 4). However, because of the high non-determinism of the program, the number of answers grows worse than exponential (1 answer for 3 nodes, 542128 answers for 4 nodes) and thus, running Simple_Floyd_Warshall on a graph with 5 nodes takes more than just a few weeks on a computer. Therefore, we cannot get an indication of the complexity class of Simple_Floyd_Warshall by the GP system.

5.5 Vertex Colouring

In this section, we introduce a non-deterministic algorithm for a vertex colouring problem (see [Jun02]). Note that the original version of the GP program, Propositions 13 and 14 and Corollary 1 are the original work of Dr. Detlef Plump for an extended version of the joint paper [PS04]. We show this program and discussion here as it completes the work in discussing a GP program with a non-deterministic result.

Given a graph $G$ and an infinite set of colours, a vertex colouring of $G$ is a graph $H$ obtained from $G$ by assigning each node in $G$ a colour such that no two nodes connected by a direct edge in $H$ have the same colour.

We next present the GP program VertexColouring that represents a non-deterministic solution to the vertex-colouring problem. Colours are represented by positive integers and are stored in node tags.

The program in Figure 5.12 produces non-deterministically a vertex colouring of the untagged, loop-free and integer labelled input graph. In general, we are not interested in the particular type of labels in the input graph as the algorithm is concerned with the graph structure, only. For reasons of simplicity, we assume the labels to be integers.
main = InitTags!; Increase!

Increase = \{Inc_1, Inc_2\}.

InitTags(x:int) =

\[ \begin{array}{c}
\text{ } \\
x \\
1 \\
\end{array} \quad \Rightarrow \quad \begin{array}{c}
\text{ } \\
x_1 \\
1 \\
\end{array} \]

Inc_1(x,y,z,k:int) =

\[ \begin{array}{c}
x_k \\
1 \\
\end{array} \quad y \quad \begin{array}{c}
z_k \\
2 \\
\end{array} \quad \Rightarrow \quad \begin{array}{c}
x_k \\
1 \\
\end{array} \quad y \quad \begin{array}{c}
z_{k+1} \\
2 \\
\end{array} \]

Inc_2(x,y,z,k:int) =

\[ \begin{array}{c}
x_k \\
1 \\
\end{array} \quad y \quad \begin{array}{c}
z_k \\
2 \\
\end{array} \quad \Rightarrow \quad \begin{array}{c}
x_{k+1} \\
1 \\
\end{array} \quad y \quad \begin{array}{c}
z_k \\
2 \\
\end{array} \]

**Figure 5.12:** The program VertexColouring
**VertexColouring** first tags every node in the graph with 1 by *InitTags*. Then *Inc*¹ and *Inc*² increase the tag of one node if two directly connected nodes have the same tag. We need both rule schemata to match all possible such situations. The program is non-deterministic as it computes one out of many possible colourings in a graph.

Consider, for example, the run in Figure 5.13 where the top left graph is the input graph. We have chosen here, the worst-case run of *VertexColouring* that needs 6 *Increase* applications for a colouring in which every node has a different colour. The best run for this input graph only needs 2 *Increase* applications to produce the colouring shown in Figure 5.14. These results are confirmed by the GP system. Running *VertexColouring* on the four node graph in Figure 5.13 with FastWorstCase results in 9600 answers and a best case of 6 and worst case of 10 graph changes where rule applications correspond to graph changes. This matches the 4 applications of *InitTags* together with 2 applications of *Increase* in the best case and 4 applications of *InitTags* together with 6 applications of *Increase* in the worst case.

We next prove partial correctness of *VertexColouring*.

**Proposition 13 (Partial correctness)** For every untagged, loop-free, integer labelled graph *G* in *G*²(ℒ), if ⟨*VertexColouring*, *G*⟩ →⁺ *H*, then *H* is a vertex colouring of *G*.

**Proof.** It is easy to check that *H* can be obtained from *G* by adding a tag to each node. Moreover, for every edge *e* in *H*, the source and target nodes of *e* have different tags as otherwise *Increase* could be applied to *e*. □

To prove that *VertexColouring* is terminating, we will make use of the following invariance property.

**Proposition 14 (Invariant)** Let ⟨*InitTags*!, *G*⟩ →⁺ *G'* and *G'* →⁺ H where *G* is untagged, loop-free and integer labelled. Then for each node *v* in *H*, tag_H(v) = 1 or tag_H(v) > 1 with tag_H(v) denoting the tag of the node label of *v* and there is a node *w* such that tag_H(w) = tag_H(v) − 1.
Figure 5.13: An example run of VertexColouring
Figure 5.14: An output for a best-case run of VertexColouring

Proof. We proceed by induction on the length $n$ of the derivation $G' \Rightarrow^* \{\text{Inc}_1, \text{Inc}_2\} H$. If $n = 0$, then $G' \cong H$ and hence every node in $H$ is tagged with 1. Let now $G' \Rightarrow^* \{\text{Inc}_1, \text{Inc}_2\} \overline{H} \Rightarrow^* \{\text{Inc}_1, \text{Inc}_2\} H$ such that $G' \Rightarrow^* \{\text{Inc}_1, \text{Inc}_2\} \overline{H}$ contains less than $n$ steps. By induction hypothesis, $\overline{H}$ satisfies the proposition. Let $v$ and $w$ be the nodes in $\overline{H}$ that are in the image of the left-hand side of the rule applied in $\overline{H} \Rightarrow^* \{\text{Inc}_1, \text{Inc}_2\} H$. Without loss of generality, let $v$ be the node whose tag is incremented by one. Consider any node $p$ in $H$. Case 1: $p = v$. Then $\mathit{tag}_H(p) > \mathit{tag}_{\overline{H}}(v) \geq 1$ and $\mathit{tag}_H(w) = \mathit{tag}_{\overline{H}}(w) = \mathit{tag}_{\overline{H}}(v) = \mathit{tag}_H(v) - 1$. Case 2: $p \neq v$. Then $\mathit{tag}_H(p) = \mathit{tag}_{\overline{H}}(p) \geq 1$. Suppose that $\mathit{tag}_{\overline{H}}(p) > 1$. Then there is a node $p'$ in $\overline{H}$ such that $\mathit{tag}_H(p') = \mathit{tag}_{\overline{H}}(p) - 1$. Case 2.1: $p' = v$. Then $\mathit{tag}_H(w) = \mathit{tag}_{\overline{H}}(w) = \mathit{tag}_{\overline{H}}(v) = \mathit{tag}_H(v) - 1$. Case 2.2: $p' \neq v$. Then $\mathit{tag}_H(p') = \mathit{tag}_{\overline{H}}(p') = \mathit{tag}_{\overline{H}}(p) - 1$. Thus $H$ satisfies the invariant. □

Corollary 1 (Upper Bound for Tags) Let $G$ be an untagged, loop-free, integer labelled graph in $G^\mathcal{E}(\mathcal{L})$ and let $\langle \text{InitTags}!, \ G' \rangle \rightarrow^{+} G'$ and $G' \Rightarrow^* \{\text{Inc}_1, \text{Inc}_2\} H$. Then for each node $v$ in $H$, $\mathit{tag}_H(v) \leq |V_G|$.

Proof. Let $|V_H| = m$. Since $\text{InitTags}$ and $\{\text{Inc}_1, \text{Inc}_2\}$ preserve the number of nodes, we have $|V_G| = m = |V_H|$. Suppose that there is a node $v$ in $H$ such that $\mathit{tag}_H(v) > m$. Let $\mathit{tag}_H(v) = n$. By Proposition 14 there are nodes $v_1, v_2, \ldots, v_n$ such that for $i = 1, \ldots, n$, $\mathit{tag}_H(v_i) = i$. Hence $|V_H| \geq n$, contradicting the fact that $|V_H| = m$. □
Proposition 15 (Termination and Complexity) Let $G$ be an un-
tagged, loop-free, integer labelled graph in $G^e(\mathcal{L})$. Then VertexColouring applied to $G$ terminates after $O(|V_G|^2)$ rule-schema applications.

Proof. Termination of InitTags! follows from the fact that every application reduces the number of untagged nodes. As there are $|V_G|$ untagged nodes in $G$, InitTags is applied $|V_G|$ times. Termination of Increase! follows from the fact that the application of Increase! is bounded because the number of different colours in the output graph is bounded by the number of nodes in $G$ as proven in Corollary 1 and hence, Increase can be applied at most $|V_G|^2$ times ($|V_G|$ times to each node). Hence, the proposition holds.$\blacksquare$

5.6 Related Work

In this chapter, we presented three case studies in the domain of graph algorithms. Case studies in our sense include a formal discussion on termination, correctness and (possibly) complexity of the algorithmic solution at hand. Surprisingly, there seems to be hardly any work on studying graph algorithms in the framework of graph transformation languages. We are only aware of a case study on Floyd-Warshall’s all-pairs shortest-path algorithm in Kwrowski and Kuske’s paper [KK99]. The paper presents a program for Floyd’s algorithm similar to the graph program Floyd_Warshall and proves its correctness as well as a cubic bound for the number of rule applications. (The program consists of rules with parameters, similar to our rule schemata, but [KK99] does not give a general formalism for such rules.) We already explained the correspondence between the GP program Floyd_Warshall and the Transformation Unit shortestpath in Section 5.4.

Although the graph-transformation languages GReAT [KAS03] and GRACE [Kus00] (Transformation Units) aim for verification, formal reasoning about GReAT and GRACE has not yet been sufficiently demonstrated.

Graph algorithms are formally discussed in [Eve89, CLR00, Jun02] in general terms and in [Sed02] for programs in the C programming language.
Most of these case studies also discuss complexity. In [Sed02], complexity is often improved by applying specific hardware concepts to increase performance of programs, e.g. using hash tables etc. As GP abstracts from such low-level implementation details, our measure of the number of rule-schema applications is rather rougher.

Our comparison of theoretical complexity results with results of the GP system compares to the usual program testing methods in software engineering. The GP system gives us an indication on correctness, termination and complexity of programs by producing all possible results or counting the graph changes in all possible derivation sequences. However, as explained in Section 5.3.3, the GP system cannot be employed for highly non-deterministic programs and graphs with more than 5 nodes as the number of answers for these runs grows at least exponentially.
Chapter 6

Adding Procedures: The Language $\text{GP}^+$

In this chapter, we extend GP with procedures and parameterised rule-schema calls to form the language $\text{GP}^+$. Intuitively, procedures are parameterised macros with an additional rule-schema declaration part. In contrast to non-recursive macro calls in GP, procedure calls in $\text{GP}^+$ may be recursive.

Parameterised procedures are a common programming concept for further abstraction in imperative programming languages [Wat04]. With the introduction of recursive procedures, $\text{GP}^+$ provides more programming comfort, better structuring and better reuse of program parts than GP. Recursive programs are sometimes more intuitive than programs using iteration. An example of this is a depth-first search program to construct a spanning tree in an undirected graph (see Example 6 in Section 6.1).

Parameters provide a way to pass information to different parts of a program. In $\text{GP}^+$, parameters are expressions of type int or string. Intermediate results that otherwise need to be encoded in the graph structure, may be passed through parameters on a higher level of abstraction. $\text{GP}^+$ only allows in-parameters, that is, parameters are used as constants.

In $\text{GP}^+$, rule-schema identifiers representing rule-schema calls, may be equipped with parameters. In such a parameterised rule-schema call,
variable identifiers in the associated rule-schema declaration are pre-instantiated with the values of the parameters. Thus, a call in general reduces the number of variable identifiers in the original rule-schema declaration. This concept provides a way to reuse rule schemata for different application purposes.

In Section 6.1, we introduce the syntax of GP+. We define the mechanism of parameter passing in Section 6.2. The structural-operational style semantics of GP+ is defined in Section 6.3. A brief comparison to other approaches is given in Section 6.4.

6.1 Syntax

In this section, we give an abstract, context-free syntax for GP+ programs, together with context conditions. The syntax of GP+ is an extension of the syntax of GP with the difference that GP+ programs are built in a flat block structure [Wat04], macro declarations and calls are replaced by procedure declarations and parameterised procedure calls, and rule-schema calls are replaced with parameterised rule-schema calls.

Figure 6.1 shows the flat block structure of a GP+ program. Procedure declarations may contain a number of rule-schema declarations but procedure declarations may not be nested. Rule-schema identifiers local to procedures hide identical global identifiers.

Example 6 (SpanningTree) Before we formally introduce the syntax of GP+, we present an example by applying the program SpanningTree to an undirected graph (for definition see Example 3 in Section 4.1). Thus, we first introduce some notions related to undirected graphs.

Let $G$ be a directed graph and $G'$ the underlying undirected graph of $G$. We refer to the label of an undirected edge $e$ in $G'$ by $m_{G'}(e)$. Whenever there is a path $e_1, \ldots, e_n$ from $v$ to $u$ with $v, u \in V_G$ in $G$, there is a path from $v$ to $u$ in $G'$ and that path consists of the undirected edges induced by $e_1, \ldots, e_n$. If $G'$ does not contain a cycle, then $G'$ is acyclic. If there
is a path \( p \) from \( v \) to \( u \) in \( G' \) with \( u = v \), then \( p \) is a cycle. If there is a path from \( u \) to \( v \) in \( G' \), then \( v \) is reachable from \( u \). An undirected graph \( G' \) is connected if any node in \( G' \) is reachable from any other node in \( G' \). An undirected tree is a connected, acyclic, undirected graph. A spanning tree of an undirected graph \( G' \) is an undirected tree \( T \) such that \( T \subseteq G' \) and \( V_T = V_{G'} \).

In pictures of undirected graphs, pairs of directed edges representing undirected edges are drawn as lines between source and target nodes with a unique label attached to the line. GP or GP\(^+\) programs working on undirected graphs must preserve undirectedness, that is, all (conditional) rule schemata may only use undirected graphs in their left and right side. Changing the label of an undirected edge implies changing the label of both directed edges representing the undirected edge. Deleting and inserting undirected edges implies deleting and inserting pairs of directed edges representing the undirected edges. Edge predicates in conditions of rule schemata
transforming undirected graphs must always be formulated for both directions of the underlying undirected edges, i.e. edge($v_2, v_1$) and edge($v_1, v_2$) must both be present in a condition.

Figure 6.2 shows a GP$^+$ program that non-deterministically computes a spanning tree in an undirected, connected, untagged, integer-labelled graph by a depth-first search strategy. Here, DFS(k:int) is a procedure declaration with formal parameter k of type int, containing two local rule-schema declarations ChildExists and Visit and a command sequence starting with while. Select is a global rule-schema declaration. DFS(1) and DFS(k+1) are parameterised procedure calls and ChildExists(k, _, _, _) and Visit(k, _, _, _) are parameterised rule-schema calls.

main = Select; DFS(1).

Select(x:int) =  

\[
\begin{array}{c}
\circ \ x \\
\hspace{1.5cm} 1 \\
\end{array} \quad \Rightarrow \quad \begin{array}{c}
\circ \ x_1 \\
\hspace{1.5cm} 1 \\
\end{array}
\]

DFS(k:int) =  

\[
\begin{pmatrix}
\text{ChildExists}(p,x,y,z:\text{int}) = \\
\begin{array}{c}
\circ \ x_p \\
\hspace{1.5cm} 1 \\
\end{array} \quad y \quad \begin{array}{c}
\circ \ z \\
\hspace{1.5cm} 2 \\
\end{array} \quad \Rightarrow \quad \begin{array}{c}
\circ \ x_p \\
\hspace{1.5cm} 1 \\
\end{array} \quad y \quad \begin{array}{c}
\circ \ z \\
\hspace{1.5cm} 2 \\
\end{array}
\end{array}
\]

Visit(p,x,y,z:int) =  

\[
\begin{array}{c}
\circ \ x_p \\
\hspace{1.5cm} 1 \\
\end{array} \quad y \quad \begin{array}{c}
\circ \ z \\
\hspace{1.5cm} 2 \\
\end{array} \quad \Rightarrow \quad \begin{array}{c}
\circ \ x_p \\
\hspace{1.5cm} 1 \\
\end{array} \quad y_{p+1} \quad \begin{array}{c}
\circ \ z_{p+1} \\
\hspace{1.5cm} 2 \\
\end{array}
\]

while ChildExists(k, _, _, _) do (Visit(k, _, _, _); DFS(k+1)).

Figure 6.2: Example program SpanningTree
The program is applied as follows. First, Select non-deterministically selects a node of the input graph and tags that node with 1. Then DFS is applied to the node tagged last, starting with the node tagged 1 (1 is passed as a parameter). If an untagged node connected with that node exists, the untagged node is tagged with the next higher integer and DFS is called recursively with that number. The application of Visit proceeds in a depth-first fashion until no untagged connected node exists which is checked by the while condition. Then, the while loop in iteration \( i \) of DFS terminates with a graph and the while loop of iteration \( i - 1 \) of DFS is continued by selecting (if possible) a sibling of a node tagged \( i - 1 \), etc. The program terminates if there are no untagged nodes left in the graph. It is not difficult to see that the tagged nodes and edges form a spanning tree of the input graph.

Note that an equivalent command sequence to the one in DFS is \((\text{Visit}(k,\_\_\_\_); \text{DFS}(k + 1))\)!. However, in this example we prefer an explicit termination condition in the form of \(\text{ChildExists}(k,\_\_\_\_\_\_)\).

Figure 6.3 shows an example run of \textbf{SpanningTree} starting with the following undirected graph.

![Undirected Graph](image)

Note that Select and Visit are applied non-deterministically. Therefore, the presented example run is only one out of 13 possible runs.

This example demonstrates the practicality of parameterised rule-schema calls. Only one rule schema needs to be declared to be applied in every iteration of the depth-first search. This solution is more intuitive than simulating depth-first search without parameterised identifiers to derive an output graph with the same labelling as \textbf{SpanningTree}. \(\square\)
Figure 6.3: An example run of SpanningTree
Note that we also allow rule-schema calls without parameters (see \texttt{Select} in Figure 6.2). These have the same semantics as in GP. However, in GP\textsuperscript{+}, we allow parameterised rule-schema calls. Parameters represent constant values (see Example 6) that pre-instantiate variable identifiers in the associated rule-schema declarations. Procedures only pass parameters on to rule-schema calls. The symbol \texttt{'}\_\texttt{'} represents values determined by graph matching in rule-schema applications, as it happens in GP. Thus, the calls \texttt{Select} and \texttt{Select}\(\texttt{'}\_\texttt{'}\) are semantically equivalent.

We next define the syntax of GP\textsuperscript{+}. We assume that the nonterminals that are not defined but used in the grammar are those defined in the syntax of GP in Section 4.1, including the associated context conditions. However, some definitions of nonterminals are repeated for better readability.

Macro declarations of GP are replaced by procedure declarations in GP\textsuperscript{+}. Thus, we first introduce procedure declarations.

\section*{Procedure Declaration}

\begin{verbatim}
ProcDecl ::= ProcId ['(' ParSection {',' ParSection '}')]' '='
            ProcBody

ProcId ::= Identifier

ProcBody ::= ['(' RuleSchemaDecl {RuleSchemaDecl '}')]' ComSeq '.

ParSection ::= VarId {',' VarId '} Type

Type ::= int | string
\end{verbatim}

Note that \texttt{ParSection} and \texttt{Type} are already defined in GP in Section 4.1.

Context Conditions:

1. All variable identifiers (\texttt{VarId}) in the parameter sections listed with the procedure identifier (\texttt{ProcId}) must be distinct.

2. All rule-schema identifiers (\texttt{RuleId}) occurring in the list of rule-schema declarations in the same procedure body (\texttt{ProcBody}) must be distinct.
3. All identifiers must be distinct from the reserved words of GP+. These are the reserved words of GP and, in addition, the word `_`.

A procedure declaration consists of a header comprising a procedure identifier and a (possibly empty) list of parameters, and a body containing a (possibly empty) list of rule-schema declarations followed by a command sequence. A rule-schema declaration consists of a header comprising a rule-schema identifier and a (possibly empty) list of parameters, and a body containing the declaration of the rule-schema graphs and a (possibly empty) condition declaration. Rule-schema declarations in a procedure body are called local.

A GP macro declaration is a special form of procedure declaration, namely a procedure without parameters that does not include any rule-schema declarations in the procedure body and that is not called recursively.

**Assumption 3 (List of Formal Parameters)** For this chapter, we assume without loss of generality that every parameter section (ParSection) occurring in the header of a rule-schema declaration or procedure declaration has the form VarId `: ?` Type. Hence, every declaration of a rule schema or procedure comes with a list of formal parameters of the form \((x_1: t_1; \ldots; x_n: t_n)\), where each pair \(x_i: t_i\) consists of a variable identifier \(x_i\) and a type \(t_i\).

**GP Programs**

\[
\begin{align*}
\text{Prog} & ::= \text{Decl} \{\text{Decl}\} \\
\text{Decl} & ::= \text{RuleSchemaDecl} | \text{ProcDecl} | \text{MainDecl}
\end{align*}
\]

MainDecl and RuleSchemaDecl are defined in the GP syntax of Section 4.1. The context conditions for these declarations also apply to GP+.
Context Conditions:

1. All procedure and rule-schema identifiers (ProcId and RuleId) declared in the list of declarations of the same program are pairwise distinct.

A GP+ program consists of one main command sequence and a number of rule schema and procedure declarations. We call this list of declarations the program's declaration list. Rule-schema declarations in the program's declaration list are called \textit{global}. We call the command sequence in MainDecl the \textit{main command sequence}.

\textbf{GP+ Command Sequences}

\begin{align*}
\text{ComSeq} & ::= \text{Com} \{';' \text{Com}\} \\
\text{Com} & ::= \text{RuleSetCall} \mid \text{ProcCall} \\
& \quad \mid \text{if ComSeq then ComSeq \{else ComSeq\}} \\
& \quad \mid \text{try ComSeq then ComSeq \{else ComSeq\}} \\
& \quad \mid \text{while ComSeq do ComSeq} \\
& \quad \mid \text{ComSeq '!'} \\
& \quad \mid \text{skip} \mid \text{fail} \\
\text{RuleSetCall} & ::= \text{ParRuleId} \mid \{'' [\text{ParRuleId} \{',', \text{ParRuleId}\}] ''\}' \\
\text{ParRuleId} & ::= \text{RuleId} ['' \text{ActPar} \{',', \text{ActPar}\} '')''\] \\
\text{ActPar} & ::= \text{Exp} \mid ''\_'' \\
\text{ProcCall} & ::= \text{ProcId} ['' \text{Exp} \{',', \text{Exp}\} '')''\]
\end{align*}

Note that Exp is already defined in Section 3.1.

Context Conditions:

1. For every rule-schema identifier (RuleId) in RuleSetCall there must be a rule-schema declaration (RuleSchemaDecl) with the same identifier either in the same procedure body or in the program's declaration list.
2. For every procedure identifier (ProcId) occurring in a procedure call (ProcCall) there must be a procedure declaration with the same identifier in the same program.

3. Expressions (Exp) in a procedure call (ProcCall) or in a rule-set call (RuleSetCall) in the main command sequence (MainDecl) must be either a numeral in [-]Num or a string in String.

4. All variable identifiers occurring in expressions (Exp) in a procedure call (ProcCall) or in a rule-set call (RuleSetCall) must have been declared in the header of the surrounding procedure.

5. If \( \text{id}(a_1, \ldots, a_n) \) is a procedure call (ProcCall) or a parameterised rule identifier (ParRuleId) in a rule-set call (RuleSetCall) and \( \text{id}(x_1; t_1; \ldots; x_m; t_m) \) with \( x_1, \ldots, x_m \in \text{VarId} \) and \( t_1, \ldots, t_n \in \text{Type} \) is the header of the declaration of \( \text{id} \), then \( n = m \) and \( \text{type}(a_i) = t_i \) for all \( a_i \) such that \( a_i \neq \_ \).

6. The type of a variable identifier is the type assigned to that identifier in the list of formal parameters of the rule-schema or procedure declaration.

For a procedure call or rule-schema call, we call the list \( (a_1, \ldots, a_n) \) attached to the procedure or rule-schema identifier (ProcCall or ParRuleId) the list of actual parameters, and \( a_1, \ldots, a_n \) the actual parameters. We call an actual parameter a constant if it is a string in String, a numeral in [-]Num or the symbol \( \_ \). We call a rule-schema identifier parameterised if the list of actual parameters is not empty. Thus, a rule-schema identifier may occur with or without parameters in a rule-set call. Every parameterless rule-schema identifier \( \text{id} \) in a rule-set call is syntactically equivalent to the parameterised rule-schema identifier \( \text{id}(\_, \ldots, \_) \). We want to keep parameterless rule-schema identifiers in calls for better readability. We call a rule-set call that contains parameterised rule-schema identifiers a parameterised rule-set call.

A procedure call is parameterised if the list of actual parameters is not empty.
In GP⁺ programs, often only the first few parameters in procedure or rule-schema calls are distinct from the symbol _-. For better readability, we also make the following assumption.

**Assumption 4 (Trailing ‘_’ Parameters)** For this chapter and Chapter 7, we assume without loss of generality, that every parameterised rule-schema identifier and every parameterised procedure call of the form \( id(a_1, \ldots, a_n, _, \ldots, _) \), where \( n \geq 1 \), may be replaced with the call of the form \( id(a_1, \ldots, a_n) \) in which all trailing _- symbols are omitted. We assume that the denotationally omitted ‘_’ parameters are implicitly given.

For example, the call \( \text{ChildExists}(k, _, _, _) \) in the program text in Figure 6.2 may be replaced with the equivalent call \( \text{ChildExists}(k) \).

We replace the context condition of Section 3.1 “Each variable identifier occurring in labels of category RightLabel must also occur in some label of category LeftLabel of the same rule-schema declaration.” for the use of variable identifiers in rule-schema declarations with the following condition:

Let \( D \) be a rule-schema declaration with header \( \text{id}(x_1 : t_1; \ldots; x_n : t_n) \) where \( x_1, \ldots, x_n \in \text{VarId} \) and \( t_1, \ldots, t_n \in \text{Type} \). Then each variable identifier \( x_i \) with \( 1 \leq i \leq n \) occurring in labels of category RightLabel must either also occur in some label of category LeftLabel of the same rule-schema declaration or (1) all calls of the rule schema \( \text{id} \) in the same program are parameterised and (2) for every such parameterised rule-schema call \( \text{id}(a_1, \ldots, a_n) \) and \( i = 1, \ldots, n, a_i \neq _- \) if \( x_i \) does not occur in some label of category LeftLabel of the same rule-schema declaration.

Analogously, this change of context condition applies to variable identifiers in rule-schema conditions of category Cond. The reason for these changes of context conditions is that they provide us with a way to label nodes and edges in the right graph or the condition with numerals or strings that are obtained from the parameters of a call (see the remarks in Section 6.2 after Definition 17).

Except for RuleSetCall and MacroCall, respectively ProcCall, all commands in GP⁺ are equivalent to the ones in GP. Macro calls are simulated by
procedure calls without parameters. RuleSetCalls of GP are still valid in GP+. As the symbol ‘_’ represents values determined by graph matching, it may only appear in rule-schema calls but not in procedure calls.

Among other things, the above context conditions require the uniqueness of procedure identifiers and rule-schema identifiers within the same program or procedure. Procedure identifiers are unambiguous. Rule-schema identifiers may be ambiguous because they may refer to both local rule-schema declarations in different procedure declarations and global rule-schema declarations.

The following example shows the possible ambiguity of rule-schema identifiers.

**Example 7 (Ambigious Rule-Schema Identifiers)** The following program fragment is a syntactically correct GP+ program with ambiguous rule-schema identifiers, where “…” stands for (a part of) a rule-schema declaration.

```
main  =  Rule; Proc1; Proc2; Proc3.
Rule  =  ...
Proc1 =  (Rule = ...) Rule.
Proc2 =  Rule.
Proc3 =  (Rule = ...) Rule.
```

GP+ is statically scoped (see [Seb02, Wat04]), i.e. each procedure call and rule-schema call is executed in the environment of the surrounding declaration. Therefore, we associate local rule-schema declarations to calls whenever these are available and global ones otherwise. In this example, this means that the rule schema associated with Rule in the main command sequence is the global rule schema declared in the program’s declaration list, the rule schema associated with Rule in Proc1 and Proc3 is the local rule schema declared in Proc1 and Proc3 respectively, and the rule schema associated with Rule in Proc2 is the global rule schema declared in the program’s declaration list. □
As rule-schema identifiers are not necessarily unique, it is a priori not clear which declaration (binding occurrence) of the identifier is associated with a call (applied occurrence). However, the usual convention of block-structured languages applies (see [Wat04]), where local declarations hide global declarations.

The standard approach to map applied occurrences to the preferred binding occurrences is to introduce so-called environments to the semantics of the language (see [Wat91, Win93, Plo04, NN07]). Environments store identifier-to-body relations. Identifiers in an environment are updated with the new body relation whenever a new block is entered that contains a declaration matching the identifier. The updating is typically done in the SOS inference rules (see [Plo04]). For more complicated programming languages such a concept is inevitable because declarations may be nested, identifiers may not be unique and the application of declarations may be dependent on the order in which these are given in the program.

However, the aim of developing GP$^+$ is to keep the language as simple as possible to facilitate verification and on the other hand make it practical by offering recursion and parameterised calls. To achieve this, GP$^+$ uses a very simple identifier concept, in which procedure identifiers are unique. Thus, for each applied occurrence of a procedure identifier, there is a unique binding occurrence. Hence, the nesting and identifier concept is so simple, that we can abstain from introducing environments. Instead, programs are syntactically converted into a form in which rule-schema identifiers of binding occurrences are unique. We implement scoping where local declarations hide global ones by syntactic program transformation. This is done by attaching to local rule-schema identifiers the identifiers of the enclosing procedures.
**Assumption 5 (Disambiguating Rule-Schema Identifiers)** For this chapter, we assume without loss of generality, that every GP+ program $P$ is transformed into a form where all rule-schema identifiers are unique. This is achieved by replacing every rule-schema identifier $r$ that is declared in the declaration of a procedure $p$, with the combined identifier $pr$ in $p$. Here, we assume that the new identifier $pr$ does not already occur in the program’s declaration list. As a consequence, every rule-schema identifier comes with a unique rule-schema declaration.

**Example 8 (Disambiguating Rule-Schema Identifiers)** Consider the GP+ program in Example 7. After transforming the program according to Assumption 5, it looks as follows:

```
main = Rule; Proc1; Proc2; Proc3.
Rule = ...
Proc1 = (Proc1Rule = ...) Proc1Rule.
Proc2 = Rule.
Proc3 = (Proc3Rule = ...) Proc3Rule.
```

This example demonstrates how rule-schema identifiers are made unique and that every rule-schema call in the resulting program comes with a unique local or global rule-schema declaration. □

### 6.2 Parameter Passing

In this section, we define the mechanism of parameter passing in GP+. In GP, the semantics of a macro call is the semantics of the command sequence defined by the macro declaration. In GP+, the semantics of a procedure call is also the semantics of the command sequence of the associated declaration. The difference is that we replace the formal parameters in the command sequence with the actual parameters of the procedure call before returning the command sequence. The replacement is not purely syntactic, however, because actual parameters need to be evaluated to constants before they replace their formal counterparts.
The semantics of a parameterised rule-schema call of the form \( r(a_1, \ldots, a_n) \) is the semantics of the declaration of \( r \) in which all formal parameters are replaced with the actual parameters \( a_1, \ldots, a_n \) if these are different from the keyword \( _{\_} \).

Thus, we define the meaning of parameterised procedure and rule-schema calls by a transformation of their declarations. Parameters are passed by procedure calls and their effect on the input graph is given by the application of parameterised rule-schema calls. To obtain proper rule-schema declarations with only simple expressions (SimpleExp) in the labels of left graphs, we have to make sure that all actual parameters in parameterised rule-schema calls are constants.

We next define a function that partially evaluates procedure calls and by returning command sequences.

**Definition 15 (eval\( P \))** Let \( P \) be a GP\(^+\) program and ProcCall\( (P) \) be the set of all procedure calls occurring in \( P \). We define the function eval\( P \):ProcCall\( (P) \rightarrow \text{ComSeq} \) as follows. Let \( p(a_1, \ldots, a_n) \in \text{ProcCall}(P) \), with \( p \in \text{ProcId} \) and \( a_1, \ldots, a_n \in \text{Exp} \), and let \( p(x_1:t_1; \ldots; x_n:t_n) \) be the header and \( S \) the command sequence of the unique declaration of \( p \) in \( P \). Then \( \text{eval}_P(p(a_1, \ldots, a_n)) \) is obtained from \( S \) by replacing every procedure call and every rule-schema call \( \text{id}(e_1, \ldots, e_k) \) with \( \text{id}(\text{sub}(e_1), \ldots, \text{sub}(e_k)) \).

Here, sub\( (e_i) \) is defined, for every parameter \( e_i \), as follows:

- For \( j = 1, \ldots, n \), \( \text{sub}(x_j) = a_j \).

- \( \text{sub}(_{\_}) = _{\_} \).

- For \( n \in \text{Num} \), \( \text{sub}(n) = n \) and \( \text{sub}(\neg n) = \neg n \).

- For \( c_1, \ldots, c_n \in \text{Char} \), \( n \geq 0 \), \( \text{sub}(_{c_1 \ldots c_n}) = _{c_1 \ldots c_n} \).

- For \( e, e' \in \text{Exp} \) and \( \odot \in \text{ArithOp} \), \( \text{sub}(e \odot e') = \text{sub}(e) \odot \text{sub}(e') \).

- For \( e \in \text{Exp} \), \( \text{sub}((e)) = (\text{sub}(e)) \).
The result of an application of eval_P is a command sequence of GP^+. Note that sub replaces every subexpression with itself except for formal parameters which are replaced with the respective actual parameter expressions. If all actual parameters of the call are constants, all expressions occurring in the resulting command sequence are variable-free.

**Claim 7 (Evaluation to Variable-free Expressions)** Let P be a GP^+ program and p(a_1,\ldots,a_n) be a procedure call in P where a_1,\ldots,a_n are constants. Then no expression occurring in eval_P(p(a_1,\ldots,a_n)) contains any variable identifiers.

**Proof.** By context conditions, every variable identifier in the command sequence of the declaration associated with p(a_1,\ldots,a_n) has to be declared in the header p(x_1:t_1;\ldots;x_n:t_n) of that declaration. Thus, all variable identifiers in p’s declaration are contained in \{x_1,\ldots,x_n\}. By eval_P(p(a_1,\ldots,a_n)), every x_i, i = 1,\ldots,n, is replaced with a_i. Since a_i is a constant, all expressions in eval_P(p(a_1,\ldots,a_n)) are free of variable identifiers.

We use this result to normalise to constants all expressions in the command sequence resulting from an application of eval_P to a procedure call in which all actual parameters are constants.

**Definition 16 (Procedure Call Evaluation)** Let P be a GP^+ program such that the actual parameters of all procedure calls occurring in P are constants. We define the function eval_P^+:ProcCall(P) \to ComSeq as follows. Let p(a_1,\ldots,a_n) \in ProcCall(P) and let \alpha be any assignment (see Definition 9). Then eval_P^+(p(a_1,\ldots,a_n)) is the command sequence obtained from eval_P(p(a_1,\ldots,a_n)) by replacing every occurring expression e \in Exp of type int with the unique numeral in [-]Num representing the value e^\alpha.

Note that expressions of type string and the symbol _ are not replaced by eval_P^+ because only expressions of type int are considered. Strings need not be considered because there are no operations on strings defined in GP^+. In the context of Definition 16, an integer expression e in eval_P(p(a_1,\ldots,a_n))
does not contain variable identifiers by Claim 7. Thus, the value $e^n$ is independent on $\alpha$ and therefore, the numeral in $\lfloor\cdot\rfloor_{\mathrm{Num}}$ is unique. Therefore, the command sequence resulting from an application of $\text{eval}^n_P$ is unique and all occurring expressions are constants.

Example 9 (Procedure Call Evaluation) Consider the GP$^+$ program $P$ in Figure 6.2 and assume that $P$ has been transformed according to Assumption 5. Then,

\[
\text{while DFSChildExists}(1,\_\_\_\_) \text{ do (DFSVisit}(1,\_\_\_\_); \text{DFS}(2))
\]

is the command sequence resulting from applying $\text{eval}^n_P$ to the procedure call $\text{DFS}(1)$. First, all formal parameters in the original command sequence are replaced with 1 by $\text{eval}_P(\text{DFS}(1))$ and the expression $1 + 1$ is replaced with 2 according to Definition 16. The command sequence resulting from applying $\text{eval}^n_P$ to the procedure call $\text{DFS}(2)$ is the following sequence:

\[
\text{while DFSChildExists}(2,\_\_\_\_\_\_) \text{ do (DFSVisit}(2,\_\_\_\_\_\_); \text{DFS}(3))
\]

Note, that by Assumption 4, the first command sequence is equivalent to

\[
\text{while DFSChildExists}(1) \text{ do (DFSVisit}(1); \text{DFS}(2))
\]

and the second sequence is equivalent to

\[
\text{while DFSChildExists}(2) \text{ do (DFSVisit}(2); \text{DFS}(3))
\]

\[\square\]

Note that all actual parameters in the resulting command sequences in Example 9 are constants. As observed above, this will always be the case for a command sequence resulting from applying $\text{eval}^n_P$ to a call in which all actual parameters are constants. We will use this fact in the inference rules of the structural operational semantics of GP$^+$ to define the semantics of procedure calls and parameterised rule-schema calls (see Section 6.3). We will prove in Proposition 16 that no transition sequence in the GP$^+$ semantics starting with the main command sequence (which is variable-free by a context condition) contains any variable-identifiers.

We next define induced rule-schema declarations.
Definition 17 (Induced Rule-Schema Declaration) Let $P$ be a $\text{GP}^+$ program and $r(a_1, \ldots, a_n)$ be a rule-schema call in $P$ where $a_1, \ldots, a_n$ are constants. Let $r(x_1: t_1; \ldots; x_n: t_n)$ be the header of the unique declaration $D$ of $r$ in $P$. Then the rule-schema declaration induced by $r(a_1, \ldots, a_n)$ is obtained from $D$ by replacing, for $i = 1, \ldots, n$, every occurrence of the variable identifier $x_i$ in $D$ with $a_i$ if $a_i \neq \_$. □

Note that the resulting rule-schema declaration is a declaration in the sense of the $\text{GP}$ grammar in Section 4.1. Actual parameters in form of strings and numerals replace variable identifiers in the rule-schema declaration. Thus, simple expressions (SimpleExp) in labels in the left graph of the original rule-schema declaration remain simple expressions in the induced declaration. Induced rule-schema declarations typically contain less variable identifiers than the original declaration.

It is easy to check that all conditions of Section 4.1 concerning the use of variable identifiers in the same rule-schema declaration are satisfied for induced rule-schema declarations. By the new context condition in Section 6.1, all variable identifiers that occur in labels in the right graph of a rule-schema declaration but not in some label in the left graph of the same declaration must be associated with some actual parameter different from the symbol $. Hence, these variable identifiers are replaced with a numeral or string by Definition 17 and the context conditions of Section 4.1 are again satisfied. This also applies to conditional ruleschemata. For example, consider the rule-schema declaration $r(x: \text{int}) = \{ \} => \{ 1 : x \} \text{ where } x < 3$ that inserts a node with label $x$, which is not allowed in Section 4.1 but in Section 6.1. Every call of $r$ induces a rule-schema declaration in which $x$ is replaced with a numeral. Thus, the rule-schema declaration induced by a call of $r$ is a rule-schema declaration according to Section 4.1.
Example 10 (Induced Rule-Schema Declaration) Consider the GP\(^+\) program in Figure 6.2 and assume that the program has been transformed into the form according to Assumption 5. The rule-schema declaration induced by a call DFSVisit(1) is the following:

\[
    \text{DFSVisit}(p, x, y, z; \text{int}) =
    \begin{array}{c}
        x_{-1} \\
        1
    \end{array}
    \xrightarrow{y}
    \begin{array}{c}
        z \\
        2
    \end{array}
    \implies
    \begin{array}{c}
        x_{-1} \\
        1
    \end{array}
    \xrightarrow{y_{-1}+1}
    \begin{array}{c}
        z_{1}+1 \\
        2
    \end{array}
\]

\[\square\]

6.3 SOS Inference Rules

In this section, we define the structural operational semantics of GP\(^+\). We use the same types of transition relation and inference rules as defined for GP in Section 4.2.2. We now assume that command sequences in ComSeq are the ones defined for GP\(^+\) (Section 6.1). We also assume that every transition sequence is started from a configuration \(\langle P, G \rangle\) where \(P\) is the main command sequence (MainDecl) of the program \(P\) under consideration. The semantic function \(S_{\text{sos}}\) for GP is the same for GP\(^+\) except that the transition sequences are derived with the inference rules for GP\(^+\). The only difference between the inference rules for GP and GP\(^+\) is the additional evaluation of procedure calls and parameterised rule-set calls in GP\(^+\).

By Definition 17, parameterised rule-schema calls in which all actual parameters are constants induce new rule-schema declarations. As argued above, each of these declarations is a rule-schema declaration according to Section 4.1. Thus, according to Definition 11, each of these rule-schema declarations induces a set of conditional rules.

Inference rules in GP\(^+\) use meta-variables \(C, Q, Q', S\) for command sequences in category ComSeq, \(G\) and \(H\) for graphs in \(\mathcal{G}(\mathcal{L})\), \(M\) for a call in category ProcCall and \(\mathcal{R}\) for a call in category RuleSetCall. All command sequences and calls are defined in relation to the program \(P\) under consideration. Given a rule-set call in category RuleSetCall of the form
\( \mathcal{R} = \{ r_1(a_1, \ldots, a_p), \ldots, r_n(a_1, \ldots, a_q) \} \) where all \( a_i \) are constants, we write \( G \Rightarrow_\mathcal{R} H \) if \( G \Rightarrow_\mathcal{U} H \). Here, \( \mathcal{U} \) is the union of the rule sets induced (according to Definition 11) by the declarations induced (according to Definition 17) by \( r_1(a_1, \ldots, a_p) \) to \( r_n(a_1, \ldots, a_q) \).

We next define the SOS inference rules for GP\(^+\).

**Definition 18 (SOS inference rules for GP\(^+\))** Figure 6.4 shows the inference rules for the structural operational semantics of GP\(^+\), where \( \rightarrow^+ \) is the transitive closure of \( \rightarrow \) and where a command sequence \( C \) **finitely fails** on a graph \( G \in \mathcal{G}_s(\mathcal{L}) \) if (1) there does not exist an infinite sequence \( \langle C, G \rangle \rightarrow \langle C_1, G_1 \rangle \rightarrow \ldots \) and (2) for each terminal configuration \( \gamma \) such that \( \langle C, G \rangle \rightarrow^+ \gamma, \gamma = \text{fail} \).

The inference rules of GP\(^+\) differ from the inference rules of GP in the rules for calls, that is, in rules [Call\(_1\)] to [Call\(_3\)]. We assume (see above) that every sequence of transitions is started with the configuration \( \langle P, G \rangle \) where \( P \) is the main command sequence of the GP\(^+\) program. The parameter passing in procedure calls is implemented by [Call\(_3\)], using the function \textup{eval}_p. (In GP the replacement of macro calls with declarations is done syntactically before a transition sequence is initialised, see Assumption 2.)

The meaning of a rule-set call \( \mathcal{R} \) of parameterised rule-schema identifiers is the application of the set of conditional rules induced by the induced rule-schema declarations of \( \mathcal{R} \). If no rule in this set is applicable, the application of \( \mathcal{R} \) fails. In contrast, a procedure call \( M \) cannot fail and always induces a normalised command sequence \textup{eval}_p(M) \) which is then further evaluated by the other rules (where it may fail, too). For the semantics, it makes no difference whether procedure calls are recursive or not as we leave the evaluation to the repeated substitution of command sequences for calls. This process may result in an infinite transformation sequence.

Note that the inference rules for the derived constructs in GP\(^+\) are the same as in GP, see Figure 4.6.
[Call1] \[
\begin{array}{c}
G \Rightarrow R \ H \\
\langle R, G \rangle \rightarrow H
\end{array}
\]
where all actual parameters occurring in \( R \) are constants

[Call2] \[
\begin{array}{c}
G \notin \text{Dom}(\Rightarrow R) \\
\langle R, G \rangle \rightarrow \text{fail}
\end{array}
\]
where all actual parameters occurring in \( R \) are constants

[Call3] \[
\begin{array}{c}
\langle M, G \rangle \rightarrow \langle \text{eval}^p(M), G \rangle \\
\end{array}
\]
where all actual parameters occurring in \( M \) are constants

[Seq1] \[
\begin{array}{c}
\langle Q, G \rangle \rightarrow \langle Q', H \rangle \\
\langle Q; S, G \rangle \rightarrow \langle Q'; S, H \rangle
\end{array}
\]

[Seq2] \[
\begin{array}{c}
\langle Q, G \rangle \rightarrow H \\
\langle Q; S, G \rangle \rightarrow \langle S, H \rangle
\end{array}
\]

[Seq3] \[
\begin{array}{c}
\langle Q, G \rangle \rightarrow \text{fail} \\
\langle Q; S, G \rangle \rightarrow \text{fail}
\end{array}
\]

[If1] \[
\begin{array}{c}
\langle C, G \rangle \rightarrow^+ H \\
\langle \text{if } C \text{ then } Q \text{ else } S, G \rangle \rightarrow \langle Q, G \rangle
\end{array}
\]

[If2] \[
\begin{array}{c}
C \text{ finitely fails on } G \\
\langle \text{if } C \text{ then } Q \text{ else } S, G \rangle \rightarrow \langle S, G \rangle
\end{array}
\]

[While1] \[
\begin{array}{c}
\langle C, G \rangle \rightarrow^+ H \\
\langle \text{while } C \text{ do } Q, G \rangle \rightarrow \langle Q ; \text{while } C \text{ do } Q, G \rangle
\end{array}
\]

[While2] \[
\begin{array}{c}
C \text{ finitely fails on } G \\
\langle \text{while } C \text{ do } Q, G \rangle \rightarrow G
\end{array}
\]

Figure 6.4: The SOS inference rules for GP⁺
Rules $[\text{Call}_1]$ to $[\text{Call}_3]$ are only applicable if the application conditions are satisfied. These require that all actual parameters in calls are constants so that Definitions 16 and 17 are applicable. If any of the application conditions of $[\text{Call}_1]$ to $[\text{Call}_3]$ does not apply, the configuration is stuck. However, transformation sequences in $\text{GP}^+$ should only get stuck in cases where transition sequences in $\text{GP}$ get stuck. Thus, the following proposition shows that all actual parameters in transition sequences are constants and that therefore transition sequences cannot get stuck because rules $[\text{Call}_1]$ to $[\text{Call}_3]$ are not applicable.

**Proposition 16 (Actual Parameters in Transition Sequences)**

Let $P$ be a $\text{GP}^+$ program. Then, all actual parameters occurring in any transition sequence starting with a configuration of the form $\langle Q, G \rangle$, where $Q$ is the main command sequence of $P$ and $G$ is a graph in $\mathcal{G}^\tau(\mathcal{L})$, are constants.

**Proof.** We prove the proposition by induction on the length of the transition sequence. Let $\langle Q, G \rangle$ be the initial configuration of a given transition sequence. Then $Q$ is the main command sequence of the program $P$. By the context conditions in Section 6.1, all actual parameters in $Q$ are constants. Thus, the proposition holds for the initial configuration. Let now $\langle Q, G \rangle \rightarrow^k \langle Q', G \rangle$ be any sequence of transitions for which the proposition holds and let $\langle Q', G \rangle \rightarrow \gamma$ be the next transition in that sequence. For $\gamma = H$ with $H \in \mathcal{G}^\tau(\mathcal{L})$ or $\gamma = \text{fail}$, the proposition is trivially satisfied. Let therefore $\gamma$ be of the form $\langle P', H \rangle$, with $P' \in \text{ConSeq}$ and $H \in \mathcal{G}^\tau(\mathcal{L})$. We need to show that all actual parameters in $P'$ are constants. The rules that can derive $\langle P', H \rangle$ are $[\text{Call}_3]$, $[\text{Seq}_1]$, $[\text{Seq}_2]$, $[\text{If}_1]$, $[\text{If}_2]$ and $[\text{While}_1]$. However, $[\text{Call}_3]$ is the only rule that alters actual parameters. Thus, for an application of any other rule, the proposition holds. By hypothesis, actual parameters in $Q'$ are constants and thus, by Claim 7 and the definition of $\text{eval}^\tau_{\pi}$, applying $[\text{Call}_3]$ yields a command sequence $P'$ in which all actual parameters are constants (confer the remark below Definition 16). Thus, the proposition holds for any transition sequence starting with $\langle Q, G \rangle$. □
Example 11 (SOS Transition Sequence) Consider the example run of SpanningTree in Figure 6.3. Let $G$ be the input graph, $G'$ be the first derived graph and $H$ the resulting graph in Figure 6.3. Then the transition sequence corresponding to that run looks as follows:

\[
\langle \text{Select; DFS(1), } G' \rangle \\
\rightarrow [\text{Call}_1] \quad \langle \text{DFS(1), } G' \rangle \\
\rightarrow [\text{Call}_2] \quad \langle \text{DFS(1); DFSVisit(1); DFS(2)), } G' \rangle \\
\rightarrow [\text{While}_1] \quad \langle \text{DFSVisit(1); DFS(2)), while DFSChildExists(1) do (DFSVisit(1); DFS(2)), } G' \rangle \\
\rightarrow [\text{Call}_1] \quad \langle \text{DFS(2); while DFSChildExists(1) do (DFSVisit(1); DFS(2)), } G'' \rangle \\
\rightarrow [\text{Call}_3] \quad \langle \text{while DFSChildExists(2) do (DFSVisit; DFS(3))), (while DFSChildExists(1) do (DFSVisit(1); DFS(2)), } G'' \rangle \\
\ldots \\
\rightarrow \langle \text{while DFSChildExists(1) do (DFSVisit(1); DFS(2)), } H \rangle \\
\rightarrow [\text{While}_2] \quad H
\]

6.4 Related Work

Procedures or methods are a common concept of abstraction in imperative and object-oriented programming languages. Procedures assist program structuring and the reuse of program parts [Wat04]. The introduction of parameters enhances procedural abstraction in that different computations are performed for a procedure depending on the values of parameters. The introduction of macros was needed for a good structuring of GP programs. Procedures in GP$^+$ enhance the properties of macros with a further level of abstraction by parameter passing. For the procedure concept in GP$^+$, parameterised calls of rule schemata and their evaluation are essential as otherwise, parameters would have no effect on the graph and would only be passed through the program.
Encapsulation of programs into blocks is not a new concept for graph-transformation languages. Examples of concepts similar to procedures in GP\(^+\) are the concept of transformation units [Kus00], transactions in the PROGRES language [Sch91, SWZ99] and methods in the Fujaba language [FNTZ98]. Rules in transformation units and in Fujaba methods are always declared locally. In transformation units this is the case because there is no ‘main’ unit and therefore no starting point for a graph-transformation. All units are evaluated at the same time. In PROGRES, declarations of rules are always global. Rule declarations in transactions are not allowed. All three procedure concepts may be parameterised in a way similar to GP\(^+\).

PROGRES and Fujaba provide not only in- but also out-parameters for both procedures and rules. Hence, the concept is similar to functions in imperative or object-oriented languages. Both languages also allow variable declarations and function declarations for expressions, which is also excluded in GP\(^+\). Introducing variables to GP\(^+\) would mean to extend the graph states of the transition relation to graphs plus variable states. This requires a more complicated semantics similar to those of traditional imperative programming languages which would make GP\(^+\) non-declarative. As our focus is on syntactic and semantic simplicity these concepts are not introduced for GP or GP\(^+\).

Another concept for abstraction is introduced in [Hof03] for the DiaPlan language. As DiaPlan is based on a different graph type than GP\(^+\) and the languages above, namely nested graphs with graph variables and structural graph types, the procedure concept is also different to the ones discussed above. In [Hof03], procedures name and encapsulate sets of rules, and their names may occur in the hyper-edges of graphs in these rules. Graphs connected with these edges are the parameters of the procedure. Such a concept is interesting for the type of graphs used in DiaPlan but does not relate well to GP’s credo of syntactic and semantic simplicity.

Module concepts for graph transformation systems are another means of further abstraction not supported by GP\(^+\). Modules are a structuring concept that supports information hiding by making only specific resources
public for use, which is relevant for large-scale programming. A comparison of different module concepts for graph transformation systems is given in [HEET97].

In general, parameterised procedures provide better structuring of programs on the one hand and more abstraction on the other hand. This is why procedures were introduced to GP+. GP+ keeps GP’s declarative character and is therefore not too close to imperative languages.
Chapter 7

Case Study: Planarity Testing

In this chapter, we present a large-scale case study on planarity testing. A graph is *planar* if it can be embedded in the plane without crossing edges. Planarity-test algorithms are originally important for circuit design since hardware elements are ideally placed on a board without crossing wires.

Several planarity-test algorithms have been proposed in the literature (see Section 7.4 for an overview). In this chapter, we implement the planarity-test algorithm due to Auslander, Parter and Goldstein [AP61, Gol63] in GP+.

Because of the complexity of the algorithm, we do not intend to formally prove correctness or termination of the GP+ planarity-test program and the procedures involved. We relate the GP+ procedures to the respective parts in the original algorithm and argue informally for the program’s correctness and termination. Where possible, we run the procedures on the GP system to test their correctness.

We review the definition of some notions related to undirected graphs in Section 7.1. The algorithm by Auslander, Parter and Goldstein is introduced in detail in Section 7.2. The GP+ program for the algorithm is given in Section 7.3. In Sections 7.3.1 to 7.3.5, we discuss the procedures of the
GP$^+$ planarity-test program. We conclude the chapter with a brief overview over planarity-test algorithms in Section 7.4.

### 7.1 Preliminaries

In the following, we repeat definitions related to undirected graphs. Some notions have already been introduced in Section 6.1 to discuss the example program $\text{SpanningTree}$. We do not repeat these notions here.

Let $G$ be an undirected graph that contains three distinct nodes $u, v, w \in V_G$ such that $u$ is reachable from $v$ but every path from $v$ to $u$ contains $w$. Then $w$ is a separation point of $G$.

**Definition 19 (Biconnected)** An undirected graph $G$ is *biconnected*, if $G$ is connected and contains no separation points.

Consider the undirected, connected graph $G$ in Figure 7.1. $G$ is not biconnected as the node labelled with $w$ is a separation point of $G$. Every path from the node labelled with $u$ to the node labelled with $v$ contains the node labelled with $w$. In contrast, the undirected, connected graphs in Figure 7.3 contains no separation points and is therefore biconnected.

![Figure 7.1: A graph that is not biconnected](image)

**Definition 20 (Bipartite)** An undirected graph $G$ is *bipartite* if the set $V_G$ of nodes in $G$ can be decomposed into two disjoint sets such that no two nodes within the same set are adjacent.

An example for a bipartite graph is shown in Figure 7.2, where the node labels 0 and 1 denote the two sets into which the nodes are partitioned.
The graph in Figure 7.1 is not bipartite as it is not possible to decompose the nodes labelled with $u, x$ and $w$ into two disjoint sets such that no two nodes within the same set are adjacent.

![Bipartite Graph](image)

**Figure 7.2:** A bipartite graph

**Definition 21 (Attachment and Piece)** Let $C$ be a cycle in an undirected, biconnected graph $G$. Then, a node $v$ in $C$ is called an attachment if $v$ is the source or target of an edge that is not in $C$. The pieces of $G$ with respect to $C$ are the connected components between $C$'s attachments in which $G$ is decomposed if all edges in $C$ are deleted (the attachments divide the connected components).

![Graph with Attachments and Separating Cycle](image)

**Figure 7.3:** An example graph for attachments and a separating cycle
A piece of an undirected, biconnected graph $G$ has at least two attachments. A cycle $C$ is called *separating* if $G$ has at least two pieces with respect to $C$. For example, consider the undirected, biconnected graph in Figure 7.3 in which a cycle $C$ is marked by the highlighted edges. The highlighted nodes in $C$ are the attachments in $C$ as only these nodes are source or target of an edge that is not in the cycle. The graph in Figure 7.3 has two pieces which are shown in Figure 7.4 (highlighted nodes and edges) together with $C$. The cycle in Figure 7.3 is a separating cycle since the graph has two pieces with respect to that cycle. The cycle highlighted in Figure 7.5 is a non-separating cycle as the remaining graph is a single piece.

![Diagram](image)

**Figure 7.4:** The pieces A and B of the graph in Figure 7.3 together with the cycle in Figure 7.3

**Definition 22 (Interlacement Graph)** Let $C$ be a cycle in an undirected, biconnected graph $G$ such that all nodes in $C$ are ordered along $C$. Let $P$ and $P'$ be pieces of $G$ with respect to $C$ with attachments $p_1$ and $p_2$ of $P$ and $p'_1$ and $p'_2$ of $P'$. Then $P$ and $P'$ *interlace* if one of the paths along $C$ meets the attachments of $P$ and $P'$ in order $p_1, p'_1, p_2, p'_2$. An *interlacement graph* $I$ of $G$ with respect to a cycle $C$ is a graph whose nodes each represent a piece of $G$ with respect to $C$ and in which two nodes $u$ and $v$ are adjacent if pieces $u$ and $v$ interlace.
The pieces of the graph in Figure 7.3 do not interlace as the attachments of piece B are equivalent to two attachments of piece A and there is no other attachment of piece A meeting the cycle between these attachments. However, if the graph in Figure 7.3 is changed to the graph in Figure 7.6 by redirecting one of the edges of piece B, the two pieces interlace. The interlacement graph of the graph in Figure 7.6 is shown in Figure 7.7, where the nodes of the graph represent the two pieces of the graph in Figure 7.6 and an edge between the nodes is inserted because the two pieces of Figure 7.6 interlace. The interlacement graph of the graph in Figure 7.3 is the graph in Figure 7.7 in which the edge between the two nodes of the interlacement graph does not exist.

By Definition 22, two pieces interlace if they cannot be drawn on the same side of the cycle without violating planarity.

**Definition 23 (Planarity)** An undirected graph is planar if it can be drawn on the plane without edge crossings.

A formalisation of the definition of planarity used in Definition 23 is given in [HT74]:

![Figure 7.5: A non-separating cycle](image)
An undirected graph $G$ is \textit{planar} if and only if there exists a mapping of nodes and edges of $G$ into the plane such that (1) each vertex is mapped into a distinct point, (2) each edge $e$ between nodes $u$ and $v$ is mapped onto a simple curve, with $u$ and $v$ mapped onto the endpoints of the curve, and (3) mappings of distinct edges have only the mappings of their common endpoints in common.

\textbf{Figure 7.6:} A graph with interlacing pieces

\textbf{Figure 7.7:} Interlacement graph of the graph in Figure 7.6

Consider as an example of an undirected, bipartite, planar graph the graph in Figure 7.8. We have already drawn the graph without crossing edges for better comprehension. An example of a non-planar graph is shown in Figure 7.9. The graph in Figure 7.9 is not planar as the edges cannot be drawn without crossing.
Figure 7.8: A planar graph

7.2 The Auslander-Parter Planarity-Test Algorithm

The Auslander-Parter algorithm [AP61] is the first efficient planarity-test algorithm introduced in the literature. It is corrected by Goldstein in [Gol63] because the original version may loop indefinitely. The algorithm uses the divide and conquer principle. The idea is that an undirected, connected graph is planar if the pieces with respect to a cycle in the graph can be drawn inside or outside the cycle without crossing edges (bipartite interlacement graph of the pieces). The problem is recursively broken down into sub-problems by testing each piece attached to the cycle for planarity separately. The smallest problem is the one where all pieces are (simple) paths. The Auslander-Parter algorithm was the first algorithm to not only test planarity but also give an instruction on how to construct a representation of a planar embedding of the graph. The first implementation of the algorithm is given by Shirey [Shi69]. The program has a run-time of $O(|V_G|^3)$, where $G$ is the input graph. The most efficient algorithm due to the Auslander-Partner method tests planarity in linear time and is introduced by Hopcroft
Figure 7.9: A non-planar graph

and Tarjan in [HT74]. The algorithm in [HT74] is highly dependant on other data structures, for example stacks and path lists. As these would need to be artificially encoded into the input graph in GP⁺, we prefer using the original algorithm in [AP61].

Given an undirected, connected graph \( G \) and a cycle \( C \) in \( G \), Auslander and Parter prove that \( G \) is planar if and only if the following two conditions are satisfied [AP61]:

(A) The interlacement graph of the pieces of \( G \) with respect to \( C \) is bipartite.

(B) For each piece \( P \) of \( G \) with respect to \( C \), the graph obtained by adding \( P \) to \( C \) is planar.

A graph is planar if and only if all its biconnected components are planar [Ber62]. By Euler’s theorem, a planar graph with \( n \) nodes has at most \( 3n - 3 \) edges. Thus, we may assume that the input graph for the algorithm is an undirected, biconnected graph with at most \( 3n - 3 \) edges.

Conditions (A) and (B) led to the algorithm proposed in [AP61, Gol63]. The algorithm is reformulated in [DETT99] in our terminology and is given
in Figure 7.10. The GP$^+$ program PlanarityTest in Section 7.3 is based on the algorithm given in Figure 7.10.

1. Compute a separating cycle $C$ of the input graph $G$.

2. Compute the pieces of $G$ with respect to $C$.

3. For each piece $P$ of $G$ with respect to $C$ that is not a path (of one or more edges):
   (a) let $P'$ be the graph obtained by adding $P$ to $C$ 
   (b) let $C'$ be the cycle of $P'$ obtained from $C$ by replacing the portion of $C$ between two consecutive attachments with a path of $P$ between them
   (c) apply the algorithm starting from 2. recursively to graph $P'$ and cycle $C'$. If $P'$ is non-planar, return “non-planar”.

4. Compute the interlacement graph $I$ of the pieces.

5. Test whether $I$ is bipartite. If $I$ is not bipartite, return “non-planar”.

6. Return “planar”.

Figure 7.10: The Auslander-Parter planarity-test algorithm

The Auslander-Parter algorithm first constructs an initial separating cycle in an undirected, biconnected graph with at most $n$ nodes and $3n - 3$ edges and then checks if conditions (A) and (B) are satisfied by recursively applying the algorithm to the pieces of the input graph and the computed cycle. The algorithm in Figure 7.10 differs from the one in [AP61] in that it always constructs a separating cycle. However, it is not necessary to construct a separating cycle in step 1 of the algorithm as the problem of finding a separating cycle is then deferred to step 3 of the algorithm. It is proven in [DETT99] that if a piece of $G$ with respect to $C$ is not a path
(of one or more edges), then $G$ has a separating cycle consisting of a sub-path of $C$ together with a path of $P$ between two attachments. Therefore, each recursive invocation of the algorithm on a cycle and a non-path piece yields a new cycle with at least two (smaller) pieces that are further analysed in the next recursive invocation. It is shown in [DETT99] that the above algorithm terminates in $O(|V_G|^3)$ time where $G$ is the input graph.

**Example 12 (Application of the Algorithm)** Consider the example graph in Figure 7.8 for an application of the algorithm in Figure 7.10 to an undirected, bipartite graph with $n$ nodes and at most $3n - 3$ edges. Let step 1 of the algorithm mark a separating cycle by labelling it with 1 (see thick edges in the input graph in Figure 7.11a). Step 2 then marks the pieces of the input graph with respect to the cycle. These are the nodes and edges labelled with $k$ where $k > 1$ in graph A in Figure 7.11a. Let us now consider the case where the piece labelled 4 is chosen in step 3 of the algorithm (the only other option is the piece labelled 5). Then step (3a) adds that piece to the original cycle and step (3b) selects a new separating cycle by choosing a path through the piece and adding a part of the original cycle (see the bottom left graph in Figure 7.11a). Step 2 of Figure 7.10 is then applied again to the newly formed graph resulting, for example, in graph B in Figure 7.11a. As all of these new pieces are (simple) paths, the algorithm proceeds with step 4 by constructing the interlacement graph in the middle of Figure 7.11c which is obviously bipartite. Next, step 3 of the algorithm in Figure 7.10 is applied recursively on the piece labelled with 5 in graph A in Figure 7.11a. The result of this application is shown in Figure 7.11b. Step 2 of Figure 7.10 is then applied again to the newly formed graph resulting, for example, in graph C in Figure 7.11b. As all of these new pieces are again (simple) paths, the algorithm proceeds with step 4 by constructing the right interlacement graph in Figure 7.11c which is obviously bipartite. Step 4 of the algorithm is then applied to the original graph A in Figure 7.11a which results in the left interlacement graph in Figure 7.11c. It is obvious that this interlacement graph is also bipartite. Thus, the input graph is planar. \[\square\]
Figure 7.11a: An example run of the Auslander-Parter algorithm
Figure 7.11b: An example run of the Auslander-Parter algorithm (continued)

Figure 7.11c: An example run of the Auslander-Parter algorithm (interlacement graphs)
To implement the above algorithm, several sub-problems have to be solved. The main sub-problems are: (1) compute a separating cycle in a graph and mark it, (2) compute the pieces with respect to the cycle, (3) compute a new graph formed by a cycle and a piece, (4) compute a separating cycle out of a cycle and a piece, (5) compute an interlacement graph and (6) check if a graph is bipartite. These sub-problems are solved by several GP+ procedures. We discuss these procedures and the GP+ program PlanarityTest in the following section.

7.3 The GP+ Program PlanarityTest

The structure of the original Auslander-Parter algorithm in Figure 7.10 is mapped to the program PlanarityTest in Figures 7.14a to 7.14c. We annotate the commands in Figure 7.14a with comments, enclosed in '%' symbols, relating each command with a step in the Auslander-Parter algorithm in Figure 7.10.

To improve readability of rule schemata in PlanarityTest in cases where large (pattern) graphs are matched but only minor changes result from a rule-schema application, we introduce the following diagrammatic convention.

Assumption 6 (Diagrammatic Convention) For this chapter, we assume without loss of generality, that every graph part of a rule-schema declaration of the form \( L \Rightarrow R \), where \( \text{Graph}(L) = \text{Graph}(M) \cup \text{Graph}(L') \) and \( \text{Graph}(R) = \text{Graph}(M) \cup \text{Graph}(R') \) with \( \cup \) being the disjoint union of two graphs, may be written as \[ \begin{array}{c}
M \\
\downarrow
\end{array} \overset{L'}{\Rightarrow} R'. \] If \( L' \) and \( R' \) are the empty graphs, \[ \emptyset \\
\downarrow
\] may be written as \[ \Rightarrow \] with empty left- and right-hand side.

For example, by Assumption 6, the rule schema in Figure 7.12 may be replaced with the rule schema in Figure 7.13. The common subgraph (frame graph) of the graphs in the left- and right-hand side is written above the \( \Rightarrow \) symbol to indicate that this graph part is indeed not altered. Hence,
Assumption 6 improves readability of rule schemata if large pattern graphs, i.e. graphs that are needed for matching but are not altered by the rule, are involved.

$$\text{Reduce}(x, y, z : \text{int}) =$$

$$\begin{array}{c}
\text{x} \\
1
\end{array} \begin{array}{c}
\text{y} \\
2
\end{array} \begin{array}{c}
\text{z} \\
3
\end{array} \quad \Rightarrow \quad \begin{array}{c}
\text{x} \\
1
\end{array} \begin{array}{c}
\text{y} \\
2
\end{array} \begin{array}{c}
\text{z} \\
3
\end{array}, \text{x+y}$$

where $x + y < z$

**Figure 7.12:** A conditional rule schema

$$\text{Reduce}(x, y, z : \text{int}) =$$

$$\begin{array}{c}
\text{x} \\
1
\end{array} \begin{array}{c}
\text{y} \\
2
\end{array} \begin{array}{c}
\text{z} \\
3
\end{array} \\
\text{z} \quad \Rightarrow \quad \text{x+y}$$

where $x + y < z$

**Figure 7.13:** The conditional rule schema of Figure 7.12 according to Assumption 6

The input graph for PlanarityTest is an undirected, biconnected, untagged, integer-labelled graph with $n$ nodes and at most $3n - 3$ edges. PlanarityTest inserts a node with label “No” if the input graph is not planar and it inserts a node with label “Yes” if the input graph is planar.

The Auslander-Parter algorithm in Figure 7.10 uses recursion to call the planarity-test on some subgraph. Recursion is realised in GP+ by recursive procedure calls. We use an if command whenever we want to keep the current graph structure for further analysis but need to apply a destructive procedure to continue the execution on a subgraph (see step 3 of the original algorithm in Figure 7.10). This approach demonstrates again the advantages of the if command in GP and GP+. The use of an if command
provides a way to continue the algorithm on a subgraph of the input graph without destroying the original graph permanently. With the combination of recursive procedure calls and the if command, the divide and conquer principle of the Auslander-Parter algorithm can be modelled.

The main part of the program PlanarityTest in Figure 7.14a consists of the main command sequence and the procedures Planar and RunRec both with an empty list of rule-schema declarations. The rule schemata Insert, NotPath, SelectAtt and PieceExists (Figure 7.14b and Figure 7.14c) are either used directly in the main command sequence or as global rule schemata in one of the procedures in the main program. The procedures Untag and DelRest (Figure 7.14b and Figure 7.14c) transform the graph in preparation for the application of some other procedure. Untag removes all tags that do not match the actual parameter of an Untag call. DelRest deletes all nodes and edges whose tags do not match the cycle tag (0) or the actual parameter of a DelRest call. Procedure Untag demonstrates that GP+ procedures enhance the reusability of program parts for different purposes. Untag is applied to different subgraphs, distinguished by their tags, in each call in the program. Therefore, Untag is broadly applicable.

The procedures CompCycle, CompPieces, SepCycle, CompInter and Bipartite solve the sub-problems of the Auslander-Parter algorithm in Figure 7.10. These procedures are discussed in Sections 7.3.1 to 7.3.5. CompCycle computes the first cycle in the input graph according to step 1 of the original algorithm. CompPieces tags the pieces with respect to a cycle with distinct integers according to step 2 of the original algorithm. SepCycle constructs a separating cycle for a given cycle and piece that is not a path according to steps (3a) and (3b) of the original algorithm. CompInter computes the interlacement graph for the currently analysed cycle and pieces according to step 4 of the algorithm and Bipartite tests if the interlacement graph is bipartite according to step 5 of the original algorithm in Figure 7.10.
main = CompCycle; % step 1 %
if Planar % steps 2–5 %
  then Insert("Yes") % return "planar" %
else (Insert("No")); % return "non-planar" %
  Untag(0); Untag(1). % undo changes %

Planar = CompPieces(2); % step 2 %
  RunRec(2); % step 3 %
  CompInter; % step 4 %
  Bipartite. % step 5 %

RunRec(k:int) = if NotPath(k) then % condition in step 3 %
  (if DelRest(k); % step 3a; deletes all nodes and edges not tagged k or 0 %
    SepCycle; % step 3b %
    Planar % recursive call of steps 2–5 in step 3c %
    then RunRec(k + 1) % step 3; piece with next higher number %
  else (fail) % graph not planar because Bipartite fails %
  else % if piece is a path or does not exist %
    (if PieceExists(k) % piece exists and is a (simple) path %
      then (RunRec(k + 1))); % step 3 %

Figure 7.14a: The program PlanarityTest
Insert(k: string) =

\[
\emptyset \quad \Rightarrow \quad k
\]

NotPath(k, x, y, z, r, s, t, p, q, u: int) =

\[
\begin{array}{c}
 s_k \quad y_k \\
 x_k \quad z_k \\
 r_p \quad t_q
\end{array}
\]

\[\Rightarrow\]

Untag(k: int) =

\[
\begin{cases}
\text{Edges}(k, x, y, z, p, q, j: int) = & \begin{align*}
 x_p & \quad y_j \\
 x_p & \quad z_q
\end{align*} \\
\text{Nodes}(k, x, j: int) = & \begin{align*}
 x_j
\end{align*}
\end{cases}
\]

\[
\begin{align*}
 & \text{where } j \neq k \\
 & \text{where } j \neq k
\end{align*}
\]

Edges(k)!; Nodes(k)!

**Figure 7.14b:** *The program PlanarityTest (continued)*
\[
\text{DelRest}(k:\text{int}) = \\
\begin{cases} \\
\text{DelEdge}(k, x, y, z, p, q, j:\text{int}) = \\
\begin{array}{c}
x_p \\
1 \\
y_j \\
j \\
z_q \\
2 \\
\end{array} \quad \Rightarrow \quad \\
\begin{array}{c}
x_p \\
1 \\
z_q \\
2 \\
\end{array} \\
\text{where } j \leq k \text{ and } j > 0 \\
\end{cases}
\]

\[
\text{DelNode1}(x, j:\text{int}) = \\
\begin{array}{c}
x_j \\
1 \\
\end{array} \quad \Rightarrow \quad \emptyset
\]

\[
\text{DelNode2}(x:\text{int}) = \\
\begin{array}{c}
x_0 \\
1 \\
0 \\
\end{array} \quad \Rightarrow \quad \emptyset
\]

\[
\text{DelEdge}(k)!, \{\text{DelNode1, DelNode2}\}!.
\]

\[
\text{SelectAtt}(k, x, y, z, p:\text{int}) = \\
\begin{array}{c}
x_0 \\
1 \\
y_k \\
2 \\
z_p \\
\end{array} \quad \Rightarrow \quad \\
\begin{array}{c}
x_1 \\
1 \\
y_k \\
2 \\
z_p \\
\end{array}
\]

\[
\text{where } k > 0
\]

\[
\text{PieceExists}(k, x, y, z, p, q:\text{int}) = \\
\begin{array}{c}
x_p \\
1 \\
y_k \\
2 \\
z_q \\
\end{array} \quad \Rightarrow \quad \\
\begin{array}{c}
x_1 \\
1 \\
y_k \\
2 \\
z_p \\
\end{array}
\]

\textbf{Figure 7.14c:} The program PlanarityTest (continued)
The main structure of the algorithm in Figure 7.10 is mapped to the program structure of PlanarityTest. Here, steps 2 to 5 of the original algorithm are mapped to the execution of Planar which first computes the pieces (CompPieces), then simulates step 3 by applying RunRec, and finally computes the interlacement graph (CompInter) and tests if the interlacement graph is bipartite (Bipartite). The ‘return’ part of the original algorithm is mapped to termination with failure or a graph of the execution of Planar or Bipartite, respectively.

The for-loop in step 3 of the original algorithm and the recursive call of the planarity test are modelled by RunRec. When called, the if condition first tests if a piece with number $k$ is a simple path (NotPath). If it is, the else-branch of the if command in RunRec is executed testing first if there is a piece with number $k$. If there is such a piece, RunRec is executed on the next higher number. If the piece with number $k$ is not a simple path, the then-branch of the if command is executed, simulating steps (3a) to (3c) on the current piece. The recursive call in step (3c) is simulated by calling Planar in the then-branch of the if command. The recursive call of Planar is executed within an if command to be able to apply the algorithm on a subgraph of the input graph (see steps (3a) to (3c) of the algorithm in Figure 7.10) without permanently destroying the input graph and thereby loosing information.

The then-branch of the if command in procedure RunRec is only executed if the condition (DelRest(k); SepCycle; Planar) does not finitely fail on its input. The latter can only be the case if Bipartite terminates with failure for all its inputs. This is due to the fact that all other procedure executions must terminate with a graph. CompPieces must terminate with a graph because the if command executes skip if all pieces are tagged. AddInter is always applicable and TagPiece and CompInter always terminate with a graph because of their structure (see Section 7.3.2 and Section 7.3.4).

After RunRec has terminated with a graph, the interlacement graph is constructed for the graph of the current recursive invocation of Planar. Then, the bipartite test is executed. If the procedure Bipartite terminates with a
graph for all input graphs computed in the recursive invocations of Planar, then Planar also terminates with a graph. Thus, the then-branch of the if command in the main command sequence is executed which inserts a node with label “Yes”.

If Bipartite terminates with failure for some input, the current invocation of RunRec terminates with failure. Thus, Planar terminates with failure. If Planar finitely fails on its input, the else-branch of the if command in the main command sequence of PlanarityTest is executed which inserts a node with label “No”.

Assuming that the procedures solving the sub-problems of the original algorithm in Figure 7.10 are correct, the main program PlanarityTest correctly models the structure of the algorithm in Figure 7.10. We discuss the remaining procedures in the next subsections.

7.3.1 The Procedure CompCycle

The procedure CompCycle in Figure 7.15a simulates step 1 of the original algorithm in Figure 7.10. CompCycle tags a cycle with integer 0 in an undirected, biconnected, integer-labelled, untagged input graph with \( n \) nodes and at most \( 3n - 3 \) edges. Here, CompCycle does not necessarily find a separating cycle. Since the problem of finding any cycle in a graph is easier to solve than finding a separating cycle, we defer the problem of finding a separating cycle, if necessary, to the procedure SepCycle.

The procedure CompCycle first uses procedure DFS in Figure 7.15b (discussed in Section 6.1) to mark a spanning tree in the input graph by tagging all nodes and edges of the tree. To mark the first edge in the cycle, the rule TagEdge tags an untagged edge and the incident node with smaller tag with \( 0 \). It is obvious that tagging any untagged edge in the graph will close a path to form a cycle because both incident nodes of such an edge are reachable from the root node (node with tag 1) of the spanning tree. Having tagged the first edge of the cycle, TagCycle then tags the rest of the cycle.
CompCycle =

(Select(x:int) =

\[
\begin{align*}
\text{x} & \quad \Rightarrow \quad x_{-1} \\
\end{align*}
\]

TagEdge(x,y,z,p,q: int) =

\[
\begin{align*}
x_{-p} & \quad \quad y & \quad \quad z_{-q} \\
\end{align*}
\]

\[
\begin{align*}
\Rightarrow & \quad \quad x_{-0} & \quad \quad y_{-0} \quad \quad z_{-q} \\
\end{align*}
\]

where \( p < q \)

TagCycle(x,y,z,p,q,k,j: int) =

\[
\begin{align*}
y_{-k} & \quad \quad p_{-k} & \quad \quad z_{-j} \\
\end{align*}
\]

\[
\begin{align*}
\Rightarrow & \quad \quad y_{-0} & \quad \quad p_{-0} \quad \quad z_{-j} \\
\end{align*}
\]

where \( k > 0 \)

Select; DFS(1); TagEdge; TagCycle!; Untag(0).

Figure 7.15a: The procedure CompCycle
DFS(k:int) =

\[
\begin{align*}
\text{ChildExists}(k,x,y,z:\text{int}) = & \\
\to & \\
\text{Visit}(k,x,y,z:\text{int}) = & \\
\to & \\
\text{while ChildExists}(k) \text{ do } (\text{Visit}(k); \text{DFS}(k+1)).
\end{align*}
\]

**Figure 7.15b: The procedure DFS**

The procedure DFS uses a depth-first search approach to tag the nodes and edges of a spanning tree. The particular tagging imprints an order on the nodes and edges of the spanning tree. Every node visited earlier in the search has a smaller tag than the later visited nodes. The root node of the tree is tagged with 1. Thus, the path in which nodes and edges were consecutively tagged is uniquely identifiable by following the path from a node with the higher number back to a root with a smaller number. Note that the other direction is not unique as different branches can be chosen.

We use this trick in the rule schema TagCycle to follow a path backwards to a particular parent node and tag the path with 0. TagCycle terminates when the node relabelled by TagEdge is reached as at this point, there will be no pair of edge and node with the same tag adjacent to a node tagged with 0.

TagEdge is applicable as otherwise the input graph would be a tree and therefore not biconnected. If Visit was indeed applied to the next adjacent,
untagged node, then one of the tags in the left-hand side graph of TagEdge must be smaller. Thus, TagEdge; TagCycle! computes a cycle in the input graph by tagging all nodes and edges on the cycle with 0.

DFS terminates because every application of Visit reduces the number of untagged nodes and edges in the graph. TagCycle! terminates because every application TagCycle reduces the number of nodes and edges tagged with integers greater than 0 in the graph. Untag terminates because every application of a rule schema in Untag reduces the number of tagged nodes and edges in a graph.

Example 13 (CompCycle Example Run) Figure 7.16 shows one of the possible runs of CompCycle. The first part of the procedure (Select; DFS(1)) tags the nodes and edges of a spanning tree of the input graph (thick drawn and tagged nodes and edges in the second graph). In Figure 7.16 TagEdge tags the edge between the node tagged 1 and the node tagged 6 to form a cycle. The result of the execution of TagCycle!; Untag(0) is shown in the final graph of Figure 7.16. Note that the result of TagCycle!; Untag(0) is deterministic. This is because TagEdge always relabels the incident node with smaller tag with 0 and the left-hand graph of TagCycle only allows the child-to-parent connection impressed by the order of the spanning tree to be relabelled. Thus, if an edge is chosen by TagEdge, the result of TagCycle! is deterministic.

Procedures are not implemented in the GP environment. However, we are able to test the second part of CompCycle, namely TagEdge; TagCycle! on the second graph in Figure 7.16 as TagEdge and TagCycle are rule schemata. Run in backtracking mode, the GP system produces exactly 8 different output graphs, that is one for each of the 8 untagged edges in the graph. This shows, that our program works as intended, assuming that DFS works correctly.
Figure 7.16: An example run of CompCycle, where $P_1 = \text{Select; DFS}(1)$ and $P_2 = \text{TagEdge; TagCycle!; Untag}(0)$
In general, there are more than 8 cycles in the graph in Figure 7.8. The procedure CompCycle does not compute all of these because only tagged edges of the spanning tree are relabelled with tag 0 by TagCycle. However, the procedure marks one cycle which is sufficient for the beginning of the algorithm.

Note that we do not show results of the test runs on the GP system in data in this chapter. The programs in the GP system are equivalent to the ones presented here. The results of one run in backtracking mode in average results in 200 graphs. Some of these graphs are equivalent to the result graphs in the examples presented in each section, some of them are isomorphic. Presenting all 200 graphs or even the polymorphic ones would exceed the limits of this thesis.

7.3.2 The Procedure CompPieces

The procedure CompPieces in Figure 7.17a simulates step 2 of the Auslander-Parter algorithm in Figure 7.10 by tagging all pieces in the graph with distinct integers. CompPieces is executed on an undirected, biconnected, integer-labelled input graph with \( n \) nodes and at most \( 3n - 3 \) edges in which the nodes and edges of a cycle are tagged with 0 and all other nodes and edges are untagged. First, CompPieces tests if there is an untagged edge or node in the graph (PieceLeft), meaning that there exists an untagged piece. If this is the case, AddInter inserts an interlacement-graph node as a preparation for the procedure CompInter. This node is labelled with the integer passed on to AddInter by the current recursive invocation of CompPieces. Next, TagPieces tags one piece in the graph with the actual parameter \( k \) of the call, where \( k > 1 \) (see call of CompPieces in Figure 7.14b). The tag of a piece is unique in the graph as TagPieces is called with the next higher integer each time. Note that PieceLeft does not need to check for untagged edges incident to untagged nodes because all pieces in the form of a graph contain at least one node.
CompPieces(k:int) =

\[
\begin{align*}
\text{AddInter}(k:int) = \\
\{ 0 \} & \implies (k_0, 0)
\end{align*}
\]

if PieceLeft then (AddInter(k); TagPiece(k); CompPieces(k + 1)).

PieceLeft =

\[
\begin{align*}
\text{EdgeLeft}(x,y,z:int) = \\
\begin{array}{c}
\text{Diagram:}
\end{array} & \implies \\
\quad x_0 & y & z_0 \\
\quad 1 & \quad & 2
\end{align*}
\]

\[
\begin{align*}
\text{NodeLeft}(x:int) = \\
\begin{array}{c}
\text{Diagram:}
\end{array} & \implies \\
\quad x \\
\quad 1
\end{align*}
\]

{EdgeLeft, NodeLeft}.

**Figure 7.17a:** The procedure CompPieces
TagPiece(k:int) =

\[
\begin{align*}
\text{TagEdge}(k, x, y, z: \text{int}) &= \\
x_0 &\quad y &\quad z_0 \\
1 &\quad 2 &\quad 2 \\
\Rightarrow &\quad x_0 &\quad y_k &\quad z_0 \\
1 &\quad 2 &\quad 2 \\
\end{align*}
\]

TagFirst(k,x,y,z:int) =

\[
\begin{align*}
\text{TagNext}(k, x, y, z: \text{int}) &= \\
x_0 &\quad y &\quad z \\
1 &\quad 2 &\quad 2 \\
\Rightarrow &\quad x_0 &\quad y_k &\quad z_k \\
1 &\quad 2 &\quad 2 \\
\end{align*}
\]

TagMid(k,x,y,z:int) =

\[
\begin{align*}
\text{TagLast}(k, x, y, z: \text{int}) &= \\
x_k &\quad y &\quad z_0 \\
1 &\quad 2 &\quad 2 \\
\Rightarrow &\quad x_k &\quad y_k &\quad z_0 \\
1 &\quad 2 &\quad 2 \\
\end{align*}
\]

try TagEdge(k) then skip else (TagFirst(k);

\{TagNext(k), TagLast(k), TagMid(k)\}).

**Figure 7.17b:** The procedure TagPiece
TagPiece first tries to tag a piece that consists only of an edge connecting two attachments (TagEdge). If TagEdge is applicable, TagPiece terminates and is called again by CompPieces with the next higher integer. If TagEdge is not applicable, then no simple piece exists. In that case, TagFirst tags the first untagged edge and untagged incident node of a piece consisting of at least two edges and one node. The selected edge is incident to an attachment. Then, the remaining part of the piece is tagged by applying the rule-set call \{TagNext(k,\_\_\_\_\_\_), TagLast(k,\_\_\_\_\_), TagMid(k,\_\_\_\_\_\_)\} as long as possible. Here, TagNext tags every adjacent untagged node of a node tagged with the current value of the parameter, TagLast tags the connections of the piece with the cycle and TagMid tags the remaining untagged edges between two tagged nodes of the piece. The latter case exists if the piece contains a cycle.

Note that there indeed exists an untagged piece in the graph as PieceLeft was applicable. The piece must be connected with the cycle since the input graph is biconnected. The rules in the rule-set call in TagPiece only tag nodes and edges connected with the node selected by TagFirst because only nodes tagged with the value of the actual parameter are matched in the rule schemata in the set. TagPiece terminates because every application of a rule schema in TagPiece reduces the number of untagged nodes and edges in the graph. CompPieces terminates because every execution of TagPiece reduces the number of untagged pieces in the graph.

**Example 14 (TagPiece Example Run)** Figure 7.18 shows a run of TagPiece on an input graph with two untagged pieces. Let TagPiece(4) be a call of TagPiece in CompPieces. Then, the second graph in Figure 7.18 represents an intermediate graph in the execution of TagPiece(4) in which the top left node of the cycle is selected by TagFirst to tag the first node of the piece in the centre of the graph with 4. Then, TagNext and TagLast are applied to form the tagged, highlighted subgraph of the selected piece. The final graph in Figure 7.18 is the result of a possible run of TagPiece(4). Note that the result of TagPiece(4) is deterministic once TagFirst has tagged a node and edge of a piece. □
Figure 7.18: An example run of TagPiece, where $P_1 = \text{TagFirst}(4) \cup \text{TagNext}(4)$; TagLast($4$) and $P_2 = \{\text{TagNext}(4), \text{TagLast}(4), \text{TagMid}(4)\}$!
As `CompPieces` is a procedure, `CompPieces` could not be tested on the GP system. To test `TagPiece` on the GP system, every variable identifier k in the local rule schemata needs to be replaced with a numeral. This is the same as semantically evaluating a parameterised procedure or parameterised rule-schema call. For the test, we replaced the variable identifier k with the numeral 4 in `TagPiece`. Running `TagPiece` on the GP system in the backtracking mode on the input graph in Figure 7.18 results in hundreds of graphs isomorphic to either the result in Figure 7.18 or a graph in which the remaining untagged piece of Figure 7.18 is tagged with 4. Hence, we will assume that the procedure computes the expected results.

### 7.3.3 The Procedure SepCycle

The procedure `SepCycle` in Figure 7.19a and Figure 7.19b simulates step (3b) of the original algorithm in Figure 7.10. `SepCycle` tags a separating cycle (with 0) in an undirected, biconnected input graph in which a cycle is tagged with 0 and that contains a single piece that is not a path and is tagged with an integer greater than 1. We may assume that every input graph for `SepCycle` that is computed in `PlanarityTest` has this form because of the following: (1) all nodes and edges except for the nodes and edges in the current piece and cycle are deleted by `DelRest`, (2) the piece is not a path because `NotPath` was applicable (see Figure 7.14b), and (3) a cycle is tagged with 0 because it was either computed by `CompCycle` or by `SepCycle` in some former recursive invocation of `SepCycle`.

The procedure `SepCycle` is split into several tasks. These tasks are chosen according to the separating cycle construction given in [DETT99]. It is proven in [DETT99], that there exists a separating cycle for any cycle and a piece that is not a path. In the proof, an algorithm for constructing a separating cycle is given.
SepCycle =
\[
\begin{align*}
\text{DelTag}(x,y,z; \text{int}) = \\
\begin{array}{c}
\text{SelectPath; TagCyclePath; DelTag!; Untag(1); Relabel.}
\end{array}
\end{align*}
\]

SelectPath =
\[
\begin{align*}
\text{TagNext}(x,y,z,p; \text{int}) = \\
\text{where } p > 1
\end{align*}
\]

TagLast(x,y,z,p; int) =
\[
\text{where } p > 1
\]

Backtrack(x,y,z; int) =
\[
\text{SelectAtt; \{TagNext, TagLast\}!;}
\]

Backtrack; \{TagNext, TagLast\}!).

**Figure 7.19a: The procedure SepCycle**
TagCyclePath =

\[
\begin{aligned}
\text{TagPath}(x, y, z: \text{int}) &= \ \\
\begin{array}{c}
\circlearrowright x_1 y_0 z_0 \\
1 & 2
\end{array} \quad \Rightarrow \quad \begin{array}{c}
\circlearrowright x_1 y_1 z_1 \\
1 & 2
\end{array}
\end{aligned}
\]

TagLast(x, y, z: int) =

\[
\begin{aligned}
\begin{array}{c}
\circlearrowright x_1 y_0 z_1 \\
1 & 2
\end{array} \quad \Rightarrow \quad \begin{array}{c}
\circlearrowright x_1 y_1 z_1 \\
1 & 2
\end{array}
\end{aligned}
\]

Select(x: int) =

\[
\begin{aligned}
\begin{array}{c}
\circlearrowright x_0 \\
1
\end{array} \quad \Rightarrow \quad \begin{array}{c}
\circlearrowright x_1 \\
1
\end{array}
\end{aligned}
\]

try SelectAtt then \{TagPath, TagLast\}! else

(Select; \{TagPath, TagLast\}!).

Relabel =

\[
\begin{aligned}
\text{Edge}(x, y, z: \text{int}) &= \ \\
\begin{array}{c}
\circlearrowright x_1 y_1 z_1 \\
1 & 2
\end{array} \quad \Rightarrow \quad \begin{array}{c}
\circlearrowright x_1 y_0 z_1 \\
1 & 2
\end{array}
\end{aligned}
\]

Node(x: int) =

\[
\begin{aligned}
\begin{array}{c}
\circlearrowright x_1 \\
1
\end{array} \quad \Rightarrow \quad \begin{array}{c}
\circlearrowright x_0 \\
1
\end{array}
\end{aligned}
\]

Edge!; Node!.

**Figure 7.19b:** *The procedure SepCycle (continued)*
First, a path is selected that connects two attachments along the nodes and edges of the piece. Then, a path in the cycle between these attachments is added to this path. If the piece has more than two attachments, the path in the cycle with at least one of the remaining attachments is chosen. This guarantees, that there is both a piece formed by the remaining parts of the original piece and another piece formed by the remaining path in the former cycle. Note that a piece has at least two attachments since the input graph is biconnected. There is a path connecting two attachments in a piece because the piece is connected.

An example of a separating cycle is shown in Figure 7.20 where the original cycle (thick lines) and piece is shown in the left-hand graph and a newly formed separating cycle (thick lines) is shown in the right-hand graph. Observe, that the new cycle would not form a separating cycle if the other part (without attachment) of the path along the original cycle had been selected.

**Figure 7.20:** Separating-cycle construction for a piece with three attachments

If the piece only has two attachments, any of the two paths on the cycle may be chosen. In this case, the path in the piece is already separating. An example of this is shown in Figure 7.21. Here, the original cycle (thick lines)
and the piece is depicted in the left-hand graph and a newly formed separating cycle (thick lines) is shown in the right-hand graph.

Figure 7.21: Separating-cycle construction for a piece with two attachments

As the algorithm was proven correct in [DETT99], we need only show that SepCycle implements it correctly.

The procedure SelectPath tags a path in the piece with 1. This integer is a unique tag number since all pieces are tagged with integers greater than 1. SelectPath uses backtracking whenever the explored path does not connect two attachments. A pointer node, labelled with the unique tag 1_1, is used to explore a path starting from a non-deterministically chosen attachment (SelectAtt) in a depth-first fashion. If the non-deterministically explored path reaches the second attachment, the second tag of the pointer node is deleted by TagLast and thus, the pointer is deleted. If TagNext and TagLast are not applicable anymore, {TagNext, TagLast}! terminates and (Backtrack; {TagNext, TagLast}! is executed. Backtrack is not applicable, if the pointer node does not exist, meaning that TagLast was applied and the path is complete. In that case, the outer iteration ('!') will terminate. If Backtrack is applicable, the explored path meets in a dead end. In this case, Backtrack models backtracking by undoing the last TagNext
application. The dead end is relabelled with an additional tag so that this path cannot be explored again by \texttt{TagNext}. Then, \{\texttt{TagNext, TagLast}\} is executed again.

\texttt{SelectPath} terminates as each rule-schema application (except for \texttt{Backtrack}) reduces the number of tags that are larger than 1 in the graph. In addition, \texttt{Backtrack} is only applicable once to every node or edge tagged with 1 because of the additional second tag 2 it adds to the node. The procedure is correct because only connected pieces are considered by the rule schemata and backtracking is triggered in case the path reaches a dead end.

The procedure \texttt{TagCyclePath} tags one of the two paths in the cycle between the two attachments of the path in the piece. If possible, \texttt{SelectAtt} selects a third attachment (tagged with 0) by tagging it with 1\_1. Then, \texttt{TagPath} spreads this tag along the path in the cycle until it meets the marked attachments (\texttt{TagLast}). If \texttt{SelectAtt} is not applicable, \texttt{Select} is applied instead, tagging any cycle node with 1\_1. The two attachments of the path in the piece are not relabelled and therefore, the path in the cycle is not further extendable. \{\texttt{TagPath, TagLast}\} terminates when all edges on the chosen path connecting the two attachments of the path in the piece are tagged with 1\_1. The procedure is correct because only nodes and edges tagged with 0 on the selected path in the cycle are relabelled. \texttt{TagCyclePath} terminates as every rule-schema application reduces the number of nodes and edges tagged with 0 in the graph.

After execution of \texttt{TagCyclePath}, all nodes and edges of the new separating cycle are tagged with 1. As cycles in \texttt{PlanarityTest} are tagged with 0, \texttt{DelTag} and \texttt{Untag(1)} are executed to remove all second tags and all tags distinct from 1. Finally, \texttt{Relabel} relabels the tags of the cycle nodes and edges with 0. Thus, all nodes and edges in the result are untagged except for the ones in the cycle which are tagged with 0.
Example 15 (SepCycle Example Run) Figure 7.22 shows an example run of SepCycle on the cycle computed by CompCycle in Example 13 and the piece tagged by TagPiece in Example 14. The second graph represents an intermediate result after execution of SelectPath where the path in the piece is highlighted and tagged with 1. Note that Backtrack was applied once as one node is tagged with 2. The third graph represents an intermediate result after execution of TagCyclePath. The result of SepCycle contains a separating cycle that is tagged with 0 and the remaining nodes and edges of the graph are untagged. In this example, the cycle forms three new pieces. Note that this is only one possible run. Another run might, for example, choose the path outlined in Figure 7.20.

The procedure SepCycle has been extensively tested on the GP system. This was possible because all the procedures are basically GP macros. A run of the procedure SepCycle on the input graph in Figure 7.22 in backtracking mode resulted in 216 graphs with correct separating cycles. In the set of 216 graphs, all the different paths in the piece were selected at least once and Backtrack was applied several times. We also tested SepCycle on the input graph with two attachments (see outline in Figure 7.21). Each of the 256 resulting graphs contained a separating cycle that was tagged with 0. Both of the paths along the cycle were chosen. The input graph is particularly suitable for testing because it contains a subgraph in which a path can reach a dead end during exploration.

7.3.4 The Procedure CompInter

The procedure CompInter in Figure 7.23a to Figure 7.23d simulates step 4 of the original algorithm in Figure 7.10. The input graph is an undirected graph in which a cycle is tagged with 0 and in which the pieces with respect to the cycle are tagged with distinct integers greater than 1. For each piece number $k$, there is an unconnected node in the input graph with label $k\_0\_0$. The graph does not contain any other nodes or edges.
Figure 7.22: An example run of SepCycle, where $P_1 = \text{SelectPath}$, $P_2 = \text{TagCyclePath}$ and $P_3 = \text{DelTag}$; Untag(1); Relabel
The input graph for \texttt{CompInter} is computed by the procedure \texttt{CompPieces}. \texttt{CompInter} computes the interlacement graph of the connected subgraph of the input graph following the characterisation of interlacing pieces given in [AP61] and in Section 7.1. The characterisation assumes an order on the cycle nodes. This order is imposed on the cycle by \texttt{Select; NumberCycle(1)} using a second tag to number each cycle node. First, \texttt{Select} selects a cycle node by adding a second tag with integer 1 to the node's label. Then, starting with this node, the nodes in the cycle are consecutively numbered by \texttt{NumberCycle}.

The nodes of the interlacement graph are uniquely identified as no other node is tagged with 0\_0. For every two intersecting pieces with tags \( k \) and \( j \), \{\texttt{Inter1, Inter2, Inter3, Inter4, Inter5, Inter6}\}! inserts an edge with label 1 between the nodes tagged with \( k\_0\_0 \) and \( j\_0\_0 \) in the interlacement graph. We need six rule schemata to be able to match all possible connections of a piece with the cycle. The rule-schema condition is satisfied if the pieces with numbers \( k \) and \( j \) interlace and there exists no edge between the two nodes of the interlacement graph. The expression \( q < p \) and \( t < s \) and \( p < t \) checks if a path along the cycle meets the attachments of the two pieces in the order \( q, p, t, s \) (see definition of interlace in Section 7.1). Cycle edges (tagged with 0) cannot be matched in any of the rules as edge tags must match the label of the interlacement nodes and these are always numbered with integers larger than 1.

The procedure \texttt{NumberCycle} terminates because every application of \texttt{Next} reduces the number of nodes tagged with 0 in the graph. The procedure \texttt{CompInter} terminates because \texttt{NumberCycle} terminates and every application of the rule-schema set reduces the number of non-adjacent nodes in the interlacement graph.

The procedure \texttt{CompInter} computes an interlacement graph for the input graph. This is the case as every possible matching is considered by the six rule schemata in \texttt{CompInter} and every rule-schema condition checks the specific ordering of interlacing pieces in the cycle.
CompInter =

\[
\text{Select}(x: \text{int}) = \\
\begin{array}{c}
\text{x}_0 \\
1
\end{array} \implies \\
\begin{array}{c}
\text{x}_0 \\
1
\end{array}
\]

\[\text{Inter1}(x,y,z,k,j,l,s,t,p,q,r,u: \text{int}) =\]

\[
\begin{array}{c}
\text{y}_0 \\
3
\end{array} \quad \begin{array}{c}
r_k \\
6
\end{array} \quad \begin{array}{c}
\text{l}_0 \\
6
\end{array} \quad \begin{array}{c}
t \\
6
\end{array} \quad \begin{array}{c}
\text{z}_0 \\
4
\end{array} \quad \begin{array}{c}
u_j \\
4
\end{array} \quad \begin{array}{c}
x_0 \\
3
\end{array} \quad \begin{array}{c}
p \\
3
\end{array} \quad \begin{array}{c}
x_0 \\
3
\end{array} \quad \begin{array}{c}
u_j \\
4
\end{array} \quad \begin{array}{c}
z_0 \\
4
\end{array}
\]

\[
\begin{array}{c}
k_0 \\
1
\end{array} \quad \begin{array}{c}
j_0 \\
2
\end{array} \implies \\
\begin{array}{c}
k_0 \\
1
\end{array} \quad \begin{array}{c}
j_0 \\
2
\end{array}
\]

where \( q < p \) and \( t < s \) and \( p < t \) and not edge(5, 6) and not edge(6, 5)

\[\text{Inter2}(x,y,z,k,j,l,s,t,p,q,u,a,b,c,d: \text{int}) =\]

\[
\begin{array}{c}
b_k \\
7
\end{array} \quad \begin{array}{c}
c_k \\
9
\end{array} \quad \begin{array}{c}
a_k \\
7
\end{array} \quad \begin{array}{c}
d_k \\
9
\end{array} \quad \begin{array}{c}
y_0 \\
3
\end{array} \quad \begin{array}{c}
l_0 \\
6
\end{array} \quad \begin{array}{c}
t \\
6
\end{array} \quad \begin{array}{c}
x_0 \\
3
\end{array} \quad \begin{array}{c}
u_j \\
4
\end{array} \quad \begin{array}{c}
z_0 \\
4
\end{array}
\]

\[
\begin{array}{c}
k_0 \\
1
\end{array} \quad \begin{array}{c}
j_0 \\
2
\end{array} \implies \\
\begin{array}{c}
k_0 \\
1
\end{array} \quad \begin{array}{c}
j_0 \\
2
\end{array}
\]

where \( q < p \) and \( t < s \) and \( p < t \) and not edge(5, 6) and not edge(6, 5)

**Figure 7.23a:** The procedure CompInter
Inter3(x,y,z,k,j,l,s,t,p,q,u,a,b,c:int) =

where q < p and t < s and p < t and not edge(5, 6) and not edge(6, 5)

Inter4(x,y,z,k,j,l,s,t,p,q,u,a,b,c,d,e,f,g,h:int) =

where q < p and t < s and p < t and not edge(5, 6) and not edge(6, 5)

Figure 7.23b: The procedure CompInter (continued)
Inter5(x,y,z,k,j,l,s,t,p,q,u,a,b,c,d,e,f,g:int) =

\[
\begin{align*}
  \text{where } q < p \text{ and } t < s \text{ and } p < t \text{ and not edge}(5, 6) \text{ and not edge}(6, 5)
\end{align*}
\]

Inter6(x,y,z,k,j,l,s,t,p,q,u,a,b,c,e,f,g:int) =

\[
\begin{align*}
  \text{where } q < p \text{ and } t < s \text{ and } p < t \text{ and not edge}(5, 6) \text{ and not edge}(6, 5)
\end{align*}
\]

Select; NumberCycle(1);

\{Inter1, Inter2, Inter3, Inter4, Inter5, Inter6\}!
\[ \text{NumberCycle}(k: \text{int}) = \]
\[ \left( \begin{array}{c}
\text{Next}(k, x, y, z: \text{int}) = \hfill \\
\begin{array}{c}
\text{x}_0, k \\
1 \\
\hline
\text{y}_0 \\
2 \\
\hline
\text{z}_0 \\
3 \\
\hline
\end{array} \\
\Rightarrow \\
\begin{array}{c}
\text{x}_0, k \\
1 \\
\hline
\text{y}_0 \\
2 \\
\hline
\text{z}_0, k+1 \\
3 \\
\hline
\end{array}
\end{array} \right) \]

try \text{Next}(k) then (\text{NumberCycle}(k + 1)).

**Figure 7.23d: The procedure \text{CompInter} (continued)**

**Example 16 (CompInter Example Run)** 7.24 shows an example run of \text{CompInter} on a graph that is computed by \text{CompPieces} executed on the graph computed by \text{CompCycle} in Example 13. The nodes of the interlacement graph are the unconnected nodes in the input graph which are tagged with \text{0-0}. An execution of \text{Select; NumberCycle}(1) results in the second graph (top right) in Figure 7.24. Note that this is only one possible execution since \text{Select} is non-deterministic. The result in Figure 7.24 shows the interlacement graph that is computed by the iteration of the rule-schema set in the procedure \text{CompInter}. Note that the interlacement graph computed by \text{CompInter} is unique as \{\text{Inter1, Inter2, Inter3, Inter4, Inter5, Inter6}\} is applied as long as there are interlacing pieces \text{p1} and \text{p2} but no edge between the nodes representing \text{p1} and \text{p2} in the interlacement graph and the six rule schemata of the set map all six possible shapes of interlacing pieces in a graph. Thus, for every two interlacing pieces in the graph, there is an edge between the nodes representing these pieces in the interlacement graph and there is no such edge in the interlacement graph if the pieces do not interlace. Therefore, the interlacement graph computed by \text{CompInter} for an input graph \( G \) is always unique. \( \square \)
Figure 7.24: An example run of CompInt, where $P_1 = \text{Select; NumberCycle}(1)$ and $P_2 = \{\text{Inter1, Inter2, Inter3, Inter4, Inter5, Inter6}\}$!
We have tested the procedure CompInter on several variations on the second graph (top right) in Figure 7.24 with the GP system in backtracking mode. Such variations are, for example, gained by a different ordering of the cycle nodes, adding pieces, removing pieces and inserting non-interlacing pieces. In the results of all runs, the interlacement graph was always of the expected shape. On the second graph in Figure 7.24, the GP system produced 24 output graphs, that all correspond to the result in Figure 7.24. Thus, we will assume that the procedure computes an interlacement graph of the given graph.

CompInter cannot be considered as an elegant solution to the problem as it tests all the possible shapes of pieces that might interlace. A more elegant solution would be to insert edges between each two attachments of a piece and label them with the respective piece number. Then only the rule schema Inter1 needs to be applied as long as possible. The problem is, that we have no concept in GP or GP\(^+\) that allows a condition on an edge with a certain label. However, one would need such a condition in this case as otherwise edges between attachments are inserted ad infinitum. Our form of edge predicate cannot be used since, for example, edges parallel to the cycle edges would not be inserted.

### 7.3.5 The Procedure Bipartite

The procedure Bipartite in Figure 7.25 simulates step 5 of the original algorithm in Figure 7.10. The input graph is an undirected graph in which the nodes in the interlacement graph are uniquely identified with the label \(x\_0\_0\) where \(x\) is the tag of the piece the node represents. Nodes in the interlacement graph may be connected by edges labelled with 1. All other edges in the input graph must be tagged. All other nodes are either tagged with an integer greater than 1 or with \(0\_x\) where \(x > 0\) (cycle nodes). A graph of this shape is produced by an execution of CompInter (see for example Figure 7.24).
Bipartite =

\[
\begin{align*}
\text{Delete}(x; \text{int}) &= \\
\begin{array}{c}
x \_0 \_0 \\
1
\end{array} & \quad \Rightarrow \\
\begin{array}{c}
x \\
1
\end{array}
\end{align*}
\]

\[
\begin{align*}
\text{Pick}(x; \text{int}) &= \\
\begin{array}{c}
x \\
1
\end{array} & \quad \Rightarrow \\
\begin{array}{c}
x \_1 \\
1
\end{array}
\end{align*}
\]

\[
\begin{align*}
\text{Propagate}(x, y, k; \text{int}) &= \\
\begin{array}{c}
x \_k \\
1
\end{array} & \quad \Rightarrow \\
\begin{array}{c}
x \_k \\
1
\end{array} & \quad \Rightarrow \\
\begin{array}{c}
y \\
2
\end{array} & \quad \Rightarrow \\
\begin{array}{c}
y \_l-k \\
2
\end{array}
\end{align*}
\]

\[
\begin{align*}
\text{Test}(x, y, k; \text{int}) &= \\
\begin{array}{c}
x \_k \\
1
\end{array} & \quad \Rightarrow \\
\begin{array}{c}
y \_k \\
2
\end{array}
\end{align*}
\]

Delete!; (Pick; Propagate)!; if Test then (fail).

\textbf{Figure 7.25: The procedure Bipartite}
The procedure Bipartite computes a bipartite test on the interlacement graph. First, the tags of all the nodes in the interlacement graph are deleted by Delete. The nodes of the interlacement graph are still uniquely identifiable because all the other nodes in the graph are still tagged. Pick then selects one of the nodes in the interlacement graph by tagging it with 1. Then, Propagate divides the nodes into two disjoint sets (see definition of bipartite in Section 7.1). This is modelled by tagging adjacent nodes in the interlacement graph with 0 and 1, alternatively. As the interlacement graph might not be connected (several pieces might not interlace), we apply (Pick; Propagate!) as long as possible. Thereby, all nodes in connected subgraphs of the interlacement graph are tagged. If Test is applicable, there exist two adjacent nodes that are in the same set and thus, the interlacement graph of the pieces is not bipartite. In this case, the procedure Bipartite finitely fails on the input graph. If Test is not applicable, the execution of Bipartite terminates with a graph.

Bipartite terminates because every application of Delete reduces the number of nodes tagged with 0_0 and every application of Pick and Propagate reduces the number of untagged nodes in the graph. Bipartite is correct because if a node is tagged with 0 then all adjacent nodes are tagged with 1, and vice versa, if the graph was bipartite. Thus, Bipartite decomposes adjacent nodes in the interlacement graph into two disjoint sets where possible.

**Example 17 (Bipartite Example Run)** Figure 7.26 shows an example run of Bipartite on the graph computed by CompInter in Example 16. We use a variation of the convention in Assumption 6 to avoid repeating identical graphs. The interlacement graph (of the frame graph) is the connected graph containing edges labelled with 1. The tags in a possible result of a Bipartite execution represent the two disjoint sets 0 and 1. The node selected by Pick could either have been the one with label 3 or 4. The remaining nodes are tagged with 0 or 1, respectively, by Propagate. The result in Figure 7.26 shows that the interlacement graph is bipartite. As a consequence, the frame graph in Figure 7.26 is planar.
Note that the nodes of the interlacement graph are uniquely identifiable in the graph as they are first tagged with 0–0 and then with tags smaller than 2. □

![Interlacement Graph Diagram]

**Figure 7.26: An example run of Bipartite**

We have tested the procedure Bipartite on the GP system in the backtracking mode. The system produces 192 graphs for an interlacement graph with two connected components, one in the form of the interlacement graph in Figure 7.26 and the other being a complete three node graph. All of these results are correct because the adjacent nodes in the component in Figure 7.26 are all tagged with 1 and 0, alternately. In addition, two of the adjacent nodes in the complete graph with three nodes are tagged with the same number. The latter indicates that the graph is not bipartite. Thus, Bipartite computes the expected results.
7.4 Related Work

Several other planarity-test algorithms have been introduced in the literature. The first known planarity-test algorithm is proposed by Kuratowski in [Kur30]. By Kuratowski’s theorem, an undirected graph is planar if and only if it has no subgraphs homeomorphic to the complete graph of five nodes, or to the complete bipartite graph of six nodes as shown in Figure 7.27. Note that the graph in Figure 7.9 contains a subgraph homeomorphic to the Kuratowski graph with six nodes (shown in Figure 7.27) and is therefore not planar. Although Kuratowski’s solution is elegant, the time complexity of the algorithm is exponential.

![Kuratowski graphs](image)

**Figure 7.27: The Kuratowski graphs**

Therefore, more efficient algorithms testing planarity were proposed in the literature, one of which is the Auslander-Parter algorithm discussed in this chapter. Another well-known approach to planarity testing is introduced by Lempel, Cederbaum and Even in [LEC67]. The algorithm by Lempel, Cederbaum and Even is based on a node-addition method. Starting with a single node in the planar embedding, the algorithm consecutively inserts adjacent nodes and incident edges to the planar embedding of a graph. This continues until the complete graph is embedded. For the algorithm to work correctly, the nodes have to be inserted in a special order. Lempel, Cederbaum and Even use so-called *st*-numbering (see [LEC67]) of the input graph. An implementation of linear run-time is given by Booth and Lueker in [BL76]. A detailed description of the algorithm of [LEC67] is given in
[TS92]. As the algorithm works on specific graph structures and additional data structures, we preferred the more intuitive Auslander-Parter algorithm for an implementation in GP⁺.

An overview of existing planarity-test algorithms is given in [Lie01]. Several theses have been published about planarity testing (see for example [Shi69, Mut94]).

The case study on planarity testing in this chapter is the first large-scale case study in the domain of graph algorithms that has been conducted in a graph transformation language. The case study on planarity testing in this chapter is not intended to give an efficient GP⁺ solution for planarity-testing but to show how large-scale programs can be written clearly using GP⁺. The presented procedures show the main advantages of GP⁺ compared to GP. Most of the presented procedures are short and comprehensible. The procedures that have parameters mostly perform computations in a predefined, deterministic order. Examples are CompPieces, RunRec and NumberCycle. In GP such procedure arguments need to be artificially simulated in the graph structure, for example, by inserting an additional tag or a uniquely identifiable node that is used as a counter. Every rule-schema declaration in a parameterised procedure can be re-used by calling the procedure with different arguments. An example is the procedure Untag that is used to delete the tags of different subgraphs while keeping those that match the value of the actual parameter.
Chapter 8

Conclusions

In this thesis, we introduced the graph programming language GP and its extension GP+. We have shown that GP is a simple, yet practical language that is extendable for different uses and that facilitates formal reasoning. We claim that GP+ provides more programming comfort and better reusability of program parts than GP.

In Section 8.1, we discuss to what extent the three main objectives of GP, namely simplicity, practicality, and suitability for formal reasoning have been achieved, as well as the virtues of the language extension GP+. We discuss future work in Section 8.2.

8.1 Contributions

This thesis makes several contributions which are discussed in the following.

Rule Schemata

The first contribution are conditional rule schemata as a formal basis for graph programming languages.

Conditional rule schemata (see Chapter 3) induce ordinary double-pushout rules and therefore allow computations on labels without using the concept of attributed graphs [HKT02, EPT04], in which graphs are usually infinite.
Performing computations on labels is inevitable when developing a practical graph transformation language. Rule-schema conditions may be expressions on labels or constraints on the (non-)existence of edges in the match. Thus, our concept of conditions combines propositional formulas over term equations [Sch92] with a special (negative) application condition [HHT96] which is typically represented graphically (see also [ERT99]). The advantage of combining both kinds of conditions in a textual form is the clear separation between positive application conditions, represented graphically, and conditions that restrict the possible applications of a rule schema.

The context conditions for the use of variable identifiers in conditional rule schemata (see Section 3.1) admit an efficient implementation of the language as proven in Section 3.3. The concept of conditional rule schemata can be used as a formal basis for other practical graph transformation languages.

**The Graph Programming Language GP**

The graph programming language GP is another contribution of this thesis. GP is simple by the fact that its syntax comprises only eight commands (see Section 4.1), namely call of a set of conditional rule schemata, sequential composition (';'), branching (if-then-else and try-then-else), iteration (while-do and '!'), skip and fail. The semantic core of the language consists of only four commands (see Section 4.2.2), namely call of a set of conditional rule schemata, sequential composition, if-then-else and while-do. The semantics of the remaining constructs is derived from these four. The meaning of GP programs is defined by a complete formal semantics in the form of structural-operational inference rules [Plo04]. This is the first time that structural operational semantics has been used to define a graph transformation language. The language PROGRES [SWZ99] uses control diagrams to define the semantics in an operational way. The advantage of structural operational semantics is that it directly admits an implementation of the language.

We have demonstrated in the examples and case studies of Chapter 5, that GP is practical in that simple and elegant GP programs can be written to
solve various graph-algorithmic problems. However, our definition of finite failure (see Section 4.2.2) for the evaluation of conditions in the branching and iteration commands, might cause efficiency problems when executing a program on a computer system. This is due to the fact that the system has to test all possible executions of a program in a condition for termination with failure because finite only then failure is decidable. When considering an implementation of GP, this is a weak point in the language design that results from the non-determinism in the language. However, backtracking can be disabled in these cases if one proves that if a program in a condition terminates with failure on an input graph, then it terminates with failure for all executions. In this case, it is sufficient to consider only one program execution. Backtracking may also be disabled if one can prove in addition, that a program is deterministic, that is all executions terminate with the same graph. Then, only one program execution (without backtracking) has to be considered.

It is not yet clear if GP is also practical for other domains, as we have only shown its practicality for the domain of graph algorithms. For example, our concept of allowing sequences of labels (see Section 3.1) in graphs (tagging) is suitable for graph algorithms as these do not usually store more than three values in labels. However, there are domains where one needs to store more values in one node, for example, when modelling a class diagram.

We have also shown by the case studies in Chapter 5 that GP facilitates formal reasoning about programs.

It is debatable whether formal reasoning about GP programs is easy or can be automated. Some of the proofs in Chapter 5 are rather complex. However, finding invariants is not a problem specific to GP but rather a general problem of program verification. In showing that properties such as correctness, termination and complexity of GP programs can be proven in a systematic way, we have demonstrated that GP is suitable for verification. With this thesis, we established a sound basis for future work on verification of GP programs.
Extending GP with Procedures: The Language GP+

The extension of GP with procedures, called GP+ (see Chapter 6), is another contribution of this thesis. GP is extended without changing its formal basis. The mechanisms of parameter-passing through procedures (Definition 16 in Section 6.2) and pre-instantiating variable identifiers in rule-schema declarations (Definition 17 in Section 6.2) are real extensions to the formal basis of GP. This shows that we have designed a language that may serve as a core language for more complex graph transformation languages.

The language GP+ is more complex than GP because procedures add a further level of abstraction to the language. However, we kept the procedure concept as simple as possible by not allowing variable declarations or nested procedures and by keeping the parameter-passing concept simple. We claim that GP+ is still simple from a syntactic and semantic point of view. In other programming settings, it has been shown that formal reasoning about programs with procedures is possible (see [Old83]) but this is not demonstrated for GP+ in this thesis.

With the case study on planarity testing in Chapter 7, we have demonstrated that GP+ can be used to formulate solutions to complex problems and that the extension of GP with procedures and parameterised rule-schema calls is very useful. The reusability of rule-schema calls for different purposes, for example, was demonstrated for the rule schema Visit in the procedure DFS in the planarity-test case study (see Section 7.3.1).

Formal reasoning about GP+ programs was not sufficiently demonstrated in this thesis. It is difficult to argue about the GP+ procedures and programs in the planarity-test case study. However, the experienced difficulties came from the complexity of the problem. Several theses have been written about planarity testing (see for example [Shi69, Mut94]). Given this background, constructing a GP+ program that tests a graph for planarity and arguing for its correctness is an achievement even if the program does not follow one of the sophisticated implementations presented in the literature [HT74, BL76]
and hence is not the most efficient one. The main purpose of extending GP to GP+ was not the implementation of an efficient planarity-test algorithm but to gain additional programming comfort and better reuse of program parts. This aim has been achieved, as was demonstrated in the examples in Chapter 6 and Chapter 7.

8.2 Future Work

There are a number of ways in which the current work can be extended. These are discussed in the following.

Modification of Semantics

Recent work on the semantics of GP, introduced in Section 4.2, has led to the conclusion that while-do can be derived from iteration as-long-as possible (!), sequential composition and if-then-else, if '!' is redefined with a (slightly) different meaning. As pointed out below Figure 4.6, an execution of P!, where P is a GP program, may terminate with failure if some executions of P terminate with a graph and others terminate with failure. If we change the semantics of iteration as-long-as-possible so that P! never terminates with failure, the semantics of while-do can be derived from the remaining constructs. As such change of semantics on the one hand gives iteration as-long-as-possible the intended meaning and on the other hand solves the problem of defining while-do in terms of '!', the structural operational semantics of GP should be modified and presented in the modified form in [PS07].

Types

As discussed in Section 4.3.2, other graph transformation languages such as AGG [ERT99] and PROGRES [SWZ99] allow the definition of “graph schemata” or “type graphs”. Type graphs define the types for nodes and edges, including the number of ingoing and outgoing edges, as a means to
control the shape of a graph during transformation. Work along these lines has also been carried out in [BPR04, DP06], towards the verification of graph shapes defined by graph reduction systems. This approach could be used in GP to define shapes of input graphs and to prove statically that a GP program preserves the shape graph throughout the transformation.

**Denotational Semantics**

The structural operational semantics for GP (see Section 4.2.2) focuses on the steps leading to a result. However, it is sometimes worthwhile to focus on the result of a computation, by giving a more abstract, mathematical meaning to programs. Such a view is provided when using denotational semantics [Win93, NN07]. Initial work on defining a denotational semantics for GP has recently been started [PS07]. After defining a denotational semantics for GP, it is clearly desirable to prove the equivalence of the operational and the denotational semantics.

**Case Studies**

In this thesis, we have only presented case studies in the domain of graph algorithms (see Chapter 5 and Chapter 7). Thus, work on case studies in other domains should be conducted to further demonstrate the practicality of GP. In addition, more case studies might reveal weaknesses of the language that have not been uncovered by the case studies in this thesis.

**Formal Reasoning**

In this thesis, we have shown that formal reasoning “by hand” about GP programs is possible (see case studies in Chapter 5). Investigating the possibility of automated program verification is future work. For this purpose, an assertion language should be developed that allows to formulate pre- and post-conditions for programs. Also, a Hoare-style calculus of proof rules for GP constructs is needed. First steps in this direction have been done by Habel, Pennemann and Rensink in [HPR06]. Further work on this topic might
include the development of a methodology for correct program development (see [AO97]).

**Abstraction of Tagging Concept**

The concept of tagging is used for different purposes throughout this thesis (see introduction in Section 3.1). One of the main uses is to mark certain subgraphs of the input graph by tagging. For example, the program Dijkstra in Section 5.3.2 tags nodes to distinguish those in the set of already reduced nodes from those that still need to be considered. Another example is the use of tags to distinguish the different pieces in a graph in the planarity-test case study in Chapter 7. In these cases, tagging is used as a low-level construct to distinguish subgraphs in an input graph for further reference. This, however, means that the programmer is left with the encoding. Future work on GP should investigate a further abstraction of the tagging concept, for example, by graph objects.

**Refinement of Parameter Concept**

Actual parameter lists in GP+ procedure calls can be long even if one only wants to pass one parameter. Consider, for example, a call of the rule schema NotPath in procedure RunRec in Section 7.3. If we are not only interested if a certain piece with tag k is not a path but want to reuse the rule schema to match such a pattern being connected with a cycle node, we would need to call NotPath using, for example, the actual parameter list ( _, _, _, _, _, _, _, 0, _, _, _ ). In future work on GP+, one should consider a more elegant parameter concept. One possibility is to provide two lists of formal parameters for rule schema declarations: one that defines variable identifiers that might be instantiated by a call and another one that defines the variable identifiers that are instantiated by graph matching. Another possibility is to match formal parameters by name. This would significantly shorten the actual parameter list in most cases.
Integrating GP into the .NET Framework

The current graph-transformation languages and systems are stand-alone systems [FNTZ98, ERT99, SWZ99, Kus00]. For a broader use of graph programming languages, an embedding of these languages and in particular GP into frameworks, such as the Microsoft .NET framework, is desirable. A compiler to translate a GP program into the Common Intermediate Language (CIL) of the .NET framework should be written. The CIL code is then executed on the virtual machine provided by .NET. The .NET framework already supports many high-level languages, for example C#, Haskell, Java, Perl, Python and Fortran (for a list of languages supported by .NET see “http://www.startvbdotnet.com/dotnet/languages.aspx”). The platform supports cross-language compatibility, i.e. .NET components can interact with each other irrespective of the languages they are written in. Thus, by providing a GP to CIL compiler, the .NET framework would allow programmers to use GP for solving graph-related problems within the context of a larger problem implemented in another integrated language. First steps in compiling graph transformation languages into the CIL have been done for the GrGen system [BG07] which is based on the single-pushout graph transformation approach [EHK+97].
Bibliography


[DP06] Dodds, M. and Plump, D. Graph transformation in constant time. In *Proc. International Conference on Graph Transforma-


Index

(time-)complexity, 81
deterministic, 44
direct derivation, 26
directed, 22
distance, 80
domain, 21
double-pushout approach with re-
labelling, 23
evironments, 135
frame graph, 161
gests stuck, 63
global, 126, 131
graph morphism, 22
graph-class specification, 81
header, 130
inclusion, 22
Inference rules, 62
injective, 22
inner nodes, 80
interface, 24
interlace, 152
interlacement graph, 152
isomorphism, 22
label alphabet, 21
left-hand side, 24
list of actual parameters, 132

abstract graphs, 23
acyclic, 124
applied occurrence, 135
assignment, 41
attachment, 151
axiom, 62
biconnected, 150
binding occurrence, 135
bipartite, 150
body, 130
case study, 82
category, 33
commands, 52
concrete graphs, 23
conditional rule schema, 44
conditional rule schemata, 31
configuration, 62
connected, 52, 125
constant, 132
context conditions, 34
correct, 82
cycle, 80, 125
cycle of negative length, 80
dangling condition, 26
list of formal parameters, 130
local, 126, 130
loop-free, 67
main command sequence, 131
match, 26
meta-variables, 62
natural, 25
non-terminating, 63
parameterised, 132
parameterised rule-set call, 132
partially correct, 82
partially labelled graph, 21
path, 52, 80
pieces, 151
planar, 149, 153, 154
premorphism, 22
program’s declaration list, 131
pullback, 25
pushout, 25
reachable, 80, 125
right-hand side, 24
rule, 24
rule-schema declaration induced, 140
semantically equivalent, 71
separating, 152
separation point, 150
shortest path, 80
simple, 80
small-step, 61
source, 21
spanning tree, 125
structural operational semantics, 61
stuck, 65
surjective, 22
tagged, 37
tags, 37
target, 21
terminal, 62
terminates, 80
terminates with a graph, 63
terminates with failure, 63
time-complexity, 81
totally correct, 82
totally labelled, 22
transitive, 80
undirected, 67
undirected graph, 52
undirected tree, 125
value, 41
vertex colouring, 116