The Implementation of Graph Programming Systems

Qualifying Dissertation

Christopher Bak

September 21, 2012

Abstract

Implementing a graph programming system is not a straightforward task. The challenge not only lies in converting a graphical graph-based program into low-level machine code, but the code must correctly perform the complex and non-deterministic operation of graph transformation through graph rewriting rules.

GP is a high-level graph programming language. Users of the GP system can write graph programs using a graphical editor by constructing rule schemata and combining them with control constructs. This report addresses the possible ways to refine the prototype implementation of GP. The current implementation uses a stack-based abstract machine tailored for graph-based computation, which is ideal for handling nondeterminism and easily adaptable to changes in the language.

We focus on two main refinements. The first concern is improving GP's subgraph matching algorithm. We examine the literature of several graph programming languages and their strategies for inspiration. The second involves modifying the current implementation to be able to execute rooted graph programs, a type of graph program that can be executed very efficiently. We present current work concerning the theory behind rooted graph transformation, including a complexity analysis of rooted graph matching and some example rooted programs in GP.

Contents

1 Introduction **5**

2 Graph Transformation **7**
2.1 Graphs, Rules and Graph Morphisms 7
2.2 Double Pushouts with Relabelling 8
2.3 Applications of Graph Transformation 12

3 GP **14**
3.1 Conditional Rule Schemata 14
3.2 Application of Rule Schemata 17
3.3 Graph Programs 19
3.4 Operational Semantics 21
3.5 Implementation 24
3.5.1 The Compiler 24
3.5.2 The York Abstract Machine 25
3.6 Summary 27

4 Other Graph Programming Languages **28**
4.1 PROGRES 28
4.1.1 Searchplan Generation 29
4.1.2 Fujaba 31
4.2 GrGEN 32
4.2.1 Searchplan Generation 32
4.3 AGG 35
4.3.1 Graph Matching by Constraint Satisfaction 36
4.4 Generation of Sierpinski Triangles 37
4.5 Overview of Graph Matching Strategies 38

5 Rooted Graph Transformation **41**
5.1 Rooted Graphs and Rooted Rules 42
5.2 A Matching Algorithm for Rooted Rule Schemata 42
5.3 Complexity of Rooted Rule Schemata 46
5.4 Case Study: Colouring Programs 50
5.4.1 Non-rooted 2-colouring 50
5.4.2 Rooted 2-colouring 52
5.4.3 Complexity Comparison 53
5.4.4 General Colouring 54

5.5 Cycle Graphs 55
5.6 Summary 57

6 Research Proposal **58**
6.1 Rewrite the GP Compiler 58
6.2 Design and Implement a Searchplan Generation Algorithm 58
6.3 Add Tools and Features 59
6.4 Further Work on Rooted Graph Programs 60
6.5 Schedule 60

1 Introduction

Graphs are an abstract structure used to model relations between a collection of objects. They have a variety of applications, both within computer science and in various other fields. Some example applications are large-scale networks, model-driven engineering and the manipulation of pointer structures [DP06b]. Because of their importance, graph theory has undergone extensive research for many decades. Manipulating graphs in popular languages such as C and Java is cumbersome and prone to error as they do not provide adequate support or syntax for graphs. The concept of programmed graph grammars has been studied since the late 1970s [Bun79], but it wasn't until the early 1990s that the first programmable graph transformation system came to fruition with PROGRES [NS91; Sch91]. Despite this, high-level programming languages designed specifically to transform graph-based models have remained in relative obscurity.

Today, numerous graph programming languages exist, including The University of York's own GP [MP08b; Plu09; Plu12]. The two primary design principles of GP are abstraction and simplicity. GP programs are written at a very high level of abstraction with the use of a graphical editor, where creating graphs and rules is visual, straightforward and intuitive. One of the most appealing features of graphs is their visual, diagrammatic nature; so why shouldn't a programmer be able to program with graphs visually? Another notable feature of GP is its brevity. The syntax and semantics are small, making it a relatively easy language to both learn and develop. However, one should not presume that GP lacks expressiveness due to its simplicity. On the contrary, its rule schemata and control constructs enable users to write powerful graph programs. GP is based on a computationally complete set of programming constructs for graphs [HP01].

My research concerns the implementation of GP. A prototype implementation of GP exists [MP08b]. However, it needs refining for several reasons. First, the design of a new version of GP [Plu12] has been created since the original implementation, and the implementation must be updated to reflect the changes in the language. Second, an improved algorithm for graph matching could greatly increase GP's speed. Third, an extension to GP involving rooted graphs is currently under design, which will need to be implemented at some point. Finally, GP would be more complete as a graph transformation tool if it supported static analysis for desirable graph program properties such as termination and confluence.

This report addresses these issues with respect to both GP and to other graph programming languages. We first examine the syntax, semantics and implementation of the GP programming language. We then look at three other languages, whose features and implementation differ greatly from those of GP. We focus on how their systems perform subgraph matching, known to be the bottleneck in the efficiency of graph transformation [GJ90], and compare them to GP's own method. This examination could greatly aid in the design of a new procedure for subgraph matching in GP. The report also presents rooted graph programs, a novel modification to standard graph programs that theoretically admits a very efficient implementation.

Chapter 2 introduces the basics of graph transformation, including the double-pushout ap-

proach to rule application. Chapter 3 describes the GP system, including the features of its language, how it enables graph programs to be written, and the underlying machine that performs graph-based computation. Chapter 4 gives an overview of three alternative graph programming languages and their varying approaches to programmed graph transformation, paying particular attention to how they cope with the difficult subgraph matching problem. Chapter 5 describes current work on the theory of rooted graph transformation and an extension of GP that utilises this mechanism. Finally, Chapter 6 proposes a schedule of further research goals.

2 Graph Transformation

In this chapter we introduce the fundamental concepts of graph transformation with respect to the double pushout with relabelling approach to graph transformation [HP02; Ste07]. This admits, in particular, a non-standard definition of graph rules. We present the definitions in this way as GP is based on graph transformations with relabelling.

2.1 Graphs, Rules and Graph Morphisms

Definition 1. A graph G over a label alphabet $\mathcal{L}$ is a system $G = (V_G, E_G, s_G, t_G, l_G, m_G)$ where V_G is the finite set of nodes, E_G is the finite set of edges, $s_G\colon E_G \to V_G$ and $t_G\colon E_G \to V_G$ are total functions that map edges to their source and target nodes respectively, $l_G\colon V_G \to \mathcal{L}$ is the partial node-labelling function and $m_G\colon E_G \to \mathcal{L}$ is the total edge-labelling function. We write $l_G(v) = \bot$ if $l_G(v)$ is undefined. If l_G is a total function then G is said to be totally labelled.

We say a node v is incident to an edge e, or vice versa, if either $v = s(e)$ or $v = t(e)$.

A distinction is sometimes made between node labels and edge labels. We only use one label alphabet here as GP's label alphabet is universal.

Definition 2. For graphs G and H, a graph morphism $g\colon G \to H$ is a pair of functions $g_V\colon V_G \to V_H$ and $g_E\colon E_G \to E_H$ that preserves sources, targets and labels. More precisely, $s_H(g_E(e)) = g_V(s_G(e))$, $t_H(g_E(e)) = g_V(t_G(e))$ and $l_H(g_V(v)) = l_G(v)$ for all edges $e \in E_G$ and $v \in V_G$ with $l_G(v) \neq \bot$. A graph morphism g is injective (surjective) if g_V and g_E are injective (surjective). A bijective graph morphism is called a graph isomorphism. A graph morphism g is an inclusion if $g(x) = x$ for all nodes and edges $x \in G$. Given two graph morphisms $f\colon G \to H$ and $g\colon H \to J$, the composition $g \circ f\colon G \to J = (g_V \circ f_V, g_E \circ f_E)$, where $\circ$ is the standard function composition operator.

The graph morphism is a formal way of describing structural relationships between graphs. As we shall see, morphisms are required to formally define graph transformation rules and double pushouts. As this report is strictly within the field of graph transformation, we refer to graph morphisms as just morphisms. In addition, we sometimes refer to morphisms by their domain and codomain. For example, the definition of morphism composition could have been written as $(H \to J) \circ (G \to H) = G \to H \to J$. We include the intermediate graph in the composition for clarity.

Definition 3. A rule $r = (L \leftarrow K \to R)$ is a pair of inclusions $K \to L$ and $K \to R$ where L and R are totally labelled graphs. L and R are referred to as the left-hand side (LHS) and right-hand side (RHS) respectively. K is called the interface.

K is not required to be totally labelled. Unlabelled nodes in the interface are precisely the nodes that are to be relabelled. This is described in detail in the next section.

Rules are the standard way to manipulate graphs. Given a rule r and a graph G (commonly referred to as the host graph), we would like to replace an occurrence of L in G with a copy of R. However, this replacement is not straightforward. Finding an occurrence of L in G is an instance of the subgraph isomorphism problem, known to be NP-complete [GJ90]. The reason for its complexity is that in the worst case, namely if no subgraph of G isomorphic to L exists, then all possible combinations will have to be considered during search. The problem is most commonly met by finding a morphism $g\colon L \to G$, called the match. We shall see that there exists a variety of ways to compute a match.

Another issue with this form of graph manipulation is that a node in G may be deleted while one of its incident edges remains. The edge will be left without either a source or a target. Such an edge is not permitted in the definition of a graph. One way to circumvent this problem is to guarantee that it will never happen. This the method used by the double pushout approach. It is achieved by forbidding morphisms that violate the dangling condition, defined below.

Definition 4. Given a rule $r = (L \leftarrow K \to R)$ and an injective morphism $g\colon L \to G$, the dangling condition states that no edge in $G - g(L)$ is incident to any node in $g(L - K)$.

The items in $g(L - K)$ are those removed from G in the first step of rule application, while the edges in $G - g(L)$ are the edges that remain in G after the first step of rule application. Thus the dangling condition states that no preserved edge can be incident to a node that is removed.

To conclude, we define some properties of graphs that are used in later chapters.

Definition 5. A path is a sequence of nodes $v_1, \ldots, v_n$ such that $\forall\ i \in \{1, \ldots, n-1\}, \exists$ edges $e_1, \ldots, e_{n-1}$ such that e_k is incident to v_k and v_{k+1}. v_1 is called the start node of the path. The edge direction is not considered. A node w is reachable from a node v if there exists a path containing the nodes v and w. An edge e is reachable from a node v if there exists a path from v containing e. A graph is connected if every node is reachable from every other node.

2.2 Double Pushouts with Relabelling

Rules were defined in the previous section. However, a formal description of how to apply a rule to a graph to obtain a new graph has yet to be described. The double pushout is a widely used algebraic framework for applying rules to graphs [Cor+97]. Habel and Plump [HP02], and later Steinert [Ste07], modified the framework to allow straightforward relabelling of nodes. This is required in GP, or any graph programming language, since being able to relabel nodes is a key component of useful graph transformation. Relabelling nodes would be challenging in the standard definition of a double pushout, where all graphs are totally labelled, as a node would have to be removed and replaced by a node with a different label, a method which could violate the dangling condition. This is not an issue for edges as they can be arbitrarily deleted and reinserted.

Definition 6. Given morphisms $A \to B$ and $A \to C$, a graph D with morphisms $B \to D$ and $C \to D$ is a pushout if the following conditions hold:

(i) *Commutativity:* $A \to B \to D = A \to C \to D$

(ii) *Universal Property:* For all pairs of morphisms $(B \to D', C \to D')$ such that $A \to B \to D' = A \to C \to D'$, there exists a unique morphism $D \to D'$ such that $B \to D' = B \to D \to D'$ and $C \to D' = C \to D \to D'$.

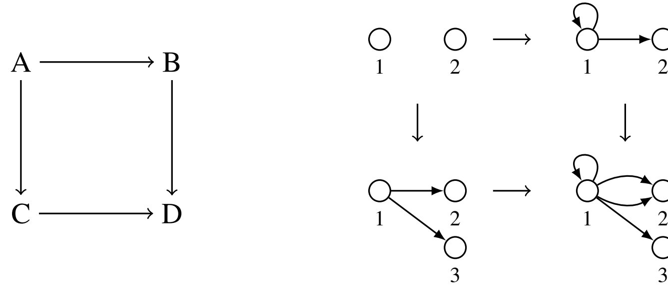


Figure 2.1: A pushout diagram and an example pushout.

The pushout is a formal way of gluing two graphs with respect to a common subgraph. The graph D is the "union" of B and D, where the items that also occur in A are merged. Note that items in A must be in both B and C for commutativity to hold. By the universal property, D is unique up to isomorphism. An abstract pushout diagram and a concrete example of a pushout are given in Figure 2.1. The numbers below the nodes are the node identifiers which are displayed to show how the morphisms map nodes. Node labels, edge labels and edge identifiers are omitted for clarity.

Definition 7. Given morphisms $B \to D$ and $C \to D$, a graph A with morphisms $A \to B$ and $A \to C$ is a pullback if:

(i) *Commutativity:* $A \to B \to D = A \to C \to D$

(ii) *Universal Property:* For all pairs of morphisms $(A' \to B, A' \to C)$ such that $A' \to B \to D = A' \to C \to D$, there exists a unique morphism $A' \to A$ such that $A' \to B = A' \to A \to B$ and $A' \to C = A' \to A \to C$.

A pushout that is also a pullback is called a natural pushout. Pushouts are sufficient for graph transformation without labels, but the introduction of partially labelled graphs can cause ambiguity. Consider the two pushout diagrams in Figure 2.2.

In these diagrams, the node outside the square is an instance of the graph A' in the definition of the universal property. The diagram on the left is not a pullback: commutativity is satisfied but the universal property is not. The morphisms b' and c' satisfy $d \circ b' = d' \circ c'$, but there is no unique morphism that makes the triangles commute. The only possible mapping is the mapping m of a single node, but m is not a graph morphism since it does not preserve labels. On the other hand, the right-hand diagram is a pullback. In this case, even though m is not a morphism, neither is c' as the image of c' is unlabelled. In fact, there are no morphisms $A' \to C$, where C is the lower left graph of the square. It follows that the universal property is trivially satisfied.

This example is an illustration of a general result Habel and Plump prove in [HP02] which defines a characterisation of natural pushouts.

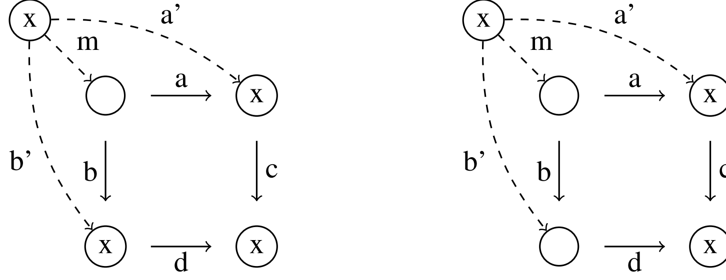


Figure 2.2: A non-natural pushout and a natural pushout.

Lemma 1. *Given two graph morphisms $f : A \to B$ and $g : B \to C$ such that f is injective, the pushout depicted by the left-hand diagram of Figure 2.1 is natural if and only if for all $z \in A, l_A(z) = \bot$ implies $l_B(f(z)) = \bot$ or $l_C(g(z)) = \bot$.*

The lemma states that a pushout is a natural pushout if and only if all unlabelled items in A have an unlabelled image in at least one of B and C. Natural pushouts are required for constructing unique double-pushouts with relabelling. If the pullback condition is not enforced, there may be more than one non-isomorphic graph produced from a particular rule application.

Given graphs G and H, a rule $r = (L \leftarrow K \to R)$, and an injective match, g: $L \to G$, a direct derivation from G to H is a pair of natural pushouts, or a double pushout, depicted in Figure 2.3. If such a derivation exists, we say $G \Rightarrow_{r,g} H$, or more commonly, $G \Rightarrow_r H$. It has been proven that $G \Rightarrow_{r,g} H$ if and only if g satisfies the dangling condition [HP02]. It follows from the definition of pushout that D and H can be constructed uniquely up to isomorphism [Plu09]:

1. To obtain D, remove all nodes and edges in $g(L - K)$ from G. For all $v \in V_K$ with $l_K(v) = \bot$, define $l_D(g_V(v)) = \bot$.
2. Add all nodes and edges, with their labels, from $R - K$ to D. For $e \in E_R - E_K$, $s_H(e) = s_R(e)$ if $s_R(E) \in V_R - V_K$, otherwise $s_H(e) = g_V(s_R(e))$. Targets are defined analogously.
3. For all $v \in V_K$ with $l_K(v) = \bot$, define $l_H(g_V(v)) = l_R(v)$. The resulting graph is H.

In this construction, the interface K represents all nodes and edges that are preserved by the rule; the items in L that are not in K are removed from G, and the items in R that are not in K are added to G. In addition, any unlabelled nodes in the interface are nodes to be relabelled. Their original labels are removed in the first stage of the rule application, and the new labels from R are added during the third stage.

Figure 2.4 depicts a complete double-pushout rule application according to the construction outlined above. Numbers below nodes are their identifiers, numbers inside nodes are their labels, and characters next to edges are edge identifiers. Edge labels are not included for clarity. The rule will match a node with the label 3 and a node with the label 5, where the latter has a looped edge and there is an additional edge from the 3-labelled node to the 5-labelled node. The rule

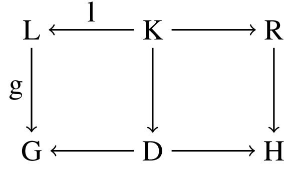


Figure 2.3: A double pushout illustrating the application of a rule $r = (L \leftarrow K \rightarrow R)$ with a match g. Both squares are pushouts.

deletes the looping edge, creates a new loop on the source of the connecting edge, and replaces the label 3 with the label 4. L is matched to G by the match g, where $g_V = (1 \mapsto 1, 2 \mapsto 2)$ and $g_E = (a \mapsto a, c \mapsto c)$. This match does not violate the dangling condition since there are no nodes in $g(L - K)$. Hence we can apply the rule:

1. $g(L - K) = \{c\}$. This edge is removed from G to give D. Since neither node in K has a label, the corresponding nodes in D also have no label.
2. $R - K = \{e\}$. This edge is added to D. Its source is defined to be the image of its source in R: $g_V(s_R(e)) = g_V(1) = 1$. Its target is defined analogously: $g_V(t_R(e)) = g_V(1) = 1$.
3. Nodes 1 and 2 are unlabelled in K, so they are assigned the corresponding labels in R to obtain the resulting graph H.

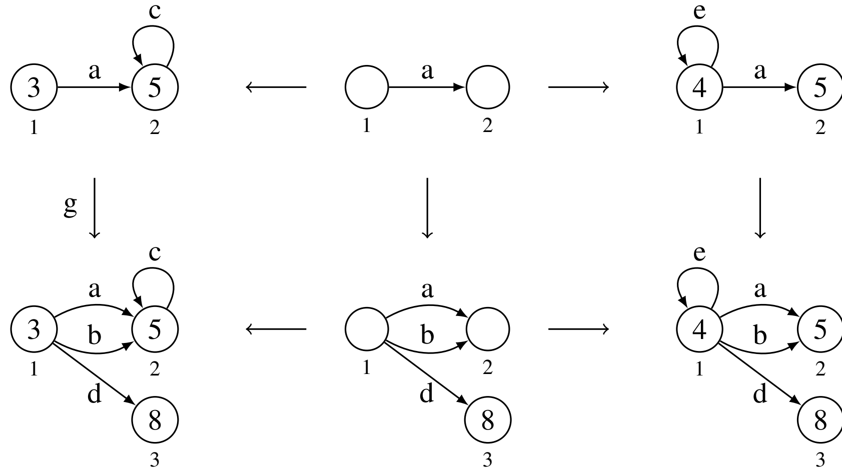


Figure 2.4: A rule application in the double pushout with relabelling construction.

Given a set of rules $\mathcal{R}$ and two graphs G and H, we say G directly derives H by $\mathcal{R}$ if $G \Rightarrow_r H$ for some $r \in \mathcal{R}$. Direct derivations can also be applied in sequence. G derives H, or $G \Rightarrow^*_{\mathcal{R}} H$, if either $G \cong H$ or $G \Rightarrow_{\mathcal{R}} G_1 \Rightarrow_{\mathcal{R}} \ldots \Rightarrow_{\mathcal{R}} H$.

Double pushouts are not only useful for formalising the application of graph transformation rules. The framework also allows us to formally specify conditions on pushouts that must hold

for certain nice properties to arise. For example, two distinct direct derivations $G \Rightarrow_{r_1,g_1} H_1$ and $G \Rightarrow_{r_2,g_2} H_2$ are parallel independent if $g_1(L_1) \cap g_2(L_2) \subseteq g_1(l_1(K_1)) \cap g_2(l_2(K_2))$. This condition states that the items in G needed by both direct derivations, which are the items in the intersection of the occurrences of L_1 and L_2 in G, are preserved by the other direct derivation. That is to say, they must be in the common occurrences of the interfaces K_1 and K_2 in G. If this condition holds, then there exist direct derivations $H_1 \Rightarrow_{r_2,g_2'} J$ and $H_2 \Rightarrow_{r_1,g_1'} J$ such that $(G \Rightarrow_{r_1,g_1} H_1 \Rightarrow_{r_2,g_2'} J) = (G \Rightarrow_{r_2,g_2'} H_2 \Rightarrow_{r_1,g_1'} J)$. Hence, the rules can be applied to the same graph in either order without changing the resulting graph. A more detailed look into double pushouts and properties of direct derivations can be found in [Ehr79; Cor+97].

Another common approach to rule application is the single-pushout approach. As the name suggests, direct derivations only consist of a single pushout, thus there is no interface graph. Instead, a partial morphism $h\colon L \to R$ specifies which nodes and edges are to be preserved, added and deleted. Items not in the preimage of h are removed, while items not in the image of h are created. All other items are kept. Furthermore, a match need not be injective, nor does it need to satisfy the dangling condition. This means that a node or edge can be matched twice, or a dangling edge can be created. These problems are dealt with by deleting the concerned items, even if the rule does not explicitly state such a deletion. Consequently, single pushout direct derivations are more flexible than their double pushout counterparts, as there always exists a direct derivation for a given match. This is not necessarily true for a double pushout, as the match may, for instance, violate the dangling condition. On the other hand, all direct derivations using the double pushout approach are invertible because of their restrictions. For a more thorough description of single pushouts and a comparison to double pushouts, we refer the reader to [Ehr+97].

In this chapter we have introduced the fundamental theoretical background of graph transformation. The next chapter showcases the theory being put into practice with a thorough breakdown of the graph programming language GP.

2.3 Applications of Graph Transformation

Graphs provide a simple and abstract way to view a set of objects and how they relate to one another. Graph transformation is an old and well-researched branch of graph theory. Consequently, graphs have become widely used in computer science for a variety of purposes.

One of the more prevalent applications of graph transformation is in model-driven engineering. This is a method used to facilitate the specification, analysis and verification of complex software systems by depicting them as abstract models that neatly capture their structure and behaviour. Such systems constantly undergo modification due to various factors such as a change in software requirements or the introduction of new technology. Model transformation is used both to formalise system evolution and to convert from high-level system specifications to low-level models that can be compiled to code. Thus, models are treated as first-class citizens, the same way that graphs are used in graph transformation systems.

It follows that diagrammatic model specifications can be represented as graphs, and their evolution is naturally modelled by graph transformation rules. There are a number of tools available for model-based computation with graphs [RSV04; Gru+05]. Fujaba [NNZ00], a language we

discuss later, is based on Unified Modelling Language (UML), a standard general purpose modelling language.

Another important application is using graphs for the verification of pointer structures in languages such as C. Programming with pointers is unsafe as imperative languages are unable to prevent pointer-related errors, such as the creation or deletion of dangling pointers. Recent work at The University of York has led to the creation of an extension of the C programming language that allows programmers to specify the shape of pointer structures through graph reduction systems [BPR04; DP06a].

There are a number of other applications. Any type of network can be visualised as a graph. This includes maps, databases, pipelines and the physical wiring configuration of an office. Well-known graph algorithms such as shortest path, minimum spanning tree and maximum flow are used to solve many of the problems faced by designers or engineers. Graph theory is also used in the design of circuit board layouts to, for example, construct a cost-effective configuration without overlapping components which may cause short circuits.

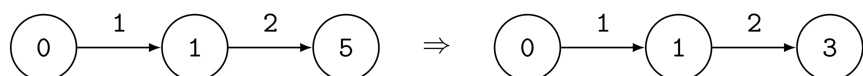
3 GP

Parts of this chapter are based on [Plu12].

In 2001, Habel and Plump showed that nondeterministic application of a rule from a set of double-pushout rules, labelled over a finite label alphabet, is computationally complete using only the control constructs of sequential composition and as-long-as-possible iteration [HP01]. This motivated the design of a small, visual and high-level graph programming language [PS04]. Since then, a complete syntax and semantics has been developed. This chapter provides a comprehensive description of GP and its features, paying particular attention to language changes introduced in a recent revision of GP, called GP2 [Plu12]. We refer to the previous version of GP as GP1.

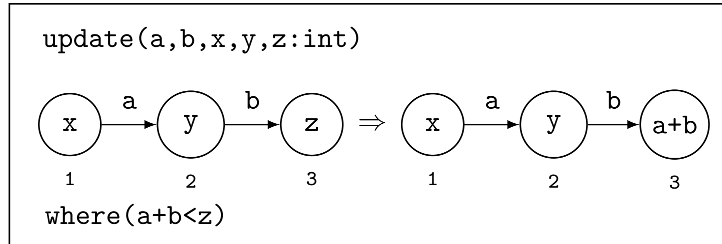
3.1 Conditional Rule Schemata

According to our definition, rules only contain fixed labels. They can modify graphs structurally and change the labels of nodes and edges. This is useful, but not sufficient for a graph programming language. For example, when computing the shortest path between two nodes, we might want to perform the following transformation:



This rule could be used in a shortest path algorithm where a node label is the distance of that node from the left-hand node, and the edge labels represent the distance between a pair of nodes. The LHS models a possible suboptimal configuration. The distance of the right-hand node is 5, yet the edges connecting it to the left-hand node have a total distance of 3. The node label can be updated to record this improved distance, which is precisely what the rule does.

To construct a complete graph program to simulate this algorithm, one would have to write such a rule for every possible combination of integers since the program will be executed on different input graphs with different node and edge labels. This is clearly impossible as there would be an infinite number of rules to write. GP's rule schemata provide an easy way to overcome this problem: nodes and edges can be labelled with variables and expressions. Rules can also have a condition, which forbids the application of the rule if the condition evaluates to false under a particular variable assignment. With these features, the infinite set of rules can be expressed as one single conditional rule schema:



This schema neatly captures the general procedure. The condition ensures that the label of the right-hand node is updated only if the sum of the edges is strictly less than its current label. Since the labels are integer variables, the rule can be applied to any integer-labelled graph.

Not only are conditional rule schemata extremely powerful, they are abstract and intuitive. A GP programmer can easily create such a schema with only a basic level of programming knowledge, and the graphical nature of the schemata makes their purpose easy to understand at a glance.

The abstract syntax of rule schema labels is given in Figure 3.1. Grammars are in Extended Backus-Naur Form. Syntax introduced in GP2 is bolded.

Integer	::=	Digit {Digit} | IVariable | '–' Integer | Integer ArithOp Integer | **(indeg | outdeg) '(' Node ')'**
ArithOp	::=	'+' | '-' | '∗' | '/'
String	::=	' " ' {Char} ' " ' | SVariable | **String '.' String**
Atom	::=	Integer | String | **AVariable**
List	::=	`empty` | **Atom** | **LVariable** | **List ':' List**
Label	::=	**List Mark**
Mark	::=	`true` | `false`

Figure 3.1: Abstract syntax of rule schema labels

GP1 only supported two variable types: `int` and `string`, although they could be sequenced arbitrarily. GP2 introduces two new variable types. The type `atom` is the union of integers and strings. The type `list` represents a possibly empty colon-separated sequence of atoms. The empty list is signified by the keyword `empty`, and it is displayed graphically as a blank label. This presents no ambiguity since both sides of a rule must be totally labelled, hence a blank label can only have one meaning. Strings can be concatenated in GP2 with the dot operator. The new operators `indeg` and `outdeg` take a node identifier and return its indegree and outdegree respectively. Finally, nodes and edges can be marked. This is represented by a boolean flag in the grammar; however, in rule schemata and host graphs, marked nodes are shaded and marked edges are dashed. Graphically distinguishing particular nodes and edges is more appealing than encoding "marks" as integer tags on node labels, a technique frequently used in GP1 programs.

In the grammar, IVariable, SVariable, AVariable, LVariable are the sets of variables of type `int`, `string`, `atom` and `list` that occur in the rule schema. Although these types overlap, variables can only be declared with one type, hence these variable sets are disjoint. Node is the set of node identifiers in the schema.

The list type allows the sequencing of GP1 to be replicated with a single type. There is also a type hierarchy: List $\supseteq$ Atom, Atom $\supseteq$ String and Atom $\supseteq$ Integer. This means that atomic, integer and string variables are syntactically equivalent to a list of length one, hence such variables can be sequenced with list variables. This enables the construction of powerful expressions and conditions.

One minor drawback to the new GP label syntax is that the grammar no longer encodes the restrictions on the left-hand graph necessary to prevent ambiguous matching. Instead, we define a simple expression and insist that expressions in the left-hand side of a rule schema must be simple.

Definition 8. An expression $e \in$ List is simple if

(i) e contains no arithmetic operators.

(ii) e contains at most one occurrence of a list variable.

(iii) each occurrence of a string expression in e contains at most one occurrence of a string variable.

As an illustration, consider the expression $s.t, s, t \in$ SVariable. This is not a simple expression, as it is a string expression with two string variables. When matched with the string constant “foo”, there are two possible assignments, excluding those involving the empty string. Either $s =$ “f”and $t =$ “oo”, or $s =$ “fo”and $t =$ “o”. This is a problem because if there is an expression on the right-hand side of the schema involving s or t, then the graph produced by the rule will no longer be unique. We want to preserve the uniqueness given by the double pushout framework, so non-simple expressions in the LHS of rule schemata are forbidden.

The abstract syntax of rule schema conditions is given in Figure 3.2 Predicates introduced in GP2 are bolded.

Condition	::=	**Type ‘(’ List ‘)’** | **List (‘=’ | ‘!=’) List** |
		Integer RelOp Integer |
		`edge` ‘(’ Node ‘,’ Node **[‘,’ List] ‘)’** |
		`not` Condition | Condition (`and`|`or`) Condition
Type	::=	`int` | `string` | `atom`
RelOp	::=	‘>’ | ‘>=’ | ‘<’ | ‘<=’

Figure 3.2: Abstract syntax of rule schemata conditions

The type predicates `int`, `string` and `atom` are new to GP. They check if their List argument is of that type. Note that, due to the hierarchy in the grammar, the non-terminal List can generate variables belonging to any of the four variable types. As a consequence, variables of any type can be compared for equality or disequality. The `edge` predicate, which checks for the existence of an edge between two nodes, has a new optional third argument, a List. This argument is used to test the label of the edge connecting the two nodes.

We conclude the section by formally defining a conditional rule schema.

Definition 9. A rule schema $(L \leftarrow K \rightarrow R)$ is a rule such that L and R are graphs over Label, K has no labelled nodes, all expressions in L are simple, and all variables in R also occur in L.

A conditional rule schema $(L \leftarrow K \rightarrow R, c)$ is a rule schema and a condition $c \in Condition$ such that all variables in c also occur in L.

3.2 Application of Rule Schemata

To describe the procedure of applying a rule schema, we require the following definition:

Definition 10. Let G and H be graphs. A premorphism g: $G \rightarrow H$ is a pair of functions that meets the requirements of Defintion 2 except for label preserving.

Conditional rule schemata differ quite substantially from regular rules. Standard rules do not contain variables or conditions, so additional mechanisms are required to construct a match and a double-pushout. There are four stages to the application of a rule schema L to a graph G . First, find a premorphism g: $L \rightarrow G$. Second, check if there exists an assignment α of variables in L to values such that g is label-preserving with respect to α. Third, check whether the condition holds under α. Finally, if a valid match and assignment has been found, apply the rule to G as described in the construction of a double pushout in Chapter 2. This involves using the assignment to evaluate all expressions in the rule schema and relabelling G accordingly.

As seen in the previous section, labels in the rule schema are taken from the syntactic category Label. Input graphs are labelled with values from the semantic domain $\mathcal{L} = (\mathbb{Z} \cup \text{Char}^*)^* \times \mathbb{B}$, where Char is a fixed set of characters. These are colon-separated sequences of integers and strings, representing the list expression, paired with a boolean value, which represents the boolean flag Mark.

Definition 11. An assignment is a family of mappings $\alpha = (\alpha_X)_{X \in \{I,S,A,L\}}$ where:

- α_I : IVariable $\rightarrow \mathbb{Z}$
- α_S : SVariable $\rightarrow$ Char*
- α_A : AVariable $\rightarrow \mathbb{Z} \cup$ Char*
- α_L : LVariable $\rightarrow (\mathbb{Z} \cup \text{Char}^*)^*$

Given a premorphism g: $L \rightarrow G$, an assignment α, and a label $l = em$ with $e \in List$ and $m \in \{\texttt{true,false}\}$, the value $l^{g,\alpha} \in \mathcal{L}$ is the pair $(e^{g,\alpha}, m)$. Hence, marked nodes (edges) cannot match unmarked nodes (edges) and vice versa.

$e^{g,\alpha}$ is the value of the List e when evaluated with respect to the premorphism g and assignment α. In addition, we define $c^{g,\alpha} \in \mathbb{B}$, which is the value of the rule schema condition when evaluated with respect to the premorphism g and assignment α. Both are defined inductively in Figure 3.3.

In the table, $e_1, e_2 \in$ List, $x, y \in$ Integer, $m, n \in$ Node, $s_1, s_2 \in$ String, and $c_1, c_2 \in$ Condition. The symbol $\oplus_{\mathbb{Z}}$ signifies the integer operation represented by $\oplus$. Similarly, $\bowtie_{\mathbb{Z}}$ is the integer relation represented by $\bowtie$. Also note that in the second and third rows, the integer (string) represented by a sequence of digits (characters) is unique.

expression	$e^{g,\alpha}$	condition	$c^{g,\alpha} = \texttt{true} \Leftrightarrow$
`empty`	empty sequence	$\texttt{int}(e_1)$	$e_1^{g,\alpha} \in \mathbb{Z}$
sequence of digits	integer represented by e	$\texttt{string}(e_1)$	$e_1^{g,\alpha} \in \text{Char}^*$
sequence of characters in quotes	string represented by e	$\texttt{atom}(e_1)$	$e_1^{g,\alpha} \in \mathbb{Z} \cup \text{Char}^*$
$-x$	$-x^{g,\alpha}$	$e_1 = e_2$	$e_1^{g,\alpha} = e_2^{g,\alpha}$
$x \oplus y$	$x^{g,\alpha} \oplus_{\mathbb{Z}} y^{g,\alpha}$	$e_1 \neq e_2$	$e_1^{g,\alpha} \neq e_2^{g,\alpha}$
variable	$\alpha(e)$	$x \bowtie y$	$x^{g,\alpha} \bowtie_{\mathbb{Z}} y^{g,\alpha}$
$\texttt{indeg}(n)$	indegree of $g_V(n)$	$\texttt{edge}(m, n)$	there exists an edge in G from $g_V(m)$ to $g_V(n)$
$\texttt{outdeg}(n)$	outdegree of $g_V(n)$	$\texttt{edge}(m, n, e_1)$	there exists an edge in G from $g_V(m)$ to $g_V(n)$ with label whose list component is $e_1^{g,\alpha}$
$s_1.s_2$	concatenation of $s_1^{g,\alpha}$ and $s_2^{g,\alpha}$	$\texttt{not}\ c_1$	$c_1^{g,\alpha} = \texttt{false}$
$e_1 : e_2$	concatenation of $e_1^{g,\alpha}$ and $e_2^{g,\alpha}$	$c_1\ \texttt{and}\ c_2$	$c_1^{g,\alpha} = \texttt{true} = c_2^{g,\alpha}$
		$c_1\ \texttt{or}\ c_2$	$c_1^{g,\alpha} = \texttt{true}$ or $c_2^{g,\alpha} = \texttt{true}$

Figure 3.3: Definitions of $e^{g,\alpha}$ and $c^{g,\alpha}$

Definition 12. $r^{g,\alpha} = (L^{g,\alpha} \leftarrow K \rightarrow R^{g,\alpha})$ is the instance of r with respect to g and α, where $L^{g,\alpha}$ and $R^{g,\alpha}$ are the graphs L and R after the replacement of their labels l with $l^{g,\alpha}$. $r^{g,\alpha}$ is a rule over $\mathcal{L}$.

Definition 13. Given a conditional rule schema $r = (L \leftarrow K \rightarrow R, c)$, and graphs G, H over $\mathcal{L}$, G directly derives r, denoted $G \Rightarrow_r H$, if there exists a premorphism g: $L \rightarrow G$ and an assignment α such that

(i) g is a morphism $L^{g,\alpha} \rightarrow G$.

(ii) $c^{g,\alpha} = \texttt{true}$.

(iii) $G \Rightarrow_{r^{g,\alpha},g} H$.

We do not introduce new notation when defining direct derivations over conditional rule schemata as opposed to traditional rules. When the notation is used, it will be clear from the context which type of direct derivation is being described.

Even with the extension of traditional rules to conditional rule schemata, uniqueness of double pushout direct derivations is preserved. For any conditional rule schema and premorphism g,

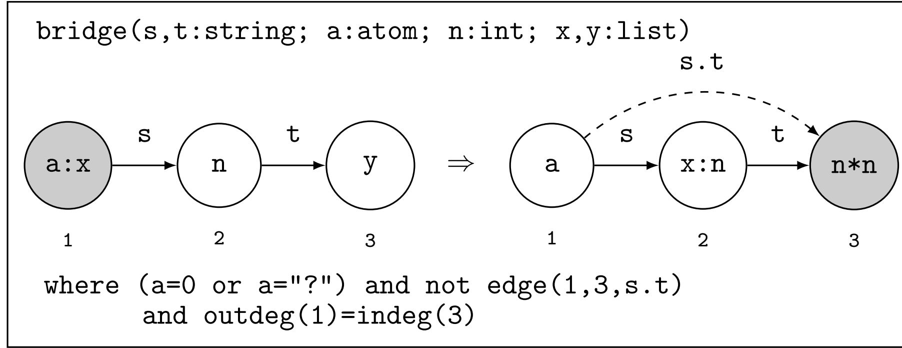


Figure 3.4: Declaration of a conditional rule schemata

there is at most one assignment that makes g a morphism since only simple expressions are allowed in L. Uniqueness (up to isomorphism) of the transformed graph follows from this and the uniqueness of the graph H in the double pushout diagram of Figure 2.3.

We demonstrate the application of a conditional rule schema to a graph by an illustrative example, taken from [Plu12]. The schema in question is shown in Figure 3.4.

The rule is declared at the top with a rule identifier followed by a list of variable declarations. Multiple variables of the same type can be declared simultaneously. Declarations of variables of different types must be separated by a semicolon. Observe that node 1 on the LHS and node 3 on the RHS are shaded. This means that Mark is set to true in the label of those nodes. The same applies for the edge linking nodes 1 and 3 on the RHS. GP2 allows more flexible conditions than GP1. For example, as in the condition, variables of type `atom` can be compared to both integers and strings.

Figure 3.5 shows the rule schema `bridge` being applied to a host graph (the lower left graph). The upper portion represents the instantion of `bridge` with respect to the premorphism $g_V\colon (1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3)$ (g_E defined in the obvious way) and the assignment α: $(a \mapsto 0, x \mapsto 1 : 2, n \mapsto 3, y \mapsto 4, s \mapsto \text{“}o\text{”}, t \mapsto \text{“}k\text{”})$. After variable assignment, g is label-preserving and hence a morphism. In addition, the condition holds with respect to g and α. Therefore the rule can be applied to the host graph. The graph after successful rule application is the lower right graph.

3.3 Graph Programs

GP programs are composed by defining a set of conditional rule schemata, and organising the schemata using a small set of control constructs. It has already been established that nondeterministically applying rules from a set, sequentially composing rules and iterating subprograms is critical for computational completeness. However, GP does offer more control constructs for usability. The abstract syntax of GP Programs is shown in Figure 3.6.

A program is a list of declarations, which can either be RuleDecl: a rule schema declaration as described in the previous section, MacroDecl: a macro declaration, or MainDecl: the keyword `main`, signifying the start of program execution, followed by a command sequence (ComSeq). It

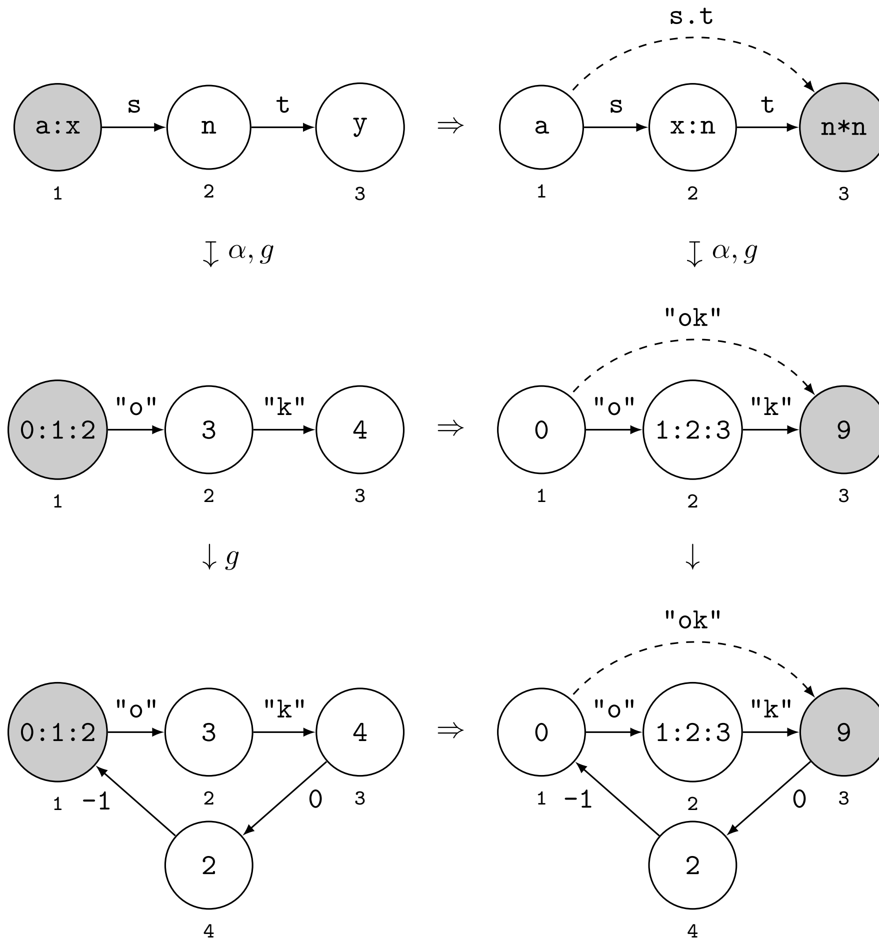


Figure 3.5: An application of the schema `bridge` [Plu12]

is required that the `main` keyword appears exactly once in a GP program. A command sequence is a sequential composition of commands:

- RuleSetCall: either a single rule call or a nondeterministic choice from a set of rules. The category RuleId represents the set of rule identifiers.
- `if-then-else`: a conditional branching construct. Note that the test is also a command sequence. We defer an explanation until the next section.
- `try-then-else`: a conditional branching construct, as above, but with different semantics.
- ComSeq!: a looping construct. The command sequence is applied as many times as possible.
- ComSeq `or` ComSeq: nondeterministic choice of two command sequences.

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

Figure 3.6: Abstract syntax of GP programs

- `skip`: do nothing. Equivalent to an application of the empty rule, which has the empty graph on both its left and right-hand side.
- `fail`: manually trigger termination in a failure state. Equivalent to an application of the empty rule set {}.

The `or`, `skip` and `fail` commands can be simulated by the other commands. Thus, GP at its core offers the `if` and `try` branching construct in addition to the control constructs required for computational completeness. A small example of a GP program is given below. Rule schemata are omitted.

```
main = start; middle; end
middle = {foo; bar}!
```

Program execution starts at the command sequence following `main`. The rule `start` is applied, followed by an execution of the macro `middle`. Macros offer no extra functionality; they are only used to write more readable programs. The program would operate equivalently if the macro identifier `middle` were replaced by the command sequence it represents (`{foo; bar}!`). The macro `middle` nondeterministically applies either `foo` or `bar` as long as possible: until either `foo` or `bar` cannot be applied to the working graph. If this happens, the loop terminates, and the rule `end` is applied once. The program then terminates since there are no commands remaining. More involved examples of GP programs are given in Chapter 5.

3.4 Operational Semantics

GP2 has a formal semantics, presented here in the style of Plotkin's structural operational semantics [Plo04]. The inference rules, shown in Figure 3.7 and Figure 3.8, inductively define a small-step transition relation $\rightarrow$ on configurations. A configuration represents a program state during any stage of program execution. This could be either an unfinished program execution,

represented by a command sequence and the current graph; the final graph, after all commands have been executed; or a failure state, represented by the special element `fail`:

$$\rightarrow \subseteq (\text{ComSeq} \times G_{\mathcal{L}}) \times ((\text{ComSeq} \times G_{\mathcal{L}}) \cup G_{\mathcal{L}} \cup \{\texttt{fail}\}).$$[1]

$$[\text{Call}_1]\frac{G \Rightarrow_R H}{\langle R, G\rangle \rightarrow H} \qquad [\text{Call}_2]\frac{G \not\Rightarrow_R}{\langle R, G\rangle \rightarrow \texttt{fail}}$$

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

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

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

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

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

Figure 3.7: Inference rules for core commands [Plu12]

The rules contain meta-variables, considered to be universally quantified. R stands for a rule call in RuleSetCall, C, P, P' and Q stand for command sequences in ComSeq and G and H stand for graphs in $G_{\mathcal{L}}$. Each rule has a premise and a conclusion separated by a horizontal bar. $\rightarrow^+$ is the transitive closure of $\rightarrow$. The notation $G \not\Rightarrow_R$ means that there does not exist a graph H such that $G \Rightarrow_R H$.

The semantics of the `if-then-else` and `try-then-else` are non-standard. Normally, a boolean predicate is tested to see which branch the program will take. However, in GP, this predicate is a command sequence C. The path taken depends on whether an application of C to the input graph succeeds or fails. The rules for conditional branching are described below.

[If_1]: If the first attempted application of the command sequence C to the graph G succeeds, generating the graph H, then continue by applying the command sequence P to G.

[If_2]: If the first attempted application of the command sequence C to the graph G fails, then continue by applying the command sequence Q to G.

[Try_1]: If the first attempted application of the command sequence C to the graph G succeeds, generating the graph H, then continue by applying the command sequence P to H.

[1] $G_{\mathcal{L}}$ is the class of graphs over $\mathcal{L}$.

[Try_2]: If the first attempted application of the command sequence C to the graph G fails, then continue by applying the command sequence Q to G.

The two branching constructs behave differently if the `then` clause is reached. `if` is non-destructive: any changes made to G by C are undone before the next command sequence is executed. On the other hand, `try` will keep the modified graph H, only reverting back to the old graph G if C fails.

It should be noted that, since rule application is nondeterministic, there could be many ways to apply the command sequence C to a graph G. It is possible that some executions fail while others succeed. In GP1, a conditional statement could only branch to the `else` clause if all possible executions of C failed on G [Plu09]. In the worst case, all such executions would have to be performed, which is extremely inefficient. The semantics were rewritten to remove this possibility. Instead, only one possible execution is tested. This semantic modification is at no cost to the programmer because, in practice, no more than one execution of the predicate command sequence is ever required. It is the programmer's responsibility to be aware of the semantics and to write their programs accordingly. Despite the non-determinstic behaviour of the semantics, constructing example GP programs involving the conditional branching constructs has proven straightforward.

As-long-as-possible iteration has been similarly modified. Now, the loop will break if the first execution of the subprogram on the graph fails, rather than if all executions fail.

$$[\text{Or}_1]\ \langle P \ \texttt{or}\ Q, G\rangle \to \langle P, G\rangle \qquad [\text{Or}_2]\ \langle P \ \texttt{or}\ Q, G\rangle \to \langle Q, G\rangle$$

$$[\text{Skip}]\ \langle \texttt{skip}, G\rangle \to G \qquad [\text{Fail}]\ \langle \texttt{fail}, G\rangle \to \texttt{fail}$$

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

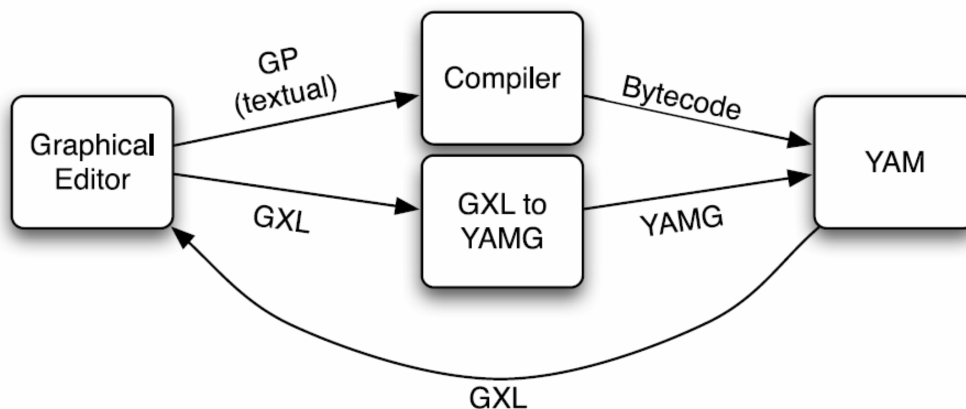
$$[\text{Try}_3]\frac{\langle C, G\rangle \to^+ H}{\langle \texttt{if}\ C\ \texttt{then}\ P, G\rangle \to \langle P, H\rangle} \qquad [\text{Try}_4]\frac{\langle C, G\rangle \to^+ \texttt{fail}}{\langle \texttt{if}\ C\ \texttt{then}\ P, G\rangle \to G}$$

Figure 3.8: Inference rules for derived commands [Plu12]

GP is one of few graph programming language with a complete set of formal semantics. One reason for this is that other graph programming languages are often significantly more complicated than GP, thus it is much harder to construct a formal semantics. The semantics of GP have two major benefits: they give clear, unambigious design goals for those implementing GP, and they are necessary to formally reason about GP programs. Significant work regarding the latter has been undertaken: a proof system has been developed for a subset of GP programs that is sound with respect to GP's formal semantics [PP12].

3.5 Implementation

The diagram below depicts the implementation of GP1. The user constructs the host graph, the rules, and the program in the graphical editor. The editor stores the host graph textually using the Graph Exchange Language (*.gxl), a standard textual graph representation format [Win+02]. The rules are also stored textually in a GPX file (*.gpx). The syntax of the GPX file is very similar to GXL (as rules are a pair of graphs) with some additional syntax for control constructs and other features not supported by GXL. The compiler, written in Haskell, transforms the input graph to an internal graph format YAMG. It also converts the GXL file to bytecode. Execution of GP programs is handled by the York Abstract Machine (YAM) which takes the bytecode and the compiled host graph as input. The output graphs are returned to the editor for the user to view.



3.5.1 The Compiler

We describe the transformation of a GP program to YAM-executable bytecode. First, a searchplan is created from each rule. A searchplan is a common implementation technique for graph matching. The matching process is decomposed into a sequence of primitive matching operations according to the left-hand side of the rule. This technique is ideal for GP since basic operations fit the model of a low-level abstract machine executing bytecode. There are various ways to generate a searchplan, some of which we shall consider in Chapter 4. GP's method is static: the searchplan generation does not take into account the input graph, only the rule. Although this method does not always generate the most efficient searchplan, there is no additional overhead to perform computations involving the host graph. This fits with GP's implementation, as its efficiency comes from the runtime execution of bytecode rather than an optimal compiling procedure.

In order to minimize choice and backtracking, primitive search operations are ordered by their determinism as follows [MP08b]:

1. Check clauses of the schemata condition whose variables have been instantiated.

2. Find source and target nodes of matched edges.

3. Find an edge whose source and target nodes have both been matched.

4. Find an edge whose source or target node has been matched.

5. For conditions of the form `not edge(v,w)` where either v or w has been matched, find the other node.

6. Find a node.

This searchplan also prioritises items with value labels over those with variable labels. Despite the ordering, there is still some choice involved, which is determined by the order a particular item appears in the text file of the rule.

Once each rule has been compiled, the generated bytecode needs to be organised according to the control constructs of the program. Sequential composition is a straightforward concatenation of the bytecode. Nondeterministically applying rules from a set is achieved by ordering the bytecode according to the textual order of the rules in the program declaration, and generating appropriate failure procedures: if one rule fails, then backtrack and try the next rule. If the last rule fails, then the whole rule set fails. As-long-as-possible iteration is implemented by first generating a failure procedure that continues as normal on failure as opposed to backtracking, then executing the bytecode for the rule, and repeating this until the rule fails. If-then-else is implemented by generating a failure procedure to execute the else branch on failure and executing the bytecode for the condition subprogram. If no failure occurs, the graph changes are undone so the then branch can be executed on the original graph as the semantics dictate.

The next section will give the reader an insight into how the YAM works to demonstrate, among other things, how the machine configures what it does upon failure and how it undoes graph changes.

3.5.2 The York Abstract Machine

This section relies heavily on [MP08b], a complete description of the YAM. Some changes made after the writing of that paper are documented in [MP08a], in particular the instructions introduced for failure handling.

The YAM was inspired by Warren's Abstract Machine for Prolog [AK91], as Prolog is also driven by choice and backtracking. Both machines use a stack-based architecture with environment frames to control backtracking.

A YAM state consists of the current graph, the three stacks and several integer values. The purpose of the three stacks are described below.

Data Stack
The data stack is used for storage and retrieval of integers and strings. Most of the YAM instructions operate on the data stack. Such instructions include stack operations such as pushing, popping and copying the top of the stack; integer operations such as binary addition and subtraction; and graph query operations, where the data stack stores an identifier or a label used by the query.

Choice Stack
The choice stack is responsible for storing all details necessary for backtracking. It contains choice points and environment frames. Both frames and choice points are created whenever a choice is made. Environment frames contain a fixed number of registers, determined at compile time, that store a copy of the data stack and a program counter. Choice points consist of the number of graph changes made at the creation time of the choice point, a program position to jump to upon failure, and pointers to its associated environment frame and the previous choice point. There are several YAM instructions to modify the choice points and hence modify the failure behaviour. An example is `UpdateFail`, which changes the go-to program position of the top choice point on the choice stack according to the value (code location) on the top of the data stack.

Graph Change Stack
This is a collection of frames that are pushed onto the stack whenever a graph-modifying instruction is executed. Each frame describes how to undo the changes made to the graph. Whenever a backtrack is made, a number of frames are popped off the stack according to the most recent choice point. The modifications contained in each frame are applied to the current graph as they are popped off. This means that only one graph needs to be stored in memory. This is necessary since creating a new graph in memory for each choice point is clearly infeasible.

This architecture also allows the YAM to return more than one output graph. The user can choose whether to output the first result of a program or all results. If the latter is chosen, a failure behaviour is created to trigger a failure upon finding each result, terminating when the machine backtracks past the first choice.

The graph data structure used by the YAM is very complex. It is designed for efficient graph querying.

- Node structure: Contains the node identifier (unique integer), the node's label, indegree and outdegree. It also has four hashtables. Two hashtables store the node's incoming edges, where the keys are labels and the values are ordered lists of edge identifiers for all edges with that label. One is indexed by the labels of these edge's source nodes, and the other is indexed by edge label. Similarly for outgoing edges.

- Edge structure: Contains the edge identifier, the edge's label and the node identifiers of its source and target.

- Graph structure: Contains two integer functions that generate the next unused node identifier and next unused edge identifier. There are also two hashtables to store nodes and edges. The keys are identifiers and the values are pointers to the associated item. Two additional hashtables map labels to ordered lists of identifiers for both nodes and edges.

The large amount of hashtables are required to efficiently manage any form of graph query by hash lookups and list intersection. Finding the set of all nodes with label l is achieved by looking up the list of node identifiers for the label l. Finding the ith node in the list will take i units of time. The more complex query of finding all edges with label x whose target is the

node n and whose source has label y is achieved by intersecting two lists: the list of incoming edges to node n indexed by edge label x, and the list of incoming edges to node n indexed by node label y. Since the lists are ordered, intersecting them is very efficient. A drawback to this complex structure is the slow updating of graphs. For instance, relabelling a node will not only modify its own node structure and hashtables, but the hashtables of all nodes incident to it and the node label hashtable in the graph structure. However, there are typically far more querying operations than update operations, so it is beneficial to enhance querying speed at the cost of slower graph updating.

During rule application, graph change operations are made in a specific order to prevent the creation of dangling edges: delete edges, delete nodes, relabel items, add nodes and add edges.

The York Abstract Machine is a very efficient graph computation machine. It can precisely control the mechanics of backtracking which supports the nondeterministic nature of graph programs and enables GP's control constructs to be easily encoded. Due to the underlying graph data structure, computation on the graph, particularly query operations, is very fast. It is also efficient in memory storage. Despite the complex graph representation, the machine never needs to store more than one graph. It is worth noting that the data stack does not contain many items as they are popped whenever a graph query is made. The YAM is also very flexible in the sense that it will not need much modification, if any, to accomodate new features, because it already has all the tools necessary to execute generic graph computations. Instead, it would primarily be the job of the compiler to organise the bytecode it produces with respect to changes in GP's syntax or semantics.

3.6 Summary

We have introduced GP, a graph programming language that prides itself on its simplicity and abstraction. These features make it straightforward for a programmer to construct powerful graph programs without delving into low-level data structures by using the high-level conditional rule schemata. Graphs and schemata are drawn in a graphical editor which abstracts away from the underlying mechanisms. Furthermore, GP's lack of complexity facilitates a small abstract syntax and complete formal semantics for the language. This in turn admits a efficient abstract machine implementation and formal reasoning about program correctness.

4 Other Graph Programming Languages

The purpose of this chapter is to examine and critique the subgraph matching implementations of three external graph programming languages: PROGRES, GrGEN and AGG. As we have already seen, GP does not take into account the host graph when generating its searchplan. In contrast, two of the systems introduced in this chapter use dynamic searchplan generation, which considers the structure and features of the host graph in order to develop an efficient searchplan. For each language we give a brief overview of their interfaces and representation of graph programs, and discuss their graph matching implementations in detail. The chapter concludes with an overview of a tool contest to compare running times of various graph transformation systems for a specific graph problem which featured GrGEN, AGG and GP, before comparing the graph matching strategies of all four languages.

4.1 PROGRES

PROGRES stands for Programming with Graph Rewriting Systems. It is a graph programming language with a supporting environment, and it is one of the oldest programmable graph rewriting systems. A comprehensive description of the language is given in [SWZ99], and a shorter description in [SWZ95]. We note some of the system's key features.

At its core, writing programs in PROGRES is almost identical to writing programs in GP. Rule schemata are declared in a graphical editor, and they are organised with imperative-style control constructs. Conditions on schemata can be specified both graphically and textually. The user also has the option of programming graphs and rules in a purely textual form.

If we look deeper, the differences between PROGRES and GP become apparent. Instead of GP's node labels and edge labels, PROGRES supports the declaration of node types and edge types. These types are graphically represented as labels, but they have additional functions. Node types constrain the node's attributes, which can be viewed as components of the node's label. PROGRES offers three standard data types for node attributes: boolean, integer and string. However, one can import external data types from a definition made in a C-compatible programming language. Edge types constrain both the number of edges of that type present in the graph and which node types it is incident to. In addition, node classes can be defined, with subclasses and class inheritance. This object-oriented approach enables users to declare not only conditional rules, but graph schema. A graph schema is an abstract way of defining a class of graphs with certain static properties. Node types and edge types allow the specification of both structural restrictions and attribute restrictions for a graph class.

As a consequence, PROGRES can be used as both a graph specification language and a graph transformation language. Due to the complexity of the language, the PROGRES environment has several analyzers and debuggers to perform syntax checking, type-checking and other forms

of verification. The environment does not seem to provide automatic verification of whether rules preserve a particular graph schema. Hence it is the responsibility of the programmer to write schema-preserving rules. This can be done with the use of declaring node and edge types, attribute conditions and other restrictions.

One interesting and unique feature of PROGRES is that it offers optional node matching in a rule. Nodes in the left-hand side of a rule can be declared as optional. If a suitable match exists for these nodes, which can be extended to a total match, then the rule is applied as standard. If there is no total match for any of the optional nodes then, instead of failing, the rule is applied to only the nodes matched.

4.1.1 Searchplan Generation

This section follows the description of PROGRES' searchplan generating algorithm in [Z9̈3].

The algorithm to generate a searchplan in PROGRES has three stages. First, the left-hand side of a rule, which may have textual restrictions, is converted into a single graph that captures the structure of the left-hand graph along with the conditions. Second, action nodes are added, each representing a primitive search operation, to form a new graph. This graph is called the action graph, and it models all possible search plans that match the LHS. Third, a searchplan is chosen from the action graph with respect to a heuristic cost function.

Figure 4.1 shows an example of a left-hand side of a rule before and after the first stage described above. The resulting graph is a diane graph which stands for a directed, attributed, node and edge-labelled graph.

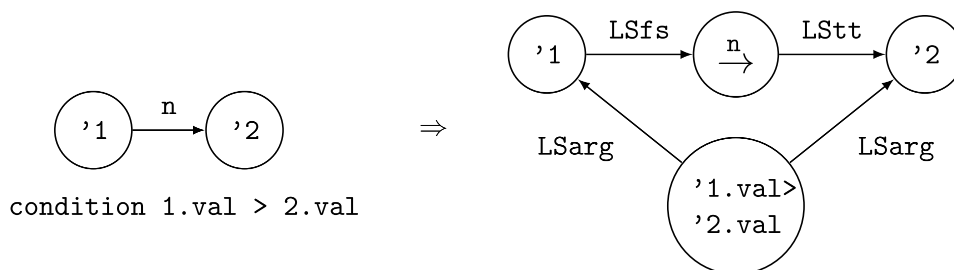


Figure 4.1: Conversion of a rule's LHS to a diane graph

The numbers in the label of the nodes refer to their identifiers. In PROGRES they are preceded by an apostrophe. We assume a node type which has a single integer attribute `val`. Thus, this rule will match two nodes connected by an n-labelled edge such that the attribute of the source node is greater than the attribute of the target node. The diane graph is a representation of the rule as a single graph with no additional text. Nodes, edges and conditions of the LHS are represented as nodes in the diane graph. The edges in the diane graph represent the dependencies of each left-hand component. The source of an LSfs-edge is a node on the left-hand side of a rule (left-node), and its target is either an outgoing left-edge or a restriction on that left-node. The source of an LStt-edge is a left-edge on, and its target is the target left-node of that left-edge. The source of an LSarg-edge is a restriction or attribute condition, and its target(s) are any left-nodes

affected by the condition. It is not clear which restrictions should be incident to LSfs-edges, and which should be incident to LS-arg edges.

The next stage is to construct the action graph from the diane graph. This involves adding action nodes for each possible search operation. Figure 4.2 depicts two typical patterns in this construction. Actions nodes are rectangular, while nodes representing items in the left-hand side are circular.

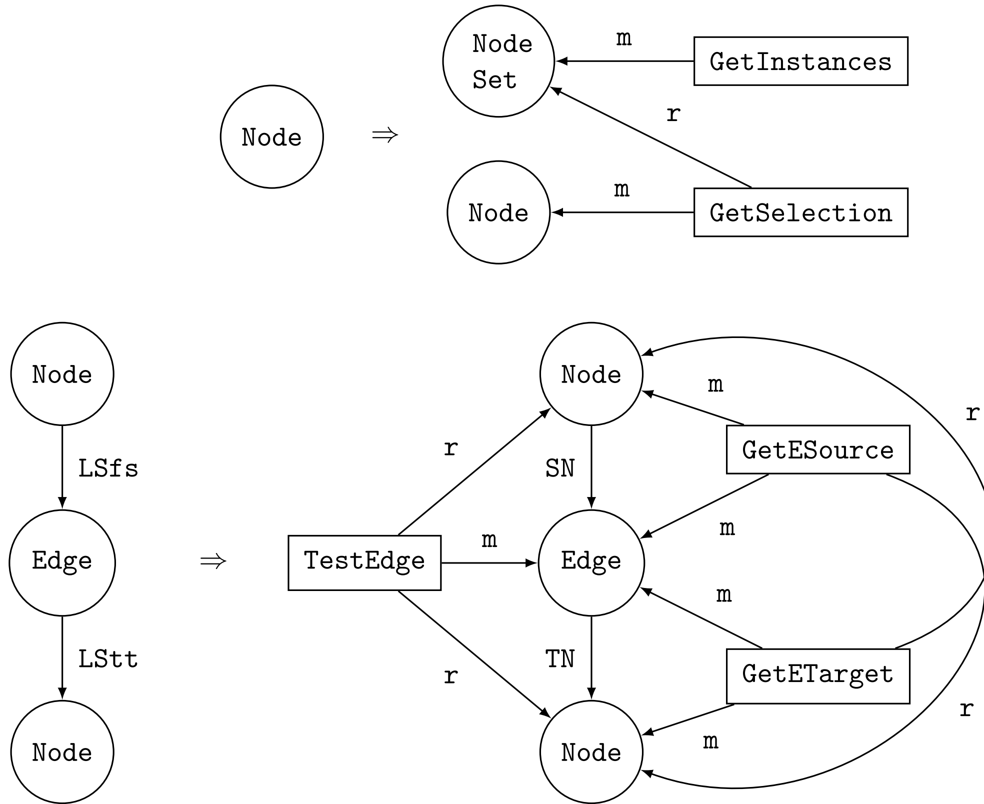


Figure 4.2: Action graphs for matching a node and matching an edge

The top pattern shows the addition of actions to match a single left-node. The GetInstances action will generate all possible matches. Since there may be more than one match, the set of nodes in the host graph that could be matched to the left-node is represented by the node labelled 'Node Set'. The m-labelled edge signifies that its target is matched by the action represented by its source. GetSelection chooses a single node from this set and matches it to the left-node. It requires a matched set as input, which is signified by an r-labelled edge from the action node to its input.

The bottom pattern shows the three actions that are generated when matching two left-nodes connected by a left-edge. The TestEdge action will match an edge given that its two incident nodes are already matched. As the edges in the action graph indicate, this action requires two matched left-nodes as input and finds a match for their common left-edge. Similarly, the other

two actions will match an edge and one of its incident nodes given that the other incident node has been matched.

PROGRES has several other actions to test attribute conditions and other restrictions.

The final stage of the algorithm is to generate a search plan from the action graph. Zündorf defines a searchplan as a sequence of actions taken from the action graph such that every item in the LHS is matched by an action in the sequence and that every item required by an action is matched by an action occurring earlier in the sequence. A searchplan always exists for every action graph since, intuitively, GetInstances and GetSelection can match all nodes without any requirements, and every other action only requires matched nodes.

Each action is assigned a cost by a heuristic cost function. This cost function does not involve an analysis of the host graph. Rather, it estimates the costs of each operation based on a number of factors and assumptions. The time complexity of each action is taken into consideration, based on the complexity of each corresponding GRAS query. GRAS is the underlying database management system for the PROGRES system, built by the PROGRES team specifically for the storage and querying of graphs. Another factor is the probability that a particular action generates no result. Actions that are more likely to fail are assigned a lower cost, since they have a higher chance of establishing that there is no match and terminating the search, thereby increasing efficiency. The most important factor, however, is the expected number of matches. It is not shown how this is computed; rather, we are led to believe that this is static and doesn't depend on the host graph. Consequently, actions such as GetSelection and GetInstances have a much higher cost than, say, an action that tests an attribute condition. This makes sense as these are highly nondeterministic actions, and should only be used when absolutely necessary.

With the action graph generated and the cost for each action computed, the searchplan can be generated. The generation is greedy: actions are iteratively added to the search plan. In each iteration, the cheapest action of the set of actions whose required items have been matched is added to the sequence until a complete searchplan has been computed.

4.1.2 Fujaba

Fujaba is a tool that supports model-driven engineering. Users can create system specifications with a UML-based specification language and use them to generate object-oriented Java code. Models are manipulated with graph transformation rules called story patterns, an extension of PROGRES' graph model. Fujaba also inherits and extends PROGRES' graph schema with more object-oriented features [Fis+00] named story diagrams. They diagramatically represent control flow of a system by linking activities, some of which can be story patterns.

The graph matching algorithm in Fujaba is the same as that in PROGRES. However, it improves over the PROGRES system: Fujaba drops the backtracking semantics of PROGRES' control structures. According to the developers, backtracking is generally not used in real-world applications, and the lack of backtracking allows for a direct translation to Java [Fis+00].

Variables in Fujaba's story patterns are either bound, unbound or maybe bound [Det+12]. A bound variable must have been matched previously, either from a previous rule application or a specified match from the user. Unbound variables are matched during graph matching. Maybe bound variables are treated as unbound variables if they have not been matched in a prior execution, otherwise they are treated as bound variables with the previous binding. Fujaba requires at

least one bound variable in each story pattern as a starting point for subgraph matching. This is akin to the concept of rooted graph programs, presented in Chapter 5.

4.2 GrGEN

GrGEN is a relatively new graph programming tool, first released in 2003. Since then, the system has been re-implemented as GrGen.NET, which replaced the original virtual machine interpreter of GrGEN with .NET assemblies that perform graph rewriting. We focus on older literature written before the development of GrGEN.NET [Gei+06; Bat06]; however, it retains the core constructs of the original GrGEN.

GrGEN offers a construct similar to the graph schema of PROGRES called a meta-model. A meta-model is a declaration of classes and types. Unlike PROGRES, GrGEN supports both node classes and edge classes, as well as other object-oriented features such as subclasses, class inheritance and attributes. Connection assertions are also available when specifying a meta-model. They restrict the connectivity and frequency of edges of a particular class or type.

Rules in GrGEN are very flexible. For instance, the user has the option of enabling non-injective matching. This is possible as GrGEN is based on the single-pushout approach to rule application which does not forbid non-injective matches. It is also possible to place conditions on attributes and types. In addition, negative application conditions can be written that specify a certain structural pattern that must not exist in the host graph in order for the rule to be applied. Rule application can cause attributes or even types of nodes and edges to be changed.

Although GrGEN does have a GUI for viewing graphs, meta-models and rules must be programmed textually. While this is less visual and abstract, the Java-like syntax lends itself nicely to the object-oriented style of meta-models. The language and syntax is tailored for writing graphs and rules, and don't require the user to fiddle with pointers and memory allocation. In addition, it is not necessary to program both graphs of a rule. Only the left-hand graph and the items that are modified by the rule, along with the nature of their modification, need to be specified.

4.2.1 Searchplan Generation

This section follows the description of GrGEN's searchplan generating algorithm in [Bat06].

GrGEN heuristically generates a searchplan at runtime based on the structure of the host graph. The searchplan generation algorithm is very similar to that of PROGRES. It generates a graph representing all possible searchplans to match the LHS of a rule and greedily constructs the cheapest searchplan with respect to a cost function. One of the major differences between the approaches of PROGRES and GRGEN is that GrGEN performs an analysis of the host graph to compute the heuristic cost of each search operation.

Another big difference is that GrGEN has only two searchplan operations. The first is lkp (lookup) which takes an item from the left-hand graph as an argument and matches it to a corresponding item in the host graph with the same label. If the argument is an edge, this operation will also check the labels of the source and target of the edge. The second is ext (extension).

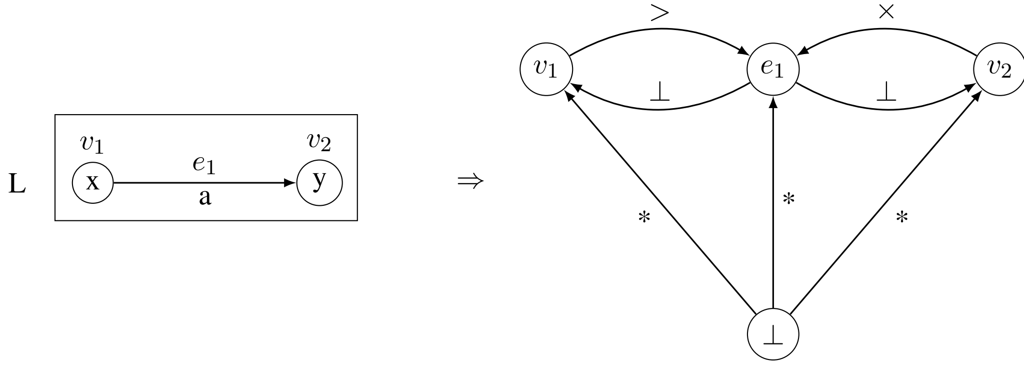


Figure 4.3: A left-hand side of a rule and its corresponding plan graph

This operation takes an edge from the LHS and one of its incident nodes which has already been matched. It matches this to an edge in the host graph incident to the image of the node parameter, and the second node incident to that edge, provided that all associated labels match. For injective matches, extra conditions are required to prevent queries from matching distinct items in the pattern graph to the same item in the host graph. The existence of only two nondeterministic operations means that an operation will often have several possible matches in the host graph.

The plan graph P, analogous to the action graph of PROGRES, is constructed from the LHS L. P has a root node labelled $\perp$. Every other node in P represents an item in L. There is a $*$-labelled edge from the root node to every other node representing a lookup operation. Each plan graph node representing a left-hand edge has an incoming edge from the plan graph nodes representing its source and target nodes, with labels $>$ and $\times$ respectively. These edges represent extension operations. There are also parallel edges in the opposite direction from the edge node to its source and target nodes. These are both labelled $\perp$. Edges in P representing operations are also labelled with their cost according to the cost function defined below. $\perp$-labelled edges have a cost of 0. Their purpose is made clear when we explain how to generate a searchplan from a plan graph. An example plan graph is shown in Figure 4.3.

The goal of GrGEN's algorithm is to remove as much choice as possible by prioritising operations with potentially less choice. The amount of choice is estimated by a cost function that assigns a cost to each operation as follows: the cost of $lkp(x)$ is the number of host graph items with the same category (node or edge) and label as x. If there are no such items, the cost is 1, otherwise it is the number of appropriate items. For injective matching, the same node or edge cannot be matched twice, hence the cost function for a lookup is an overestimation as the cost may be greater than the amount of items it can actually be matched to.

Let $v \in V_L$ and $e \in E_L$ have labels α and β respectively. An instance of (v, e) in a graph G is an occurrence of a node in G with label α incident to an edge with label β. The multiplicity of an instance is the number of edges with label β connected to this node. Direction is also considered: instances in which v is the source of e have a different multiplicity from those in which v is the target of e. The cost function of $ext(v, e)$ is then defined as

$$\text{cost}(ext(v,e)) = \max\left(1, \big(\prod_{j=1}^{m} n_j\big)^{\frac{1}{|V_S|}}\right)$$

where $n_1, \ldots, n_m$ are the multiplicities of all instances of (v, e) and V_S is the set of all nodes in the host graph with the same label as v.

This is the multiplicative mean of all the multiplicities of instances of (v, e) in the host graph. This is a better representation of the growth of the set of partial morphisms, after a match for e and v is added, than the more common arithmetic mean. If the multiplicities were to be summed, a single instance with a large multiplicity would greatly increase the cost, even if there were many other instances with a small multiplicity. This is not desirable since it might be more likely to match one of the instances with a small multiplicity. The multiplicative mean avoids this problem.

The data needed to compute these costs is obtained from an analysis of the host graph. This proceeds as follows:

1. For each ε in the node label alphabet, create a counter and iterate over nodes in G, incrementing the counter for every node found with label ε.

2. Perform Step 1 for edges.

3. Compute the multiplicities of a node as follows: pick a node $v \in V_G$: For all edges e incident to v, increment the counter $u_{\beta,\varepsilon,d}$ where β is the label of e, ε is the label of the other node incident to e and $d = out$ if e is outgoing on v, otherwise $d = in$. Then set $r_{\alpha,\beta,\varepsilon,d} = r_{\alpha,\beta,\varepsilon,d} \cdot \kappa$ where α is the label of v in G and $\kappa = u_{\beta,\varepsilon,d}$ if $u_{\beta,\varepsilon,d} \geq 1$, otherwise $\kappa = 1$. The u counters are set to 0 after the r counters are updated to ensure multiplicities of instances are only considered once, while the r counters are initialised to 1.

4. Repeat Step 3 for all nodes in V_H.

The cost of a search plan with operations $o_1, \ldots, o_k$ is $c_1 + c_1c_2 + \ldots + c_1c_2 \ldots c_k$ where c_i is the cost of the operation o_i. The intuition is that the first operation, which is always a lookup operation, will match c_1 items in the host graph. This gives c_1 places to apply the next operation in the searchplan. This process repeats until all the operations have been applied. This is not an exact cost since the cost of a lookup is an overestimation in the case of injective matching, and the cost of an extension is not exact.

A spanning arborescence of a directed graph G with root r is a subgraph of G with a unique path from r to all other nodes in the subgraph. In [Bat06] it is proved that finding the spanning arborescence of a plan graph is equivalent to finding a valid set of primitive operations. That is, a set of primitive operations that can be ordered in such a way that no lookup operation takes a matched node as its argument, no extension operation takes an unmatched node or a matched edge as its arguments, and the entire set matches every item in the LHS. Not all operations in the set have to be included in this ordering. It follows that computing a minimal spanning arborescence (MSA), a spanning arboresence with minimum cost, is equivalent to finding the cheapest set of primitive search operations.

Once this set has been found, it is ordered by a simple best-first traversal of the MSA starting at the root node. Such a traversal would not be possible without the $\perp$-labelled edges in the plan graph. The operations are then sequenced in the order they were reached by the traversal. This means that not only is the overall set the cheapest one available, but the operations are ordered in a roughly increasing order of cost, further increasing efficiency of matching. To guarantee the validity of the resulting searchplan, items that have already been checked by an operation are marked, and any future operations which check the same item are omitted from the final plan. Due to the structure of the pattern graph, there can only ever be one lookup operation which is guaranteed to be the first operation in the searchplan.

4.3 AGG

AGG is a visual language to facilitate graph transformation [Tae00; Tae03]. Graphs and rules are constructed with a graphical editor similar to that of GP. AGG also has a node and edge type system where nodes and edges can have types and attributes. As we have seen, object-oriented themes are common among graph programming languages, but AGG has two nice features that make it stand out. First, types can be declared with identifiers as standard, but they can also be determined solely by the graphical representation of the nodes and edges. Various node shapes, colours and arc styles are available. This is a very nice exploitation of the inherent visual nature of graphs, which enables users to establish types at a glance. Second, AGG integrates with Java, which means that attributes can be both declared and modified with any Java type or method. This allows the construction of powerful rules without the need to learn a system-specific syntax.

Graph transformation in AGG, like GrGEN, is based on the single-pushout approach. As well as Java types, the left-hand side may contain variables, and the right-hand side may contain complex Java expressions to specify the renaming of attributes with respect to any variable assignments made, not unlike GP's relabelling with integer and string operations. Negative application conditions are also supported. As in GrGEN, these are graphs representing patterns that must not occur in the host graph if the rule is to be applied. A description of the application of a rule set is give in the AGG User Manual [Run06]. This is achieved in one of two ways. The first is step mode, where only one rule application is made for a selected rule. The user can define the match, or it can be generated automatically. This is a good way to quickly verify or debug a rule. The second is interpretation mode, where a set of rules is applied nondeterministically until either no rules from the set can be applied, or the user manually stops computation. Rule layering is the only control construct available. Each rule is assigned an integer-valued layer. When execution starts, only rules from the lowest layer are chosen. If no rules from that layer can be applied, control passes to the next highest layer. This repeats until no rules in the highest layer are applicable.

The AGG system offers some tools for property checking. The first is critical pair analysis. For any two rules, one can construct a set of pairs of subgraphs of these rules that represent all possible conflicts between the rules. That is to say, the application of one rule may change the structure of the graph in such a way that another rule cannot be applied. Critical pairs can be used to check for a desirable property of graph grammars called confluence [Plu05]. If a graph grammar is confluent, then none of its rules depend on one another, and will thus generate the

same graph regardless of the order of rule application. This is valuable because if confluence is proven, then backtracking is unnecessary and can be disabled, greatly increasing efficiency. AGG also offers graph parsing to test whether a graph belongs to a certain graph grammar by applying inverse rules. Critical pairs are also used in this procedure to attempt to minimise the number of inverse rules applied, preferring rules that have the fewest potential conflicts. The final tool is consistency checking. Users can define a number of conditions that they wish to preserve in their program, and AGG verifies if the rules preserve these conditions for a particular start graph.

4.3.1 Graph Matching by Constraint Satisfaction

AGG is unique among the systems discussed in this report as it enables the programmer to define a match from the left-hand side of a rule to the host graph. However, this does not remove the need for a matching algorithm. So far we have seen the construction of a searchplan, a sequence of basic search operations, to build match. AGG takes a radically different approach: it solves graph matching as a constraint satisfaction problem (CSP).

The CSP is an old and well-studied problem in computer science and mathematics. The idea is to assign values from a finite domain to a finite set of variables such that certain conditions are met. Formally a CSP is a triple (X, D, C), where X is a finite set of variables, D is a finite set of domains corresponding to each $x \in X$, and C is a finite set of constraints. Each constraint is defined on a set of variables $(x_1, \dots, x_n)$, called its scope. A constraint is a subset of $(D_1 \times, \dots, \times D_n)$, specifying all legal assignments of values to variables in its scope. A solution to a CSP is a total assignment of variables to values in their domain that satisfies all the constraints. There are several ways to represent graph matching as a CSP. We follow the model in [Rud98] as it is the method cited in AGG's user manual.

Given a LHS of a rule L and a host graph G:

- $X = L_V \cup L_E$
- Variables representing nodes have domain G_V. Variables representing edges have domain G_E
- Label constraint: $\forall x \in X, C^{type}_{(x_i)} = \{d \in D_i \mid l(x_i) = l(d)\}$
- Source constraint: $\forall v \in L_V, e \in L_E$ with $s_L(e) = v$,
  $C^{src}(v, e) = \{(d_V, d_E) \in D_V \times D_E = G_V \times G_E \mid s_G(d_E) = d_V\}$
- Target constraint: $\forall v \in L_V, e \in L_E$ with $t_L(e) = v$,
  $C^{tar}(v, e) = \{(d_V, d_E) \in D_V \times D_E = G_V \times G_E \mid t_G(d_E) = d_V\}$

This CSP is equivalent to the subgraph matching problem. That is, a valid solution of the CSP above represents a morphism $L \to G$. Nodes (edges) in L can only take values from their domains: the set of nodes (edges) in G. Hence nodes can only be matched to nodes, and edges can only be matched to edges. The label constraint ensures that items in L can only be matched to items in G with the same label. Finally, the source and target constraints respect the source and target preserving requirements in the definition of a morphism.

As the constraint satisfaction problem is well-researched, there are a variety of efficient algorithms and technique to solve it. Therefore this approach removes a lot of the concerns that can arise when designing a matching algorithm from scratch. However, there is some overhead in converting the problem to CSP and back, and existing CSP solving strategies may not be completely suitable for this particular instance of the problem.

4.4 Generation of Sierpinski Triangles

In 2008, various tools took part in a benchmark designed to evaluate the performance of graph transformation systems [Tae+08]. The goal was to create a program that iteratively generates Sierpinski triangles. The Sierpinski triangle is a fractal structure that, when modelled as a graph, grows exponentially in the number of nodes and edges with each generation. The initial triangle and the first generation are shown in Figure 4.4. Intuitively, to construct the next generation, one must place a node at the midpoint of each edge of every triangle in the structure and connect them with additional edges.

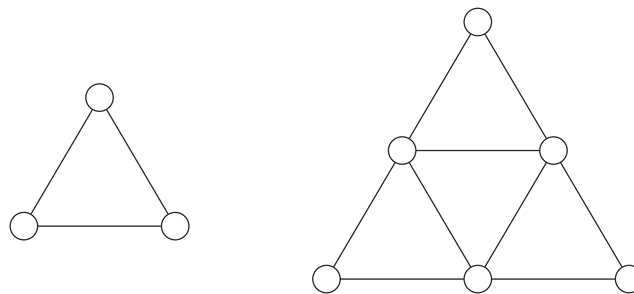


Figure 4.4: Initial and first generation of the Sierpinski Triangle

Since the graphs exhibit exponential growth, this is a strong benchmark for both time complexity and space complexity. The problem is also sufficiently simple as to be computable by any good graph transformation tool. Twelve tools participated in this benchmark, among them GP, Fujaba, AGG and GrGEN. The performance of all the participating tools is shown in Figure 4.5.

Fujaba clearly exhibited the best performance. It computed triangles up to generation 17, three generations more than any other tool, and it was in general faster than the other tools. GrGEN also performed very well. It was able to compute up to 14 generations, the 14th being computed in under two minutes. GP had an impressive showing, being the third fastest tool. In contrast, AGG was one of the slower performing tools.

The benchmark was designed to test the overall computational power of graph transformation tools, so there are too many factors involved to make any concrete assertions about this data. However, as subgraph matching is often the bottleneck of graph transformation with regards to speed, it is not unreasonable to state that the implementation of matching is one of the biggest factors contributing to the efficiency of a graph transformation system.

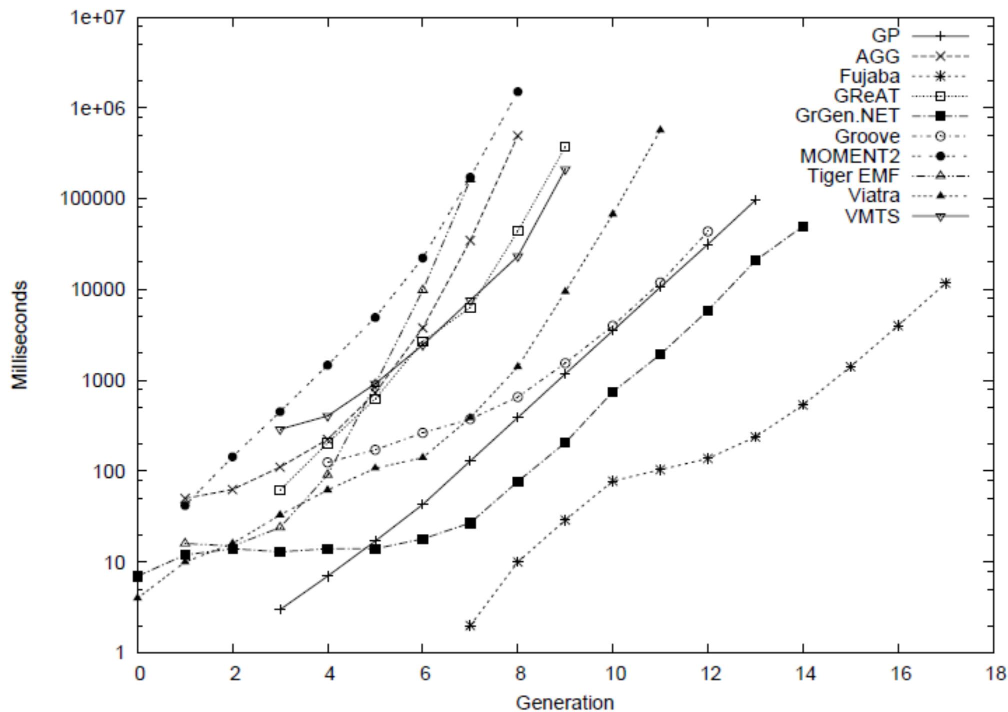


Figure 4.5: Running time of graph transformation tools generating Sierpinski Triangles. The time scale is logarithmic. GP's performance is shown by the line marked with crosses.

4.5 Overview of Graph Matching Strategies

There have been many approaches to efficiently performing subgraph matching. Among these is searchplan generation, which has been the main focus of the chapter. This method is most relevant to GP for two reasons. GP already uses searchplan generation to modest success, and it could be improved with a refined algorithm. Furthermore, clever searchplan generation algorithms are the driving force between the two fastest graph transformation tools according to the benchmark detailed in the previous section. We briefly cover alternative techniques before comparing searchplan implementations.

Bunke, Glauser, and Tran [BGT91] describe an algorithm that constructs a dynamic network, called a RETE-network, from a set of graph transformation rules at compile time. Each node in the RETE-network represents a distinct subgraph occurring in one or more rules. The whole RETE-network depicts an item-by-item buildup of all the left-hand sides in the rule set, where complete left-hand sides link to a production node representing the application of the corresponding rule. At run-time, the host graph G is fed into the network one item at a time, starting with its individual nodes. Due to the structure of the network, this ensures that all possible subgraphs of G on which a rule can be applied are stored in the production nodes of the appropriate rule. The RETE-network can then dynamically update itself upon the application of a rule, according to the changes in the host graph, to prepare for the next rule application. The efficiency

of this algorithm arises from the fact that the RETE-network, although large, only needs to be constructed once: minor changes in the host graph only require minor changes in the RETE-network. If a particular subgraph occurs in more than one LHS, it will only be represented once in the RETE-network, making the network a compact representation of a rule set. Moreover, the whole host graph is only processed once, before the first rule application. The drawback of the algorithm is its space complexity. The size of the RETE-networks grows rapidly as the number of rules increases.

Dörr [Dör95b; Dör95a] presents an iterative algorithm for subgraph matching which matches a node v in the LHS of a rule to a node in the host graph, and extends this partial morphism one edge at a time according to a pre-defined ordering of edges called an edge enumeration. The main contribution of Dörr's work is optimising the algorithm by reducing branching encountered during morphism extension as much as possible. He identifies places in the host graph where branching can take place by defining strong V-structures: a node with multiple incoming (outgoing) edges with the same label whose source (target) nodes also have the same label. Dörr then gives criteria for the existence of an edge enumeration that avoids strong V-structures, gives a set of predicates necessary for a rule application to introduce strong V-structures, and presents an efficient static analysis algorithm that computes an approximation to the number of strong V-structures. Under the appropriate conditions, Dörr's algorithm can perform subgraph matching in constant time. Host graphs do not always meet these conditions, but the static analysis techniques can be used to optimise the choice of edge enumeration. In fact, GrGEN's searchplan generation algorithm performs analysis using mechanisms similar to those in Dörr's work.

Finally, there is AGG's strategy. The left-hand graph of a rule and the host graph are used to transform the matching problem into an equivalent constraint satisfaction problem. This also requires querying the host graph, so this will involve some overhead. One benefit is that AGG leaves the murky depths of subgraph matching and transfers the problem to a well-studied area with many algorithms and solving techniques. An additional benefit is that this method is independent of the graph data structure, so future modifications to the data structure do not necessitate any changes to the matching algorithm. Despite this, AGG's implementation might not be the most efficient, as evidenced by its poor performance in the Sierpinski Triangle benchmark.

Now we move on to searchplans. First we consider the distinction between static searchplan generation and dynamic searchplan generation. These different approaches illustrate a tradeoff between the time taken to generate a searchplan and the overall efficiency of that searchplan. Since static searchplans only take into account the left-hand side of a rule, which in practice is small relative to the size of the host graph, there is little overhead in searchplan generation. However, this means that the searchplan is generic rather than being tailored to a specific host graph, so the execution of the searchplan will more often than not be slower than its dynamic counterpart. On the other hand, although dynamic searchplans enable a match to be found in near optimal time, they require overhead to analyse the host graph and compute costs.

PROGRES and GrGEN dynamically compute a searchplan in a similar fashion. They both build a new graph from the left-hand graph of a rule that represents all possible searchplans, and they both use a heuristic cost function to greedily select the cheapest searchplan from that graph. PROGRES' algorithm is not completely dynamic: the host graph is not explicitly taken into account. Instead, the heuristic costs of operations are estimated based on various criteria. GrGEN, in contrast, rigorously analyses the structures of both the host graph and the left-hand

side of the rule in order to generate a heuristic cost for each operation. This analysis is linear in the size of the host graph [Bat06] (where size is the number of nodes plus the number of edges), so it is still quite fast. GrGEN's plan graph encodes the operations in the edges, while PROGRES' action graph encodes them as nodes. This results in the action graphs of PROGRES being much larger than GrGEN's plan graphs, especially considering PROGRES supports nine primitive searchplan operations compared to GrGEN's two. Overall, GrGEN's algorithm is likely to be the more space efficient of the two, and it generates the superior searchplan since it explicitly interrogates the host graph. This should outweigh the fact that GrGEN's algorithm is more complex.

In contrast, GP has a static searchplan generation strategy. While this does not generate optimal searchplans, the underlying low-level abstract machine will execute these searchplans efficiently, as evidenced by GP's performance in the tool contest of the previous section. However, GP could be more efficient if it were to optimise its searchplans, admitting an even faster execution. The overhead of a revised searchplan generation algorithm would have to be considered, which is an interesting research topic.

5 Rooted Graph Transformation

This chapter contains content from [BP12].

So far we have only considered graph matching at the implementation level. However, we can also consider changing the nature of the high-level graph programs themselves to facilitate quick subgraph matching. This idea has been explored by Dörr [Dör95b], who proposes the idea of a label of a unique label in the host graph to initiate graph matching, and more recently with the concept of rooted graph transformation [DP06b; Dod08]. Intuitively, the programmer chooses a root node in the left-hand side of a rule, and a root node in the host graph. These nodes are matched immediately at the start of graph matching, and the remainder of the search takes place in the local neighbourhood of the host's root. This reduces the time it takes to find a match. Specifically, if the left-hand side is connected and the outdegree of all nodes in the host graph is bounded, then graph matching can be achieved in constant time [DP06b].

Rooted graph reduction rules can be used to efficiently determine whether a graph belongs to a particular graph language. This idea has been applied to pointer structures in C [DP06a]. An extension to C, named C-GRS, was developed to allow the programmer to construct shapes, transformers, and graph reduction rules to safely manipulate and verify pointer structures using C-like syntax. A translation to ANSI-C is provided in the documentation. An example graph reduction system to recognise cyclic lists is given in Figure 5.1.

The root node is not part of the pointer structure itself; it is used to make the matching process deterministic, which is necessary in C. In addition, anything slower than constant time would not be acceptable in the context of verifying C pointer structures. The rule `Reduce` is applied to the test graph as much as possible. All cyclic lists will reduce to the rooted 2-cycle, which is transformed into the accepting graph with the rule `Finish`. Conversely, any non-cyclic list will

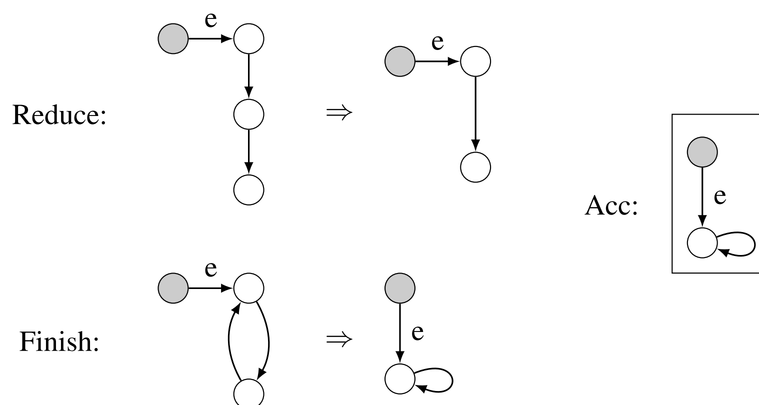


Figure 5.1: A graph reduction system for cyclic lists. Root nodes are shaded [DP06b].

not reduce to a 2-cycle, and hence cannot be transformed into the accepting graph.

Previous research only considered a single root node, but we show that this concept can be extended to a greater number of root nodes. It is our desire to utilise the efficiency of rooted rules in full generality. We use this as the basis for an extension of GP2 as it does not currently provide support for rooted rules and rooted matching. This feature would not only allow the programmer to construct rooted rules, but to combine rooted rules and non-rooted rules in a single program for maximum efficiency and flexibility.

5.1 Rooted Graphs and Rooted Rules

The standard definitions of Chapter 2 are extended to include rooted nodes in both the host graph and the rules.

A rooted graph is a pair (G, P_G) where G is a graph and $P_G \subseteq V_G$ is the set of root nodes. A morphism $g : G \to H$ is root-preserving if $g(P_G) \subseteq P_H$.

A rooted rule $r = \Big((L, P_L) \leftarrow (K, P_K) \to (R, P_R)\Big)$ is a pair of root-preserving inclusions $(K, P_K) \to (L, P_L)$ and $(K, P_K) \to (R, P_R)$. A direct derivation for a rooted graph rule is identical to the double-pushout given in Figure 2.3, with the added condition that both $L \to G$ and $R \to H$ are root-preserving.

5.2 A Matching Algorithm for Rooted Rule Schemata

GP's rule schemata can trivially accomodate root nodes. A formal extension is not provided here, but in the example GP programs of this chapter, root nodes in rules are displayed with thick edges. In this section, we consider the time complexity of applying a rooted rule schema to a host graph, and show that under certain conditions, it can be achieved in time independent of the host graph. This is due to the fast subgraph matching made possible by the presence of root nodes. The matching algorithm of Dodds and Plump [DP06b; Dod08] only considers connected graphs with a unique root node, distinguished by a special label. This is extended to accomodate for graphs with multiple roots and to incorporate label matching with respect to GP's syntax. In addition, unique labels are no longer required to identify the root nodes; they are encoded into the graph data structure.

The following assumptions are made for the rest of this chapter:

1. In the LHS of a rule, all nodes are reachable from some root.
2. Rule schemata and programs are fixed, instead of being considered as part of the input.
3. The LHS of a rule has at least one root node.
4. Integer and character operations are computed in unit time.

The second assumption fits within the context of graph programs: in program analysis, the program is considered fixed. The fourth assumption is consistent with the uniform cost criterion for random access machines, the standard complexity model in algorithm analysis [Ski08]. Furthermore, in our case study, the left-hand sides of all rules are left-connected.

Before stating the algorithm, we introduce some notation. Given partial graph morphisms $f, g\colon G \to H$, f extends g (f ext g) by a node v if Dom$(f_V) =$ Dom$(g_V) \cup \{v\}$ and Dom$(f_E) =$ Dom(g_E). f ext g by an edge e if Dom$(f_V) =$ Dom$(g_V) \cup \{s_G(e), t_G(e)\}$ and Dom$(f_E) =$ Dom$(g_E) \cup \{e\}$. Given a rooted graph (G, P_G), an edge enumeration (G, p), where $p \in P_G$, is a list of edges $e_1, \ldots, e_n$, such that $\{e_1, \ldots, e_n\}$ is the set of all edges reachable from p, e_1 is incident to p, and for $i = 2, \ldots, n$, e_i is incident to the source or target of an edge in $e_1, \ldots, e_{i-1}$

Given a rooted host graph (G, P_G), the left-hand side (L, P_L) of a fixed rooted rule schema, and an edge enumeration $e_{p_1}, \ldots, e_{p_n}$ for each $p \in P_L$, the algorithm in Figure 5.2 computes all matches of (L, P_L) in (G, P_G). The algorithm assumes that each node in L is reachable from some root. It incrementally constructs a set A of pairs of partial premorphisms $h\colon L \xrightarrow{par} G$ and partial assignments α_h. A *partial assignment* is a subset of Var$(L) \times (\mathbb{Z} \cup \text{Char}^*)^*$, where Var$(L)$ is the set of variables occurring in L. The roots in L are tagged whenever they are matched; initially they are all untagged.

Rooted Graph Matching Algorithm
$A \leftarrow \{(h : L \xrightarrow{par} G, \emptyset) \mid \text{Dom}(h) = \emptyset\}$
while $\exists$ untagged node $p \in P_L$ **do**
  $A_0 \leftarrow \{(h : L \xrightarrow{par} G, \alpha_{h'}) \mid h$ injective & root-preserving,
    $\exists\, (h', \alpha_{h'}) \in A : h$ ext h' by $p\}$
  tag p
  AssignmentUpdate(A_0)
  for $i = 1$ to n **do**
    $A_i \leftarrow \{(h : L \xrightarrow{par} G, \alpha_{h'}) \mid h$ injective & root-preserving,
      $\exists\, (h', \alpha_{h'}) \in A_{i-1} : h$ ext h' by $e_i\}$
    if $s(e_i) \in P_L$ **then** tag $s(e_i)$
    if $t(e_i) \in P_L$ **then** tag $t(e_i)$
    AssignmentUpdate(A_i)
  end for
  $A \leftarrow A_n$
end while
return A

Figure 5.2: The Rooted Graph Matching Algorithm.

The algorithm calls the procedure AssignmentUpdate, defined in Figure 5.3, which itself calls the procedure Check, defined in Figure 5.4. Both procedures are based on an observation on the structure of expressions in L.

Definition 14. An atomic expression is one of the following:

- An integer constant.
- An integer variable.
- An atomic variable.
- A string expression containing at most one string variable.

```
AssignmentUpdate
for each (h, α) in the input set do
    for unchecked items l ∈ Dom(h) do
        x ← label(l)                         ▷ label = l_G if x ∈ V_G, label = m_G if x ∈ E_G
        y ← label(h(l))
        if Mark(x) XOR Mark(y) then Reject
        a ← x.first; b ← y.first
        while a ≠ NULL do
            if a is a list variable then break
            if b = NULL then Reject              ▷ x has more atomic expressions than y
            if not Check(a, b, α) then Reject
            a ← a.next; b ← b.next
        end while
        if a = NULL then       ▷ check if a and b have the same number of atomic expressions
            if b = NULL then exit else Reject
        else
            temp ← b
            a ← x.last; b ← y.last
            if a is a list variable then   ▷ no atomic expressions to match after the list variable
                if b = NULL then               ▷ a assigned the sublist of h(l) from temp to b
                    α ← α ∪ {(a.first ↦ temp), (a.last ↦ b)}
                exit
            else
                while a not a list variable do
                    if &b = &temp.prev then Reject
                    if not Check(a, b, α) then Reject
                    a ← a.prev; b ← b.prev
                end while
                α ← α ∪ {(first(a) ↦ temp), (last(a) ↦ b)}
                exit
    end for
    Verify(α)
    mark l as checked
end for
```

Figure 5.3: The procedure AssignmentUpdate.

Since list expressions in the LHS of a rule must be simple (Definition 8), they are either a sequence of atomic expressions or a:l:a' where a and a' are sequences of atomic expressions and l is a list variable. Similarly, string expressions are either a string constant or $w.s.w'$ where w and w' are string constants and s is a string variable. The two procedures exploit this structure to efficiently test if a list of the form a:l:a' in the LHS matches with a label x in the host graph (Section 3.2 gives a formal description of label matching). Since x contains no variables, it

suffices to check if x has a prefix that matches with a and a suffix that matches with a'. Then the list variable l can be assigned to the possibly empty remainder of x without further verification. The matching of string expressions is analogous. For example,

- $1 : l : 1 : 2$ matches $1 : 1 : 2$ and $1 : 2 : 3 : 4 : 1 : 2$ where $l \mapsto$ `empty` and $l \mapsto 2 : 3 : 4$ respectively. However, it does not match $1 : 2$.
- "hello".s."world" matches "helloworld" and "hellomynameisworld", but not "hello" or "helloworl".

Lists and strings are stored internally as a doubly linked list with pointers $first$ and $last$ that point to the first and last element respectively.[1] Hence the first and last elements of a list, as well as the predecessor and successor of the current element, can be accessed in unit time. These lists are null-terminated at both ends. In addition, only the pointers to the first and last element are required to specify a mapping for a list, atomic or string variable as the rest of the list or string can be accessed through the $next$ and $prev$ operators.

Check takes three arguments. The first argument a is the atomic expression in a label l of L, the second argument b is the corresponding atomic expression in the image of l and the third is the partial assignment α passed from AssignmentUpdate. Check returns false if the two expressions cannot be matched, otherwise it updates the assignment accordingly and returns true.

The bulk of the procedure lies in the case where a is a string expression. No match is possible if b is an integer (values of type integer differ from strings representing an integer). The procedure then compares the strings character by character, returning false if a comparison fails, until the string variable s in a is found. If s is reached, and it is the last element in a, then the algorithm immediately updates the assignment with a mapping from s to the remainder of b. Otherwise, further character comparison starts at the end of both strings, progressing backwards. b may have fewer characters than a, in which case the strings do not match, and a character in b may be reached that was already compared during the first comparison phase. This is detected by comparing the address of the current position in b with that of $temp$. Hence, if $temp$ is reached before s, then false is returned. Otherwise, when s is reached again, it is assigned the unchecked portion of b.

AssignmentUpdate iterates over its input: a set of pairs of partial premorphisms and partial assignments. For each premorphism-partial assignment pair, it iterates over all unchecked items l in the domain of h to see if the label of l and the label of $h(l) \in G$ can be matched. If they can, the assignment is updated accordingly, otherwise the procedure removes the pair (h, α) from the set.

First, the Mark boolean flags of both labels are tested for equality. If they differ, then the labels do not match and the Reject subprocedure is called, which removes the current premorphism-partial assignment pair from the set and exits the procedure. Otherwise, a while loop is used to compare the labels one atomic expression at a time. Each comparison is performed by Check. When the while loop breaks, the current element in x is either the null symbol, marking the end

[1] In the procedure definitions, the node name a is used to refer to the value stored in a node a of a linked list, rather than the more standard $a.val$. $\&a$ refers to the address of a in memory.

Procedure Check*(a,b: atomic expression,* α*)*
case a is
integer constant: **if** $b \in \mathbb{Z}$ **then return** $(a = b)$ **else return** false
string constant: **if** $b \in$ Char* **then return** $(a = b)$ **else return** false
integer variable: **if** $b \in \mathbb{Z}$ **then** $\alpha \leftarrow \alpha \cup \{(a \mapsto b)\}$; **return** true **else return** false
atomic variable: **if** $b \in \mathbb{Z} \cup$ Char* **then** $\alpha \leftarrow \alpha \cup \{(a \mapsto b)\}$; **return** true **else return** false
string expression $w.s.w'$: w, w' string constants, s string variable
if $b \in \mathbb{Z}$ **then return** false
$c \leftarrow a.first$; $d \leftarrow b.first$
while $c \neq s$ **do**
if $c \neq d$ **then return** false
$c \leftarrow c.next$; $d \leftarrow d.next$
end while
$temp \leftarrow d$; $c \leftarrow a.last$; $d \leftarrow b.last$
if $c = s$ **then** $\alpha \leftarrow \alpha \cup \{(a \mapsto b)\}$ ▷ $s.first \mapsto temp$, $s.last \mapsto d$;
return true
else while $c \in$ Char **do**
if $\&d = \&temp.prev$ **then return** false
if $c \neq d$ **then return** false
$c \leftarrow c.prev$; $d \leftarrow d.prev$
end while
$\alpha \leftarrow \alpha \cup \{(a \mapsto b)\}$ ▷ $s.first \mapsto temp$, $s.last \mapsto d$
return true

Figure 5.4: The procedure Check.

of the list expression, or a list variable. In the first case, a check is made to see if the end of y has been reached. If so, then the for loop is exited by **exit**. Otherwise, there are expressions remaining in y that cannot be matched, and the premorphism is rejected. In the second case, atomic expressions are checked from the last elements of the list backwards, analogously to Check. If no rejection has occurred then Verify(α) is called. This subprocedure will reject if α maps the same variable to two different values, otherwise the partial assignment is valid, the item l is marked, and the process repeats until all items in Dom(h) have been verified.

5.3 Complexity of Rooted Rule Schemata

As seen in Chapter 3, applying a rule in GP requires several steps: constructing a injective premorphism from the left-hand side of a rule to the host graph, finding a valid assignment of variables to values for that premorphism, verifying the schemata condition and the dangling condition, removing items from $L - K$, adding items from $R - K$, and relabelling nodes.

Proposition 1. *The Rooted Graph Matching Algorithm returns the set of all total root-preserving injective premorphisms $L \to G$.*

Proof. First, the algorithm is guaranteed to terminate since $|P_L|$ is finite, and there are only a finite number of premorphism extensions possible as L has a finite number of nodes and edges.

We show that, once the algorithm terminates, A contains all total root-preserving injections $L \to G$ by induction. Let $\{p_1, \ldots, p_r\} \subseteq P_L$ be a set of root nodes in L that are not reachable from one another. Define L_i to be the subgraph of L consisting of all nodes and edges reachable from $p_1, \ldots, p_i$. Note that if two or more of L's roots are connected, L_i may contain more than i roots, but it cannot contain more than i roots from the set $\{p_1, \ldots, p_r\}$ by construction. After the ith iteration of the while loop,

$$\{h : L_i \to G \mid h \text{ is injective and root-preserving}\} \subseteq A.$$

Without loss of generality, let p_i be the ith root node of L chosen by the algorithm, and let $i = 1$. First, A_0 becomes the set of all premorphisms that map $p = p_1$ to a root node in G. Then the algorithm enters the for loop. In the first iteration, the premorphism set A_1 becomes the set of all injective, root-preserving extensions to premorphisms in A_0 by e_{p_1}. If a premorphism in A_0 has no such extension, then it is discarded and can no longer be considered. This process repeats, extending the premorphisms edge by edge and pruning non-injective and non-root-preserving premorphisms until the for loop terminates. At this point, by definition of an edge enumeration, all nodes and edges in L_1 have been considered. A is assigned the set A_n, which is precisely the set of all total injective, root-preserving premorphisms $L_i \to G$. Hence the statement holds.

Next, assume the statement holds for L_k where $k < r$. That is, A contains all total injective root-preserving premorphisms $L_k \to G$. Now consider the $k + 1$st iteration of the while loop. p_{k+1} is not reachable from any of $\{p_1, \ldots, p_k\}$ as all root nodes reachable from them were tagged in a previous iteration of the while loop. Therefore p_{k+1} is untagged, and thus not in the domain of any premorphism in A. A mapping from p_{k+1} to an unmatched root in G is added to all premorphisms in A. Then, as before, nodes and edges are added until all edges in the enumeration for (L, p_{k+1}) and their incident nodes have been matched, after which A becomes the set of injective root-preserving morphisms from $L_{k+1} \to G$.

The program terminates after the rth iteration of the while loop. By Assumption 1, all nodes in L are reachable from some root, hence $L_r = L$. It follows that A contains all total injective root-preserving premorphisms $L \to G$.

□

Theorem 1. *The procedure AssignmentUpdate runs in constant time on a single label x if each list, string or atomic variable occurs at most once in L.*

Proof. First, note that by Assumption 2, L and its labels are fixed. Let s be the maximum number of characters in a single string expression in L, and let t be the maximum number of atomic expressions in a single list expression in L.

In the worst case, x is a list expression with a single list variable which can legally match to the label of its image with a suitable assignment. All atomic expressions in x are evaluated which requires t applications of Check. The list variable is then assigned the unevaluated sublist of the host graph's label. We assume a single assignment update takes unit time.

The worst case running time for Check(a, b) occurs when b is a string and a is a string variable followed by a string constant equal to b. There are s character comparisons, $2s$ pointer traversal operations and s pointer address comparisons. The assignment is then updated, giving a time complexity of $4s + 1$.

No verification of the assignment is performed because all mappings added to the assignment have been for either string variables or list variables which only occur once in L by the condition. Therefore the total running time is $1 + t(4s + 1)$. □

Theorem 2. *The Rooted Graph Matching Algorithm runs in time independent of the host graph if the condition of Theorem 1 holds, the number of root nodes in G is bounded by r, and the degree of each node in G is bounded by b, where the degree of a node is the sum of its indegree and outdegree.*

Proof. First we count the number of times the set of premorphisms is updated. We assume a data structure where adding either a node, an edge, or a node and an edge to an existing morphism takes unit time. Let l be the number of roots in L. Then there are at most l iterations of the while loop and, within each iteration, at most $|E_L| = m$ iterations of the for loop. Note that by Assumption 2, m is constant.

Consider the execution of the first iteration of the while loop. First, a single root from L is matched to all unmatched roots in G. Since no roots have been matched yet, r partial morphisms are created. Then, in each iteration, either a single edge or an edge and a node is added to the domain of one of more morphisms in the current set. No more than b additions can take place. This gives a worst-case running time of $r + b|A_0| + b|A_1| + ... + b|A_{m-1}|$. A_0 contains at most r morphisms, A_1 contains at most br morphisms, etc. It follows that the running time can be written as

$$r + br + b^2r + \ldots + b^m r = r \sum_{i=0}^{m} b^i$$

Next, the second root node of L is matched. Since no fewer than one root node in G has already been matched, the maximum size of the new morphism set is $b^m r(r-1)$. By the same argument, the running time after the second iteration of the while loop is

$$r \sum_{i=0}^{m} b^i + r(r-1) \sum_{i=m}^{2m} b^i$$

After the lth and final iteration of the while loop, the total running time is

$$r \sum_{i=0}^{m} b^i + r(r-1) \sum_{i=m}^{2m} b^i + \ldots + r(r-1)\ldots(r-l+1) \sum_{i=(l-1)m}^{lm} b^i$$

The AssignmentUpdate procedure is executed after each update of the set of premorphisms. Each execution verifies at most two labels for every premorphism in the set since at most two new items are added to the domain premorphism on each update. Thus it follows from Theorem 1 that the total execution time is

$$2(1+t(4s+1))\left(r\sum_{i=0}^{m}b^{i}+r(r-1)\sum_{i=m}^{2m}b^{i}+\ldots+r(r-1)\ldots(r-l+1)\sum_{i=(l-1)m}^{lm}b^{i}\right)$$

□

This is an intimidating expression, but it is a constant. In addition, in order to simplify the proof, we made some assumptions for worst-case running time that are mutually exclusive in the case where $P_L > 1$. If the while loop is executed once for each root, then no roots in L are connected, hence it is impossible for the for loop to ever execute $|E_L| = m$ times. Conversely, if the for loop executes $|E_L| = m$ times, then all roots must be connected, implying a single execution of the while loop. Therefore, for $l > 1$, the derived bound will not be reached.

We define a simple condition to be any condition in GP2's syntax [Plu12] except for edge predicates and the comparison of list, atomic and string variables. We also call the right-hand side of a rule simple if it contains no repeated list, atomic and string variables.

Theorem 3. *A conditional rooted rule schemata is applicable in constant time under the conditions of Theorem 1 and Theorem 2, provided that the schema condition and RHS are simple.*

Proof. Since we have proven that constructing a match and assignment takes constant time, we need only prove constant time execution for the remaining steps of rule application. For a simple GP condition, the only predicates are relational operators between integers, verifiable in unit time by Assumption 4; and type checking of a List variable (int, string or atom), also verifiable in constant time. The dangling condition for an injective morphism $g\colon L \to G$ can be checked by comparing the degrees of all nodes v in $L - K$ with the degree of their images $g(v)$. We assume a graph representation where node degrees can be obtained in one unit of time. This operation then takes $|L_V|$ time. Given a valid match, removing the items in $g(L - K)$ can be executed in time $|L| - |K|$. Similarly, the addition of nodes and edges takes time $|R| - |K|$. Finally, relabelling a string or list only requires redirecting the first and last pointers to a particular node or edge in G. For string concatenation, two more pointer redirections are required to combine the two strings. There are at most $|K|$ relabellings, and the worst case time is of order $1|K|$. These expressions are bounded by the size of the rule which, by Assumption 2, is constant. □

Some concessions must be made in order for rooted rule schemata application to operate in constant time. However, the overall time complexity is largely determined by the number of root nodes in both the rule and the host graph. This is to be expected since the number of root nodes available for matching will increase the nondeterminism of the matching process. Indeed, if all nodes were roots, then rooted matching would be identical to traditional graph matching. For this reason, in practice, we aim to limit the number of total root nodes. As we demonstrate in the next chapter, one root node per rule is sufficient for a simple and powerful graph program. Additionally, bounded node degree is often satisfied in practice. For example, traffic networks, digital circuits and social networks usually have an upper bound on the number of edges attached to nodes.

5.4 Case Study: Colouring Programs

Graph colouring is among the most common and important problems in graph theory. This chapter explores graph colouring in GP2. First, a non-rooted and a rooted program are compared. Both programs test graphs for 2-colourability: if the input graph G is 2-colourable, or bipartite, a possible 2-colouring of G is returned, otherwise G is returned uncoloured. Second, a rooted GP program that colours a graph is discussed.

For the rest of this paper, we use the term "rooted graph" to mean a connected graph with a single root.

Lemma 2. *Let there be an algorithm that labels nodes of a connected, undirected graph G by assigning each node a label from the set $\{0, 1\}$ as follows. The algorithm first assigns an arbitrary node the label 0, after which it repeats the following procedure until all nodes have a label: nondeterministically find an unlabelled node connected to an labelled node v, and label it with the opposite integer to v.*

Then the following statement holds: G is not bipartite if and only if, at any point in the algorithm, two connected nodes have the same label.

Proof. If G is not bipartite then, by definition, there is no way to assign integers to nodes without labelling two connected nodes with the same integer.

We prove the other direction by the contrapositive: we assume G is bipartite, and we show that the algorithm never assigns the same label to two connected nodes. Let v be the initial node of the algorithm. v is given the label 0. We prove, by induction, that no neighbours of v can be given the label 0. This is trivially true if v has one neighbour. If v has two neighbours, one of them can be labelled 0 only if it is connected to v's other neighbour. Hence G is a 3-cycle and not bipartite, a contradiction.

Now assume that if v has fewer than n neighbours, the algorithm cannot assign the label 0 to any of them. Let v have n neighbours $w_1, \dots, w_n$. By the induction hypothesis, without loss of generality, none of $w_1, \dots, w_{n-1}$ will be given the label 0, so we assume they have all been assigned the label 1. If w_n is not reachable from any of $w_1, \dots, w_{n-1}$ except through v, then w_n cannot be labelled 0. So assume there is a path from one of v's neighbours, say w_1, to w_n, that does not contain v. w_1 has the label 1, so for w_n to be given the label 0 there must be an odd number of edges from for w_1 to w_n. It follows that G contains a cycle of odd-length consisting of the two-edge path w_n, v, w_1 and the odd-length path from w_1 to w_n. Therefore G is not bipartite, a contradiction, and by induction, no neighbours of v can be given the same label as it.

This argument can be extended to every other node in the graph, hence a node can never be assigned the same colour as one of its neighbours.

□

5.4.1 Non-rooted 2-colouring

A GP program is given that will either find a 2-colouring of a connected graph, where the two colours are represented by the integers 0 and 1, or return the uncoloured input graph if no such colouring exists. The program is shown in Figure 5.5. Rule schemata with bidirectional

edges represent a set of two distinct rule schemata with unidirectional edges such that the edge direction is the same on both sides of the rule. The name of a bidirectional rule in a program or in the text refers to the nondeterministic choice of one of the two unidirectional rules it represents. This is only a notational convenience used for brevity and clarity; GP2 does not allow bidirectional rule schemata. This convention will be used throughout the paper.

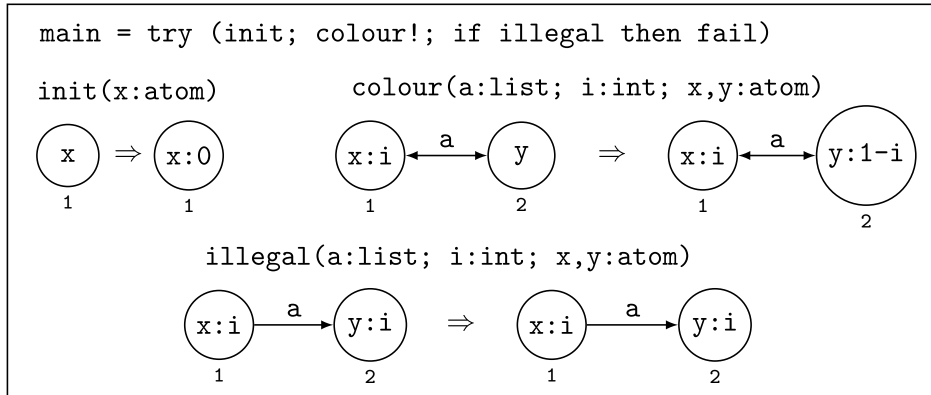


Figure 5.5: The GP program 2-colouring

The program exploits the semantics of the `try` statement. It branches to the `then` clause if the input graph is bipartite, which will maintain the graph obtained from the rule applications in the predicate subprogram: precisely the rules that colour the graph. On the other hand, if the graph is not bipartite, the program takes the `else` branch, which reverts back to the initial uncoloured input graph as required, without the need to remove colours from nodes with further rule applications. This is described in further detail in the following proposition.

Proposition 2. *Given a connected graph G with atomic-labelled nodes as input,* `2colouring` *terminates and is correct. That is to say, it returns a 2-colouring of G if G is bipartite, otherwise it returns G unchanged.*

Proof. Using the rule `init`, the program first marks an arbitrary node of G and gives it the colour 0. Then the loop `colour!` nondeterministically finds uncoloured nodes that are connected to coloured nodes and colours them with the contrasting colour. Since the rule `colour` decreases the number of uncoloured nodes, the macro `colouring` will terminate, precisely when each node has a colour. At this point, the `if` statement uses `illegal` to check if there exist two connected nodes with the same colour. If this is true, failure is triggered which, by the semantics of `try`, causes the original uncoloured graph to be returned. Otherwise, the `if` statement skips through the else clause. Control passes to the `try` statement which branches to the then clause, performing a `skip` operation on the modified graph. There are no more rules to apply, so the current graph G is returned, which has a 2-colouring due to the previous application of `colour!`. This behaviour is correct by Lemma 2. □

5.4.2 Rooted 2-colouring

A rooted version of the 2-colouring program `2colouringR` is presented in Figure 5.6. Root nodes have a thick border.

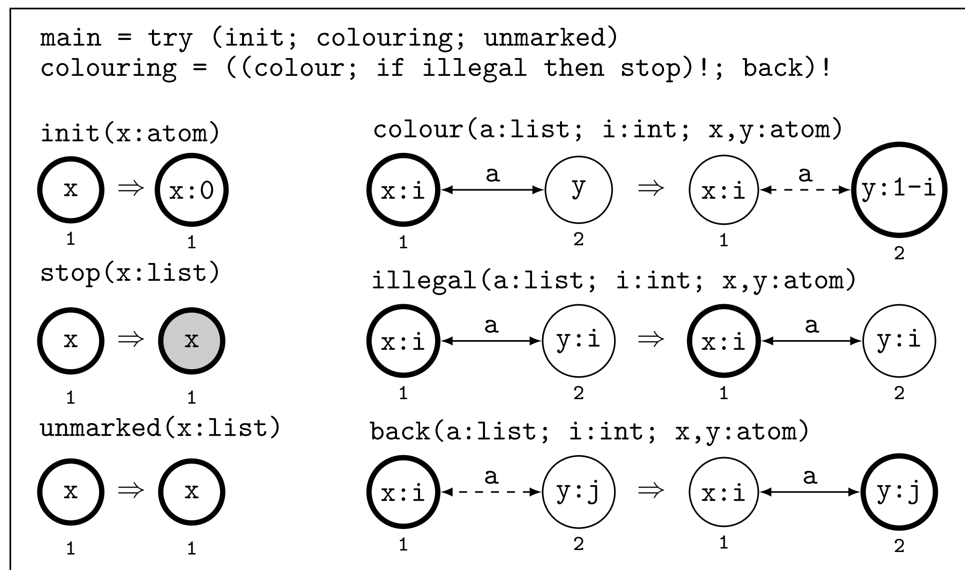


Figure 5.6: The GP Program `2colouringR`

It is clear to see that this program is more complicated than the non-rooted version. It has nine rules, over twice the number of rules of the non-rooted program. Due to the locality of left-connected rooted rules, operations can only take place on nodes directly connected to the root. Therefore rules are required to change the root node in the host graph, otherwise it is impossible to operate on all nodes of the graph. The process of changing the root node in a rule is called moving the root or making a node the root. Not only does this introduce some overhead in frequently moving the root, but care must be taken to ensure that the program terminates. Without a marker to signify which nodes or edges have been visited, it is possible for the root to, for example, move back and forth indefinitely between two coloured nodes.

Lemma 3. *Let v be the unique root node of a connected, atomic-labelled graph G with no marked edges. Upon executing `2colouringR` on G, after the application of the rule `init`, the following property is an invariant of the program: there is a path (Definition 5) of marked edges from v to the current root node. Every node on the path has a colour. There are no other marked edges.*

Proof. The property trivially holds immediately after the application of `init`. The only additional rules we need to consider are `colour` and `back` as the remaining rules do not modify the graph in any way except for marking a node. Let w be the current root node. Then there is an path of marked edges from v to w consisting of only coloured nodes, with no other marked edges in the graph. After an application of `colour`, this path has been extended by a single marked edge connecting w and one of its neighbours w'. Since w' has a colour and is the new

root node, and since `colour` creates no additional marked edges, the property still holds. A similar argument shows that `back` also preserves the property. □

Proposition 3. *Given an atomic-labelled rooted graph G as input, 2colouringR terminates and is correct. That is to say, it returns a 2-colouring of G if G is bipartite, otherwise it returns the original graph.*

Proof. To prove termination, we need only show the macro `colouring` terminates. The rest of the program terminates as `init` and `unmarked` are single rule applications. First, note that if `illegal` is ever applicable, the rule `stop` will be applied. No further rules in the macro `colouring` can be applied as no rules have a marked node on their left-hand sides. Hence `colouring` terminates, and the application of `unmarked` fails. So we can assume that the `if` statement has no effect. Every execution of the nested loop `colouring` contains only finitely many applications of `colour` as `colour` decreases the number of uncoloured nodes whereas `back` does not change this number. This implies finitely many applications of `back` since each `colour` application adds precisely one marked edge, while `back` removes precisely one marked edge.

To prove correctness, we first show that every node in the graph is coloured if the macro `stop` is never reached. In this case, the if statement will skip every time, so the inner loop reduces to {`colour!; back`}`!`. Assume that there exists an uncoloured node v in G when this loop terminates. v cannot be the root node, since neither `colour` nor `back` makes an uncoloured node the root. v cannot be adjacent to a coloured node w as w must have been the root at some stage in the computation, and the rule `colour` would have matched for v and w, either when w was first made the root or when it was made the root from an application of `back`. Therefore v is connected to only uncoloured nodes. We can inductively apply this argument to conclude that the whole graph is uncoloured. This is a contradiction since the rule `init` colours the initial root node.

By Lemma 2, if G is bipartite, no two connected nodes will have the same colour at any point. It follows that the macro `stop` is never reached, thus no node is marked and each node is given a colour. At this point, `colour` is no longer applicable, so `back` is applied as long as possible. By Lemma 3, this will remove all marked edges. The loop breaks, and the current graph is a valid 2-colouring of G with no marked nodes or edges. `unmarked` is applicable, so the `then` branch of the `try` statement is taken, and the current graph is returned. If G is not bipartite, then at some point `stop` will be applied, which marks the root node. No more rules in `colouring` are applicable, so control passes to the `try` statement. As the root node is marked, `unmarked` doesn't match, hence the `else` branch is taken which returns the original input graph. □

A nice feature of the rooted program is that it checks whether the root node has the same colour as one of its neighbours immediately after it has been coloured. This means that it could terminate without visiting all nodes in a non-bipartite input graph.

5.4.3 Complexity Comparison

First we consider the worst case number of rule applications, including rules such as `illegal` which do not modify the host graph, used by both versions of the program.

Assume a connected input graph G with n nodes. `2colouring` requires $n+1$ rule applications: one application of `init`, $n-1$ applications of `colour` and one application of `illegal`. `2colouringR` is not as straightforward. In the worst case, G is bipartite. Then it takes one application of `init` and $n-1$ rule applications of `colour` to colour the whole graph, each followed by an application of `illegal`. There are also $n-1$ rule applications of `back`, one for each marked edge created by an application of `colour`. This is a total of $3n-2$ rule applications. However, `2colouringR` could terminate early on a non-bipartite input graph, whereas `2colouring` will have n rule applications on any input graph.

With respect to the number of rule applications, both programs are linear in the number of nodes of the host graph. However, the differences become more significant when we consider complexity of rule application. As we wish to highlight the contrast between rooted programs and non-rooted programs, we disregard the construction of assignments, restructuring, and relabelling, which do not depend on the presence or absence of root nodes.

The most frequently occurring rule in `2colouring` is `colour`. This is a non-rooted rule, but it only needs to match two nodes with a connecting edge. Assuming a non-naive matching algorithm, it is no worse than $O(n)$. A linear number of rule applications gives a total time complexity of $O(n^2)$. On the other hand, `2colouringR` satisfies the assumptions of Section 5.2: the rules are left-connected, so all nodes are reachable from some root, and the host graph is assumed to be connected. In addition, some of the conditions are already met. Both the left-hand side of rules and the host graph have exactly one root. Let G have a bounded degree b. Then, by Theorem 1, constructing the premorphism takes at most time b. This is a constant, so the total complexity is $O(n)$, a significant improvement over the quadratic complexity of `2colouring`.

5.4.4 General Colouring

Figure 5.7 shows a rooted GP program that produces a valid colouring of its integer-labelled connected input graph. It is not guaranteed to construct a minimal colouring. Colours are represented by positive integers appended to the labels of each node.

A naive program would assign a new colour to each successive node in a path, a method which would almost never produce a minimal colouring. The given program limits the introduction of new colours to the graph. Assuming an input graph with more than one node, the first application of the rules `init` and `colour1` introduce the colours 1 and 2 respectively. Larger colours are introduced only if two adjacent nodes have the same colour. The colouring rules are applied deterministically since it is never possible for both of them to match in the same stage of computation.

Much like `2colouringR`, marked edges ensure the program terminates. In particular, this program has the same invariant as `2colouringR` (see Lemma 3), so there will be no marked edges when the program terminates. The program is also correct; that is to say, it always produces a valid colouring. Whenever a node has its colour changed, it becomes the root node. If the colour change caused the root node to have the same colour as one of its neighbours, the rule `correct` will increment its colour until it no longer conflicts with its neighbours.

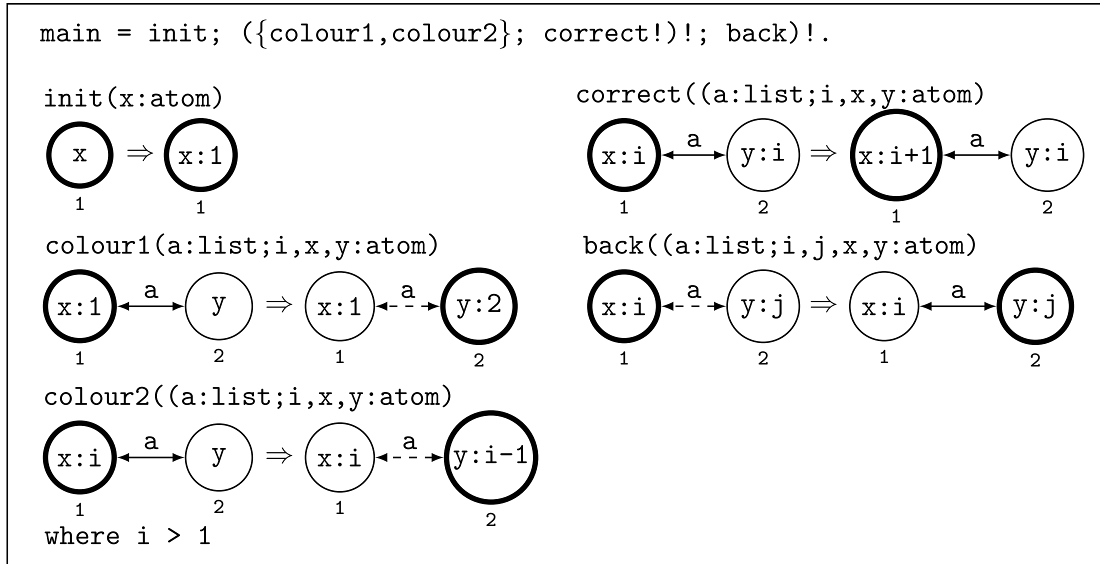


Figure 5.7: A rooted GP program for general graph colouring

5.5 Cycle Graphs

Rooted rules are local by their very nature; this is what makes them so efficient. As a consequence, using them is not the simplest way to program algorithms that require a computation on all nodes or edges of a graph. The previous section makes this evident. `2colouring` is remarkably straightforward. Only three rules are required. The program is both easy to write and to understand. Limiting ourselves to only rooted rules makes a program that performs the same computation far harder to construct. Ensuring that the program is correct requires elaborate management of marked nodes and edges, as well as a complicated sequencing of the rule schemata. Speed is gained at the expense of GP's ideology of intuitive, easily-writable graph programs.

At the start of this chapter we showed an application of rooted graph rules to verify pointer structures through the use of graph reduction system. Rooted rules are ideal for reduction-based graph programs as they are not only highly efficient, but no more complicated than a typical non-rooted graph program in this context. We illustrate this with two examples involving cycle graphs.

Definition 15. A cycle graph is a graph in which every node and edge is part of a cycle, and where each node has exactly one outgoing edge and exactly one incoming edge.

The program of Figure 5.8 is an adaptation of the graph reduction system for cyclic lists of Figure 5.1. It tests whether the atomic-labelled input graph is a cycle graph. In the original system [DP06b], the two reduction rules are `reduce` and `finish`, and the resulting graph is tested for isomorphism against a single node with a loop. This is easily replicated in GP by a third rule `accept`, which has a condition to ensure that the loop is the only edge incident to the node.

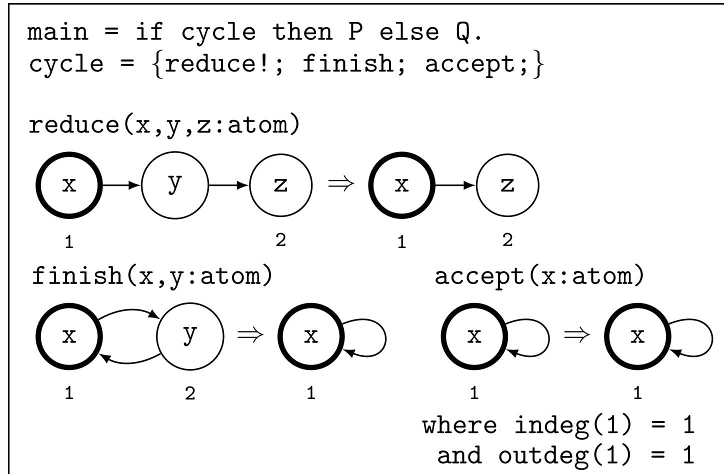


Figure 5.8: A macro `cycle` to check whether a rooted graph is a cycle or not.

It is easy to see that `reduce!; finish` will transform all cycle graphs to a single node with a looping edge. `reduce!` repeatedly removes an intermediate node from the cycle until the cycle graph with two nodes remains. This is matched by `finish` to give a single node with a loop as required. Conversely, if the input graph is not a cycle graph, then one of its nodes must have an additional incident edge e. Then the macro will fail in one of three ways: (1) an application of `reduce` fails as e is incident to the match of the y-labelled node, and removing it would violate the dangling condition. (2) `finish` fails for the same reason. (3) `accept` fails as the node has at least one more incident edge, violating the condition.

In this particular program, the presence of rooted nodes gives little to no efficiency gain. In a cycle graph, any node will take part in a match of `reduce` or `finish`, so no search is required and standard subgraph matching will perform just as well as rooted matching. There may, however, be a slight improvement for non-cyclic graphs. If the input graph has a cycle subgraph, then

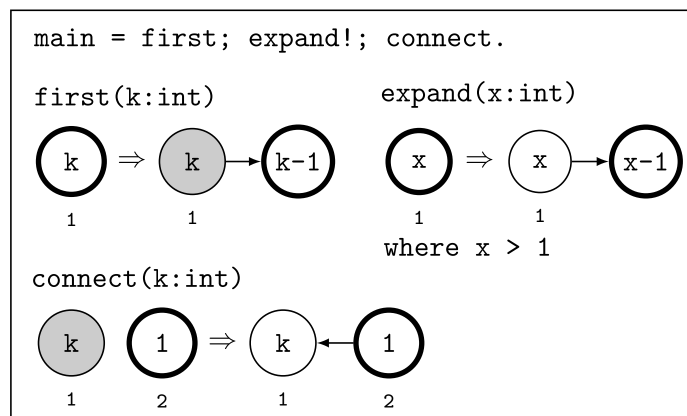


Figure 5.9: A GP program that constructs a cycle of length k.

a non-rooted version would have to apply all the reduction rules to this subgraph. This is not necessarily the case for the rooted program: if the root node is not part of the cyclic subgraph then the macro could terminate with fewer applications of the reduction rules.

Contrary to the 2-colouring programs, this rooted program is very simple and no more complicated than a non-rooted program would be. Our next example shows that rooted programs can be even simpler than a non-rooted version. Figure 5.9 shows a program that, given an input graph of a single root node labelled with an integer $k > 1$, constructs a cycle graph of k nodes.

This program is correct because of rooted matching: the cycle can only be extended at the current root node. This ensures that each node has exactly one incoming edge and one outgoing edge. Of course, this can be achieved without root nodes. The roots act as markers, so one can replace the bolded node edges, except for the LHS of `first`, with a tag by, say, appending the string "current" to the integer label. The main drawback to this approach is that there is no rooted matching. This means that the machine will search for the tagged node at each rule application, instead of the root node being automatically matched. Hence, as k grows, the non-rooted program will perform slower in comparison to the rooted program. In addition, it is far more natural to use roots as tags rather than the more artificial method of appending a marker to the node's label.

5.6 Summary

Rooted graph programs enable an efficient implementation of subgraph matching, widely known as the biggest challenge in implementing graph transformation rules. Rooted rules only match in the neighbourhood of graph roots, making the search for a match faster due to the locality of the search. In fact, under reasonable conditions, our algorithms provide constant time rule application.

The double pushout approach has been cleanly extended to incorporate rooted graphs and rooted rules, and this extension has been applied to graph programs in GP2. We have shown an algorithm which matches a large class of rooted conditional rule schemata in constant time, provided that host graphs have bounded node degrees. The case study demonstrates that algorithms of practical importance, such as graph colouring, can be implemented with rooted GP2 programs whose time complexity is as good as that of programs in imperative languages.

This efficiency gain comes at a cost, as rooted programs are more challenging to construct. The root node may need to be moved around the host graph, which causes a growth in the number of rule applications and necessitates the use of perhaps convoluted techniques to ensure program correctness and termination. However, we have seen in our cycle graph programs that this is not always the case. In addition, users will have the choice of using rooted rules, so they will not be mandatory. The aim is to allow programs that contain both rooted and non-rooted rules, although the ramifications of this are as yet unclear.

6 Research Proposal

The overall goal of my research is to develop an improved and complete implementation of GP, perhaps one that can be made available for public use. This report has covered three main areas concerning this goal:

1. The changes introduced in the new version of GP.
2. The searchplan generation strategies of three graph programming systems.
3. Rooted graph transformation in the context of GP.

I outline a plan of action based largely on these areas.

6.1 Rewrite the GP Compiler

Due to the language changes detailed in Chapter 3 and the proposed extension to GP introduced in Chapter 5, GP's compiler needs to be modified to manage the changes in the syntax and semantics of the language. I plan to write a new compiler from scratch instead of updating the existing compiler for several reasons:

- The current GP compiler is poorly documented. Therefore it would likely take longer for me to untangle, understand and update the code than it would for me to start anew.
- Writing a new compiler will enhance my programming skills, my compiler-writing skills, and it will give me a far better understanding of the inner workings of GP system than I would have gained from editing the existing compiler.
- The code will be commented and documented as it is being written, making it more readable and comprehensible for both myself and for anybody that may work on GP in the future.

6.2 Design and Implement a Searchplan Generation Algorithm

Chapter 4 covered different approaches to subgraph matching implementation. I aim to design a dynamic searchplan generation algorithm similar to those of PROGRES and GrGEN, as they are both very efficient. Moreover, as GP does not have a specific application domain, its host graphs could be extremely varied in structure. Therefore a searchplan that adapts to the host graph would be more beneficial to GP than it would to a system such as PROGRES which focuses on graph use in software engineering.

The techniques used by PROGRES and GrGEN largely concern the analysis of rules and host graphs, so they should be adaptable to the GP system. I believe the best approach is to generate costs based on an explicit analysis of the host graph as GrGEN does, rather than PROGRES' estimated costs. Again, GP does not have a specific application domain in mind, therefore it is impossible to categorise a typical host graph from which heuristic costs can be generated. It is more sensible to interrogate the host graph. In addition, this should be efficient because of the complex graph data structure that enables fast graph querying.

There are two options for GP's computation engine. The first option is to maintain the York Abstract Machine. The YAM's power comes from basic bytecode operations and fast graph querying. These are the exact things that a searchplan provides since it reduces the subgraph isomorphism problem into primitive search operations. However, the YAM is also poorly documented, which might make it hard to update.

The second option is to translate GP programs straight to C code. The language C is appropriate as its low-level operations should make executing graph programs very efficient. In addition, on a personal level, it is the programming language I am most familiar with. This option will also create a smoother and faster compilation process as there is no overhead involved in compiling to YAM bytecode.

Either way, the searchplan design should take this choice into consideration.

6.3 Add Tools and Features

Currently, the GP system only offers the ability to write and run graph programs. This is, of course, the main goal of graph programming systems, thus it is something that should be prioritised over anything else. However, many external graph programming systems offer additional tools to assist the user. For instance, AGG has critical pair analysis that is used for detecting graph grammar confluence and for graph grammar parsing. A second static analysis option for graph programs is termination. It would be very nice for GP to support such tools. Not only would they offer support for a GP programmer, but in some cases they could increase efficiency. For example, if a graph program is proven to be confluent, then backtracking can be disabled and execution would be faster.

Another possibility is to adopt PROGRES' optional node matching. This would allow more flexible GP programs to be written. However, this would have to be justified both by theory and by examples of useful applications of optional node matching. Finally, another new feature to take into consideration is the explicit construction of the initial match for a particular program and host graph. This would give a programmer more control over their programs, and it would increase the speed of program execution since at least one match will not have to be computed by the machine.

GP's graphical editor also needs to be polished or even rebuilt. The current editor, while functional, has many minor bugs, and it was written before the design of GP2 was made. The new editor should support rooted programs and generate appropriate text files for the parser.

6.4 Further Work on Rooted Graph Programs

There are several aspects of rooted graph programs which have yet to be explored.

The first aspect is programs using rules that either add or remove root nodes. Such rules could, for example, be used in a program that creates a root node for a computation on a small area of a graph, then removes the root node once that computation is finished, allowing a subsequent global computation that doesn't have the drawback of the complex constructs that arise from moving the root node around a graph. Graph programs that can add roots are not guaranteed to run in constant time as the host graph could contain an arbitrary number of roots. However, in practice, they could be potentially be extremely useful.

The second aspect is the combination of rooted rules and non-rooted rules in a single GP2 program. So far we have ensured that every rule in a rooted graph program has at least one root, thus keeping non-rooted programs and rooted programs separate. For rooted programs, this guarantees efficient subgraph matching at the expense of added program complexity. It is the other way around for non-rooted programs: they are less complicated but also less efficient. Therefore, if both rules could be used in harmony, the program would have the best of both worlds. Intuitively, combining the two classes of rules appears straightforward, but there could be complications that arise after a closer examination.

The third aspect is to look further into applications for graph programs with multiple roots. One use of multiple roots is to apply a single graph algorithm to a graph with a number of disconnected components where each component has a single root. For example, `2colouringR` (Figure 5.6) can be modified to be applicable on both connected and disconnected input graphs. However, this is an extension of a single-rooted application rather than a stand-alone purpose for multiple roots. It is unclear how multiple roots can be used for efficient graph processing, but it is an interesting research topic.

The fourth aspect is to write more examples of practical rooted graph programs to further showcase their usefulness. The 2-colouring program in Figure 5.6 is fairly small, elegant, and runs in linear time. There are likely to be rooted graph programs composed largely, or even entirely, of fast rule schemata that perform useful functions such as testing a host graph for acyclicity or connectedness, or computing a minimum spanning tree.

The final aspect is to consider alternate conditions for fast rooted graph matching. For instance, in [DP06b], only the outdegree of nodes in the host graph are bounded, with the caveat that each node in the left-hand graph of a rooted rule must be reachable from the root by a directed path. Under this framework, matching is even faster if each distinct edge with the same source node has a distinct label.

6.5 Schedule

To conclude this report, I propose a schedule for the rest of my PhD. This is pictorially represented as a Gantt chart in Figure 6.1 which includes the course milestones. Shaded bars represent writing activities which will run in parallel with my main work. I give a brief description of each activity.

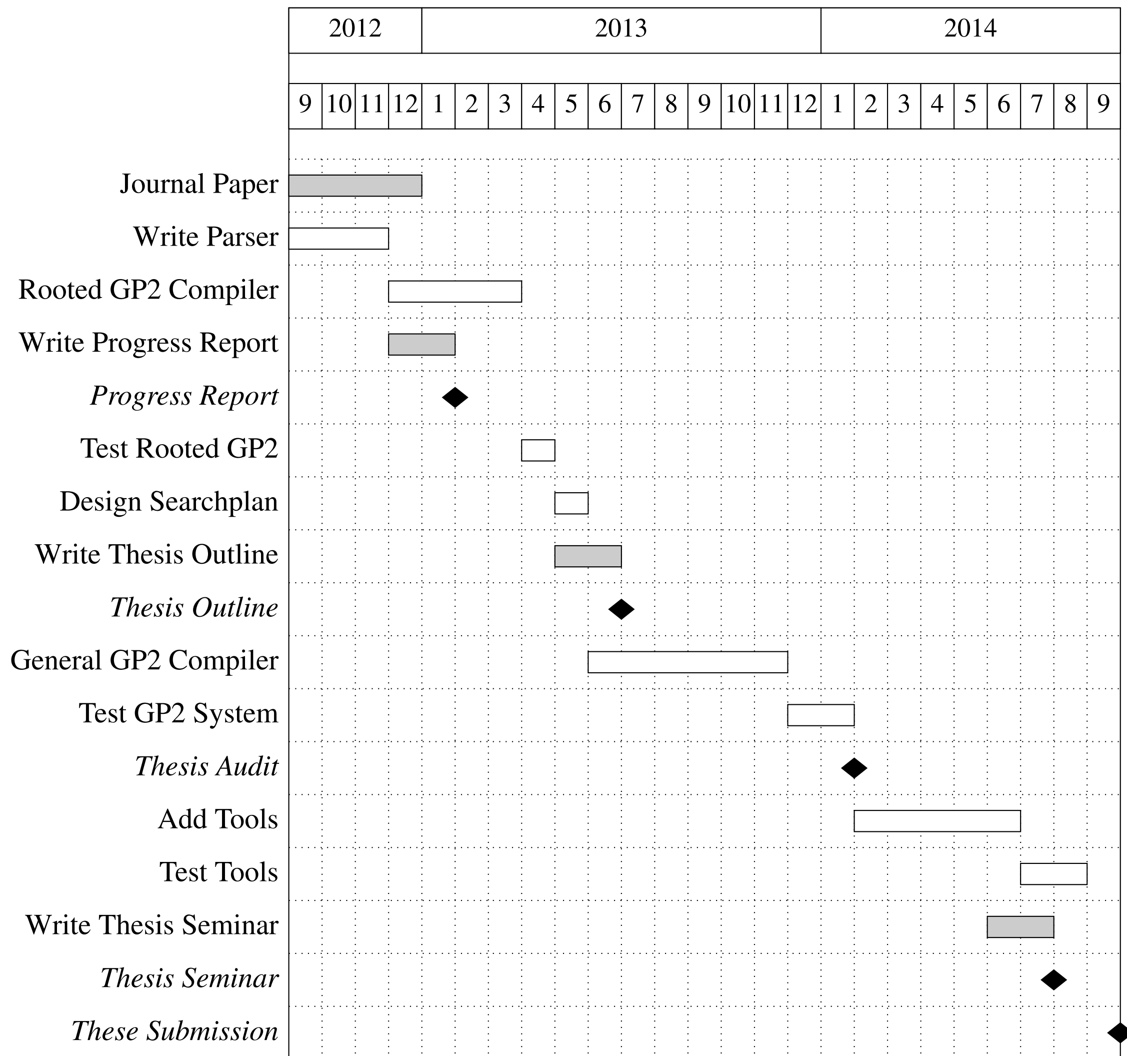


Figure 6.1: My work schedule.

- **Journal Paper.** The work in Chapter 5 should be extended to a journal paper, as the conference proceedings had a strict page limit which severely restricted the content of the paper. In addition, examining the issues outlined in Section 6.4 will provide more insight into rooted graph programs which will not only add to the journal version of the paper, but aid the development of GP2.

- **Write Parser.** The parser will transform the *.gxl and *.gpx files produced by the graphical editor (see Section 3.5) into an abstract syntax tree for the compiler. The syntax of GXL and GPX in the GP system will require slight modification to support rooted graphs and to reflect changes in GP's syntax.

- **Rooted GP2 Compiler/Test Rooted GP2.** Before reworking the entire GP system, I will write a compiler for a subset of GP2 featuring only rooted graph programs. This will require the design of a suitable graph data structure to accomodate root nodes. The compiler will translate GP2 programs to C, as rooted graph matching is more deterministic than standard graph matching and it will enable the theoretical results of Chapter 5 to be tested without too much delay. Furthermore, this work should provide some insight on whether it is better to maintain the YAM or to translate all GP2 programs directly to C. Tests will involve comparison with graph programs in the old GP implementation and comparison with graph algorithms in conventional programming languages.

- **Design Searchplan.** The structure of the algorithm would be similar to those of GrGEN or PROGRES, but the primitive search operation and cost function need to be clearly defined. The design may need to take into account how the algorithm will use the bytecode of the YAM, which will require a detailed examination of the instruction set and the code of the YAM.

- **General GP2 compiler.** The output requirements of the compiler will be made clear when the design of the searchplan algorithm is complete. This will not only include the translation of non-rooted GP2 programs to either YAM bytecode or C, but also the integration of rooted and non-rooted rules and the searchplan generation algorithm.

- **Test GP2 System.** The system must be rigorously tested to ensure it performs correct graph transformation with respect to the language's syntax and semantics. It would also be nice to participate in a tool contest as in Section 4.4, but this may not be feasible if a graphical editor for writing GP programs does not yet exist.

- **Add Tools/Test Tools.** To make GP2 more complete as a graph programming system, I aim to implement the graphical editor as well as static analysis tools such as those described in Section 6.3.

Bibliography

[AK91] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991. URL: http://wambook.sourceforge.net/.

[Bat06] Gernot Veit Batz. *An Optimization Technique for Subgraph Matching Strategies*. Tech. rep. 2006. URL: http://www.info.uni-karlsruhe.de/papers/TR_2006_7.pdf.

[BGT91] H. Bunke, T. Glauser, and T.-H. Tran. "An Efficient Implementation of Graph Grammars Based on the RETE-Matching Algorithm". In: *Proc. Graph Grammars and Their Application to Computer Science and Biology*. Vol. 532. LNCS. Springer-Verlag, 1991, pp. 174–189.

[BP12] Christopher Bak and Detlef Plump. "Rooted Graph Programs". In: *Proceedings of the Seventh International Workshop on Graph-Based Tools (GraBaTs 2012)*. Electronic Communications of the EASST. To appear. 2012.

[BPR04] Adam Bakewell, Detlef Plump, and Colin Runciman. "Specifying Pointer Structures by Graph Reduction". In: *Applications of Graph Transformations With Industrial Relevance (AGTIVE 2003), Revised Selected and Invited Papers*. Vol. 3062. Springer-Verlag, 2004, pp. 30–44.

[Bun79] Horst Bunke. "Programmed Graph Grammars". In: *Graph-Grammars and Their Application to Computer Science and Biology*. Vol. 73. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1979, pp. 155–166.

[Cor+97] A. Corradini et al. "Algebraic Approaches to Graph Transformation. Part I: Basic Concepts and Double Pushout Approach". In: *Handbook of Graph Grammars and Computing by Graph Transformation*. Ed. by Grzegorz Rozenberg. World Scientific Publishing Co., Inc., 1997, pp. 163–245.

[Det+12] M. von Detten et al. *Story Diagrams - Syntax and Semantics*. Tech. rep. Software Engineering Group, Heinz Nixdorf Institute, 2012.

[Dod08] Mike Dodds. "Graph Transformation and Pointer Structures". PhD thesis. The University of York, 2008. URL: http://www.cs.york.ac.uk/plasma/publications/pdf/DoddsThesis.08.pdf.

[Dör95a] Heiko Dörr. "Bypass Strong V-Structures and Find an Isomorphic Labelled Subgraph in Linear Time". In: *Graph-Theoretic Concepts in Computer Science*. Ed. by W. Mayr, Ernst, Gunter Schmidt, and Gottfried Tinhofer. Vol. 903. 1995, pp. 305–318.

[Dör95b] Heiko Dörr. *Efficient Graph Rewriting and its Implementation*. Vol. 922. Springer-Verlag, 1995. DOI: 10.1007/BFb0031909.

[DP06a] Mike Dodds and Detlef Plump. "Extending C for Checking Shape Safety". In: *Proc. Graph Transformation for Verification and Concurrency (GT-VC 2005)*. Vol. 154(2). Electronic Notes in Theoretical Computer Science. Elsevier, 2006.

[DP06b] Mike Dodds and Detlef Plump. "Graph Transformation in Constant Time". In: *Proc. International Conference on Graph Transformation (ICGT 2006)*. Vol. 4178. LNCS. Springer-Verlag, 2006, pp. 367–382.

[Ehr79] Hartmut Ehrig. "Introduction to the Algebraic Theory of Graph Grammars". In: *Proc. Graph-Grammars and Their Application to Computer Science and Biology*. Vol. 73. LNCS. Springer-Verlag, 1979, pp. 1–69.

[Ehr+97] H. Ehrig et al. "Algebraic Approaches to Graph Transformation. Part II: Single Pushout Approach and Comparison with Double Pushout Approach". In: *Handbook of Graph Grammars and Computing by Graph Transformation*. Ed. by Grzegorz Rozenberg. World Scientific Publishing Co., Inc., 1997, pp. 247–312.

[Fis+00] Thorsten Fischer et al. "Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language". In: *Theory and Application of Graph Transformations (TAGT'98), Selected Papers*. Vol. 1764. Springer-Verlag, 2000, pp. 296–309.

[Gei+06] Rubino Geiß et al. "GrGen: A Fast SPO-Based Graph Rewriting Tool". In: Lecture Notes in Computer Science (2006), pp. 383 –397.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. GT48, page 202. W. H. Freeman & Co., 1990.

[Gru+05] Lars Grunske et al. "Using Graph Transformation for Practical Model-Driven Software Engineering". In: *Model-Driven Software Development*. Ed. by Sami Beydeda, Matthias Book, and Volker Gruhn. Springer-Verlag, 2005, pp. 91–117.

[HP01] Annegret Habel and Detlef Plump. "Computational Completeness of Programming Languages Based on Graph Transformation". In: *Proceedings of the 4th International Conference on Foundations of Software Science and Computation Structures*. FoSSaCS '01. Springer-Verlag, 2001, pp. 230–245.

[HP02] Annegret Habel and Detlef Plump. "Relabelling in Graph Transformation". In: *Proc. International Conference on Graph Transformation (ICGT 2002)*. Springer-Verlag, 2002, pp. 135–147.

[MP08a] Greg Manning and Detlef Plump. "The GP Programming System". In: *Proc. Graph Transformation and Visual Modelling Techniques (GT-VMT 2008)*. Vol. 10. Electronic Communications of the EASST. 2008.

[MP08b] Greg Manning and Detlef Plump. "The York Abstract Machine". In: *Proc. Graph Transformation and Visual Modelling Techniques (GT-VMT 2006)*. Vol. 211. Electronic Communications in Theoretical Computer Science. Elsevier, 2008, pp. 231–240.

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

[NS91] Manfred Nagl and Andy Schürr. "A Specification Environment for Graph Grammars". In: *Proceedings of the 4th International Workshop on Graph-Grammars and Their Application to Computer Science*. Springer-Verlag, 1991, pp. 599–609.

[Plo04] Gordon D. Plotkin. "A Structural Approach to Operational Semantics". In: *Journal of Logic and Algebraic Programming* 60-61 (2004), pp. 17–139.

[Plu05] Detlef Plump. "Confluence of Graph Transformation Revisited". In: *Processes, Terms and Cycles: Steps on the Road to Infinity: Essays Dedicated to Jan Willem Klop on the Occasion of His 60th Birthday, volume 3838 of LNCS*. Springer, 2005, pp. 280–308.

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

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

[PP12] Christopher M. Poskitt and Detlef Plump. "Hoare-Style Verification of Graph Programs". In: *Fundamenta Informaticae* 118.1-2 (2012), pp. 135–175.

[PS04] Detlef Plump and Sandra Steinert. "Towards Graph Programs for Graph Algorithms". In: *ICGT*. Vol. 3256. LNCS. Springer, 2004, pp. 128–143.

[RSV04] A. Rensink, Á. Schmidt, and D. Varró. "Model Checking Graph Transformations: A Comparison of Two Approaches". In: *International Conference on Graph Transformations (ICGT)*. Ed. by H. Ehrig et al. Vol. 3256. Lecture Notes in Computer Science. Berlin: Springer Verlag, 2004, pp. 226–241. URL: http://doc.utwente.nl/66360/.

[Rud98] Michael Rudolf. "Utilizing Constraint Satisfaction Techniques for Efficient Graph Pattern Matching". In: *6th International Workshop on Theory and Applications of Graph Transformation (TAGT)*. Springer, 1998, pp. 238–251.

[Run06] Olga Runge. *The AGG 1.5.0 Development Environment: The User Manual*. Aug. 2006. URL: http://user.cs.tu-berlin.de/~gragra/agg/AGG-ShortManual/AGG-ShortManual.html.

[Sch91] Andy Schürr. "PROGRESS: A VHL-Language Based on Graph Grammars". In: *Proceedings of the 4th International Workshop on Graph-Grammars and Their Application to Computer Science*. Springer-Verlag, 1991, pp. 641–695.

[Ski08] Steven S. Skiena. *The Algorithm Design Manual*. second. Springer-Verlag, 2008.

[Ste07] Sandra Steinert. "The Graph Programming Language GP". PhD thesis. The University Of York, 2007. URL: http://www.cs.york.ac.uk/ftpdir/reports/2007/YCST/15/YCST-2007-15.pdf.

[SWZ95] A. Schürr, A. Winter, and A. Zündorf. “Visual Programming with Graph Rewriting Systems”. In: *In 11th IEEE Symp. on Visual Languages*. IEEE Computer Society Press, 1995, pp. 326–335.

[SWZ99] Andy Schürr, Andreas Winter, and Albert Zündorf. “The PROGRES Approach: Language and Environment”. In: *Handbook of Graph Grammars and Computing by Graph Transformation*. Ed. by H. Ehrig et al. Vol. 2. World Scientific, 1999. Chap. 13, pp. 487–550.

[Tae00] Gabriele Taentzer. “AGG: A Tool Environment for Algebraic Graph Transformation”. In: *in AGTIVE, ser. Lecture Notes in Computer Science*. Springer, 2000, pp. 481–488.

[Tae03] Gabriele Taentzer. “AGG: A Graph Transformation Environment for Modeling and Validation of Software”. In: *Proc. Tool Exhibition at Formal Methods*. 2003.

[Tae+08] Gabriele Taentzer et al. “Generation of Sierpinski Triangles: A Case Study for Graph Transformation Tools”. In: *Applications of Graph Transformation with Industrial Relevance (AGTIVE 2007), Revised Selected and Invited Papers*. Vol. 5088. LNCS. Springer-Verlag, 2008, pp. 514–539.

[Win+02] Andreas Winter et al. “An Overview of the GXL Graph Exchange Language”. In: *Revised Lectures on Software Visualization, International Seminar*. Springer-Verlag, 2002, pp. 324–336.

[Z9̈3] Albert Zündorf. *A Heuristic for the Subgraph Isomorphism Problem in Executing PROGRES*. 1993. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.44.9607&rep=rep1&type=pdf.