The Semantics of Graph Programs

Detlef Plump
Department of Computer Science
The University of York, UK

Sandra Steinert
Department of Computer Science
The University of York, UK

GP (for Graph Programs) is a rule-based, nondeterministic programming language for solving graph problems at a high level of abstraction, freeing programmers from handling low-level data structures. The core of GP consists of four constructs: single-step application of a set of conditional graph-transformation rules, sequential composition, branching and iteration. We present a formal semantics for GP in the style of structural operational semantics. A special feature of our semantics is the use of *finitely failing* programs to define GP's powerful branching and iteration commands.

1 Introduction

This paper defines the semantics of GP, an experimental nondeterministic programming language for high-level problem solving in the domain of graphs. The language is based on conditional rule schemata for graph transformation (introduced in [16]) and thereby frees programmers from handling low-level data structures for graphs. The prototype implementation of GP compiles graph programs into bytecode for the York abstract machine, and comes with a graphical editor for programs and graphs [11].

GP has a simple syntax as its core contains only four commands: single-step application of a set of rule schemata, sequential composition, branching and as-long-as-possible iteration. Despite its simplicity, GP is computationally complete in that every computable function on graphs can be programmed [8]. A major goal of the GP project is the development of a practical graph-transformation language that comes with a concise formal semantics, to facilitate program verification and other formal reasoning on programs. Also, a formal semantics provides implementors with a rigorous definition of the language that does not depend on a compiler or machine.

To define the meaning of GP programs, we adopt Plotkin's method of structural operational semantics [14]. This approach is well established for imperative programming languages [13] but is novel in the field of graph transformation. In brief, the method consists in devising inference rules which inductively define the effect of commands on program states. Whereas a classic state consists of the values of all program variables at a certain point in time, the analogue for graph transformation is the graph on which the rules of a program operate.

As GP is nondeterministic, our semantics assigns to a program P and an input graph G *all* graphs that can result from executing P on G. A special feature of the semantics is the use of failing computations to define powerful branching and iteration constructs. (Failure occurs when a set of rule schemata to be executed is not applicable to the current graph.) While the conditions of branching commands in traditional programming languages are boolean expressions, GP uses arbitrary programs as conditions. The evaluation of a condition C succeeds if there *exists* an execution of C on the current graph that produces a graph. On the other hand, the evaluation of C is unsuccessful if all executions of C on the current graph result in failure. In this case C *finitely fails* on the current graph.

In logic programming, finite failure (of SLD resolution) is used to define negation [4]. In the case of GP, it allows to "hide" destructive executions of the condition C of a statement `if` C `then` P `else` Q. This is because after evaluating C, the resulting graph is discarded and either P or Q is executed on the

I. Mackie and A. Martins Moreira (Eds.): Tenth International
Workshop on Rule-Based Programming (RULE 2009)
EPTCS 21, 2010, pp. 27–38, doi:10.4204/EPTCS.21.3

graph with which the branching statement was entered. Finite failure also allows to elegantly lift the application of as-long-as-possible iteration from sets of rule schemata (as in [16]) to arbitrary programs: the body of a loop can no longer be applied if it finitely fails on the current graph.

Control constructs which allow programmers to write "strategies" for applying rewrite rules have long been present in term-rewriting languages such as Elan [2] and Stratego [3]. These languages allow recursive definitions of strategies whereas GP is based on a small set of built-in, non-recursive constructs. (See [19] for an extension of GP with recursive procedures.)

Another difference between GP and languages such as Elan and Stratego is that strategies in the latter languages rely on the structure of the objects that they manipulate, that is, on the tree structure of terms. In both languages, term-rewrite rules are applied at the root of a term so that traversal operations are needed to apply rules and strategies deep inside terms. In contrast, the semantics of GP's control constructs does not depend on the structure of graphs and is completely orthogonal to the semantics of rule schemata. This provides a clear separation of concerns between rules and the control of rules, making it easy to adapt GP's semantics to different formats of rules or graphs.[1]

The contributions of this paper can be summarised as follows:

- A graph-transformation language with *simple* syntax and semantics, facilitating understanding by programmers and formal reasoning on programs. Our experience so far is that very often short and easy to understand programs can be written to solve problems on graphs (see [15] for various small case studies).
- The first formal operational semantics for a graph-transformation language (to the best of our knowledge). Well-known languages such as AGG [6], Fujaba [12] and GrGen [7] have no formal semantics. The only graph-transformation language with a complete formal semantics that we are aware of is PROGRES [18]. Its semantics, given by Schürr in his dissertation [17], translates programs into control-flow diagrams and consists of more than 300 rules (including the definition of the static semantics) .
- A powerful branching construct based on the concept of finite failure, allowing to conveniently express complex destructive tests on input graphs. In addition, finite failure enables an elegant definition of as-long-as-possible iteration. These definitions do not depend on the structure of graphs and can be used for string- or term-based rewriting languages, too.

The rest of this paper is structured as follows. The next section reviews the graph-transformation formalism underlying GP, the so-called double-pushout approach with relabelling. Section 3 introduces conditional rule schemata as the building blocks of GP programs. In Section 4, we discuss an example program for graph colouring and define the abstract syntax of graph programs. Section 5 presents our formal semantics of GP in the style of structural operational semantics. In Section 6, we conclude and mention some topics for future work.

2 Graph Transformation

We briefly review the model of graph transformation underlying GP, the double-pushout approach with relabelling [9]. Our presentation is tailored to GP in that we consider graphs over a fixed label alphabet, and rules in which only the interface may contain unlabelled nodes.

GP programs operate on graphs labelled with sequences of integers and strings. (The reason for using sequences will become clear in Section 4.) To formalise this, let $\mathbb{Z}$ be the set of integers and Char be a

[1] In the extreme, one could even replace the underlying formalism of graph-transformation with some other rule-based framework, such as string or term rewriting.

finite set of characters—we may think of Char as the characters that can be typed on a keyboard. We fix the label alphabet $\mathscr{L} = (\mathbb{Z} \cup \text{Char}^*)^+$ consisting of all nonempty sequences made up from integers and character strings.

A *partially labelled graph* over $\mathscr{L}$ (or *graph* for short) is a system $G = (V_G, E_G, s_G, t_G, l_G, m_G)$, where V_G and E_G are finite sets of *nodes* (or *vertices*) and *edges*, $s_G, t_G \colon E_G \to V_G$ are the *source* and *target* functions for edges, $l_G \colon V_G \to \mathscr{L}$ is the partial node labelling function and $m_G \colon E_G \to \mathscr{L}$ is the (total) edge labelling function. Given a node v, we write $l_G(v) = \perp$ to express that $l_G(v)$ is undefined. Graph G is *totally labelled* if l_G is a total function.

The set of all totally labelled graphs over $\mathscr{L}$ is denoted by $\mathscr{G}$. GP programs operate on the graphs in $\mathscr{G}$, unlabelled nodes occur only in the interfaces of rules (see below) and are necessary in the double-pushout approach to relabel nodes. There is no need to relabel edges as they can always be deleted and reinserted with changed labels.

A *graph morphism* $g \colon G \to H$ between graphs G and H consists of two functions $g_V \colon V_G \to V_H$ and $g_E \colon E_G \to E_H$ that preserve sources, targets and labels (that is, $s_H \circ g_E = g_V \circ s_G$, $t_H \circ g_E = g_V \circ t_G$, $m_H \circ g_E = m_G$, and $l_H(g(v)) = l_G(v)$ for all v such that $l_G(v) \neq \perp$). Morphism g is an *inclusion* if $g(x) = x$ for all nodes and edges x. It is *injective* if g_V and g_E are injective.

A *rule* $r = (L \leftarrow K \rightarrow R)$ consists of two inclusions $K \to L$ and $K \to R$ where L and R are totally labelled graphs. Graph K is the *interface* of r. Intuitively, an application of r to a graph will remove the items in $L - K$, preserve K, add the items in $R - K$, and relabel the unlabelled nodes in K. Given a graph G in $\mathscr{G}$, an injective graph morphism $g \colon L \to G$ is a *match* for r if it satisfies the *dangling condition*: no node in $g(L) - g(K)$ is incident to an edge in $G - g(L)$. In this case G *directly derives* the graph H in $\mathscr{G}$ that is constructed from G as follows:[2]

1. Remove all nodes and edges in $g(L) - g(K)$.
2. Add disjointly all nodes and edges from $R - K$, keeping their labels. For $e \in E_R - E_K$, $s_H(e)$ is $s_R(e)$ if $s_R(e) \in V_R - V_K$, otherwise $g_V(s_R(e))$. Targets are defined analogously.
3. For each node v in K with $l_K(v) = \perp$, $l_H(g_V(v))$ becomes $l_R(v)$.

We write $G \Rightarrow_{r,g} H$ (or just $G \Rightarrow_r H$) if G directly derives H as above.

Figure 1 shows an example of a direct derivation. The rule in the upper row is applied to the left graph of the lower row, resulting in the right graph of the lower row. For simplicity, we do not depict edge labels and assume that they are all the same. The node identifiers 1 and 2 in the rule specify the inclusions of the interface. The middle graph of the lower row is an intermediate result (omitted in the above construction). This diagram represents a double-pushout in the category of partially labelled graphs over $\mathscr{L}$.

To define conditional rules, we equip rules with predicates that restrict sets of matches. A *conditional rule* $q = (r, P)$ consists of a rule r and a predicate P on graph morphisms. Given totally labelled graphs G, H and a match $g \colon L \to G$ for q, we write $G \Rightarrow_{q,g} H$ (or just $G \Rightarrow_q H$) if $P(g)$ holds and $G \Rightarrow_{r,g} H$. For a set of conditional rules $\mathscr{R}$, we write $G \Rightarrow_{\mathscr{R}} H$ if there is some q in $\mathscr{R}$ such that $G \Rightarrow_q H$.

3 Conditional Rule Schemata

A GP program is essentially a list of declarations of conditional rule schemata together with a command sequence for controlling the application of the schemata. Rule schemata generalise rules in that labels can contain expressions over parameters of type integer or string. In this section, we give an abstract

[2] See [9] for an equivalent definition by graph pushouts.

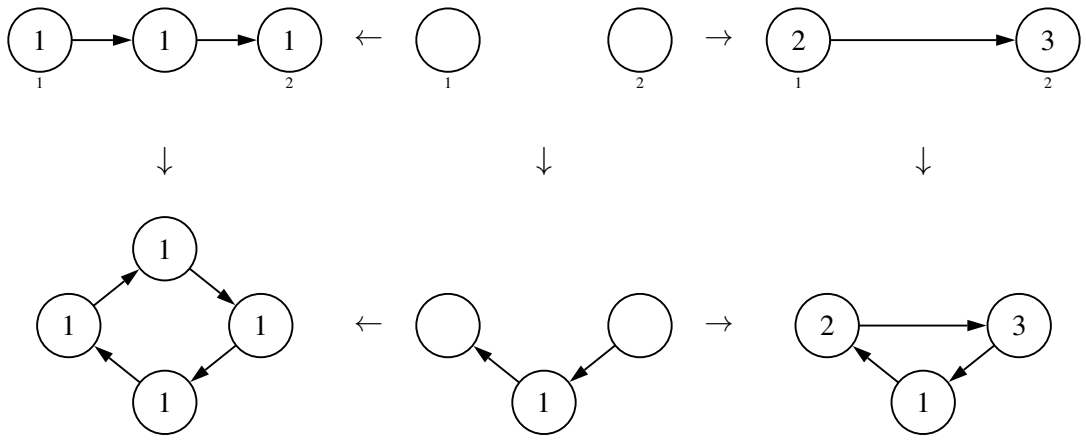

Figure 1: A direct derivation

syntax for the textual components of conditional rule schemata and interpret them as sets of conditional rules.

Figure 2 shows an example for the declaration of a conditional rule schema. It consists of the identifier `bridge` followed by the declaration of formal parameters, the left and right graphs of the schema which are labelled with expressions over the parameters, the node identifiers `1`, `2`, `3` determining the interface of the schema, and the keyword `where` followed by the condition.

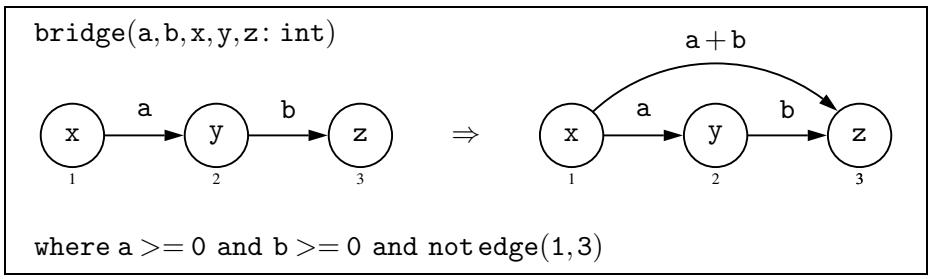

Figure 2: A conditional rule schema

In the GP programming system [11], rule schemata are constructed with a graphical editor. Figure 3 gives a grammar in Extended Backus-Naur Form for node and edge labels in the left and right graph of a rule schema (categories LeftLabel and RightLabel).[3] Labels can be sequences of expressions separated by underscores, as will be demonstrated by Example 1 in Section 4. We require that labels in the left graph must be simple expressions because their values at execution time are determined by graph matching. All variable identifiers in the right graph must also occur in the left graph. Every expression in category Exp has type `int` or `string`, where arithmetical operators expect arguments of type `int` and the type of variable identifiers is determined by their declarations.

The condition of a rule schema is a boolean expression built from expressions of category Exp and the special predicate `edge`, see Figure 4. Again, all variable identifiers occurring in the condition must

[3]The grammars in Figure 3 and Figure 4 are ambiguous, we use parentheses to disambiguate expressions where necessary.

LeftLabel	::=	SimpleExp ['_' LeftLabel]
RightLabel	::=	Exp ['_' RightLabel]
SimpleExp	::=	['-'] Num | String | VarId
Exp	::=	SimpleExp | Exp ArithOp Exp
ArithOp	::=	'+' | '−' | '∗' | '/'
Num	::=	Digit {Digit}
String	::=	'"' {Char} '"'

Figure 3: Syntax of node and edge labels

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

Figure 4: Syntax of conditions

also occur in the left graph of the schema. The predicate `edge` demands the (non-)existence of an edge between two nodes in the graph to which the rule schema is applied. For example, the expression `not edge(1,3)` in the condition of Figure 2 forbids an edge from node 1 to node 3 when the left graph is matched.

We interpret a conditional rule schema as the (possibly infinite) set of conditional rules that is obtained by instantiating variables with any values and evaluating expressions. To define this, consider a declaration D of a conditional rule-schema. Let L and R be the left and right graphs of D, and c the condition. We write $\mathrm{Var}(D)$ for the set of variable identifiers occurring in D. Given x in $\mathrm{Var}(D)$, $\mathrm{type}(x)$ denotes the type associated with x. An *assignment* is a mapping $\alpha\colon \mathrm{Var}(D) \to (\mathbb{Z} \cup \mathrm{Char}^*)$ such that for each x in $\mathrm{Var}(D)$, $\mathrm{type}(x) = \texttt{int}$ implies $\alpha(x) \in \mathbb{Z}$, and $\mathrm{type}(x) = \texttt{string}$ implies $\alpha(x) \in \mathrm{Char}^*$.

Given a label l of category RightLabel occuring in D and an assignment α, the value $l^\alpha \in \mathscr{L}$ is inductively defined. If l is a numeral or a sequence of characters, then l^α is the integer or character string represented by l (which is independent of α). If l is a variable identifier, then $l^\alpha = \alpha(l)$. Otherwise, l^α is obtained from the values of l's components. If l has the form $e_1 \oplus e_2$ with $\oplus$ in ArithOp and e_1, e_2 in Exp, then $l^\alpha = e_1^\alpha \oplus_\mathbb{Z} e_2^\alpha$ where $\oplus_\mathbb{Z}$ is the integer operation represented by $\oplus$.[4] If l has the form e_m with e in Exp and m in RightLabel, then $l^\alpha = e^\alpha m^\alpha$ (the concatenation of e^α and m^α). Note that our definition of l^α covers all labels in D since LeftLabel is a subcategory of RightLabel.

The value of the condition c in D not only depends on an assignment but also on a graph morphism. For, if c contains the predicate `edge`, we need to consider the structure of the graph to which we want to apply the rule schema. Consider an assignment α and let L^α be obtained from L by replacing each label l with l^α. Let $g\colon L^\alpha \to G$ be a graph morphism with $G \in \mathscr{G}$. Then for each Boolean subexpression b of c, the value $b^{\alpha,g}$ in $\mathbb{B} = \{\texttt{tt}, \texttt{ff}\}$ is inductively defined. If b has the form $e_1 \bowtie e_2$ with $\bowtie$ in RelOp and e_1, e_2 in Exp, then $b^{\alpha,g} = \texttt{tt}$ if and only if $e_1^\alpha \bowtie_\mathbb{Z} e_2^\alpha$ where $\bowtie_\mathbb{Z}$ is the relation on integers represented by

[4] For simplicity, we consider division by zero as an implementation-level issue.

$\bowtie$. If b has the form $\texttt{not}\, b_1$ with b_1 in BoolExp, then $b^{\alpha,g} = \texttt{tt}$ if and only if $b_1^{\alpha,g} = \texttt{ff}$. If b has the form $b_1 \oplus b_2$ with $\oplus$ in BoolOp and b_1, b_2 in BoolExp, then $b^{\alpha,g} = b_1^{\alpha,g} \oplus_{\mathbb{B}} b_2^{\alpha,g}$ where $\oplus_{\mathbb{B}}$ is the Boolean operation on $\mathbb{B}$ represented by $\oplus$. A special case is given if b has the form $\texttt{edge}(v,w)$ where v, w are identifiers of interface nodes in D. We then have

$$b^{\alpha,g} = \begin{cases} \texttt{tt} & \text{if there is an edge from } g(v) \text{ to } g(w), \\ \texttt{ff} & \text{otherwise.} \end{cases}$$

Let now r be the rule-schema identifier associated with declaration D. For every assignment α, let $r^\alpha = (L^\alpha \leftarrow K \rightarrow R^\alpha, P^\alpha)$ be the conditional rule given as follows:

- L^α and R^α are obtained from L and R by replacing each label l with l^α.
- K is the discrete subgraph of L and R determined by the node identifiers for the interface, where all nodes are unlabelled.
- P^α is defined by: $P^\alpha(g)$ if and only if g is a graph morphism $L^\alpha \to G$ such that $G \in \mathscr{G}$ and $c^{\alpha,g} = \texttt{tt}$.

The *interpretation* of r is the rule set $\mathrm{I}(r) = \{r^\alpha \mid \alpha \text{ is an assignment}\}$. For notational convenience, we sometimes denote the relation $\Rightarrow_{\mathrm{I}(r)}$ by $\Rightarrow_r$. Note that $\mathrm{I}(r)$ is a (possibly infinite) set of conditional rules in the sense of Section 2, grounding rule schemata in the theory of the double-pushout approach with relabelling [9].

For example, the upper rows of Figure 5 show the rule schema `bridge` of Figure 2 (without condition) and its instance $\texttt{bridge}^\alpha$, where $\alpha(\texttt{x}) = 0$, $\alpha(\texttt{y}) = \alpha(\texttt{z}) = 1$, $\alpha(\texttt{a}) = 3$ and $\alpha(\texttt{b}) = 2$. The condition c of `bridge` evaluates to the predicate P^α which is true for a match g of the left-hand graph if and only if there is no edge from $g(1)$ to $g(3)$. (The subexpressions $\texttt{a} >= 0$ and $\texttt{b} >= 0$ evaluate to `tt` and hence can be ignored.) The lower rows of Figure 5 show an application of $\texttt{bridge}^\alpha$ by a graph morphism satisfying P^α.

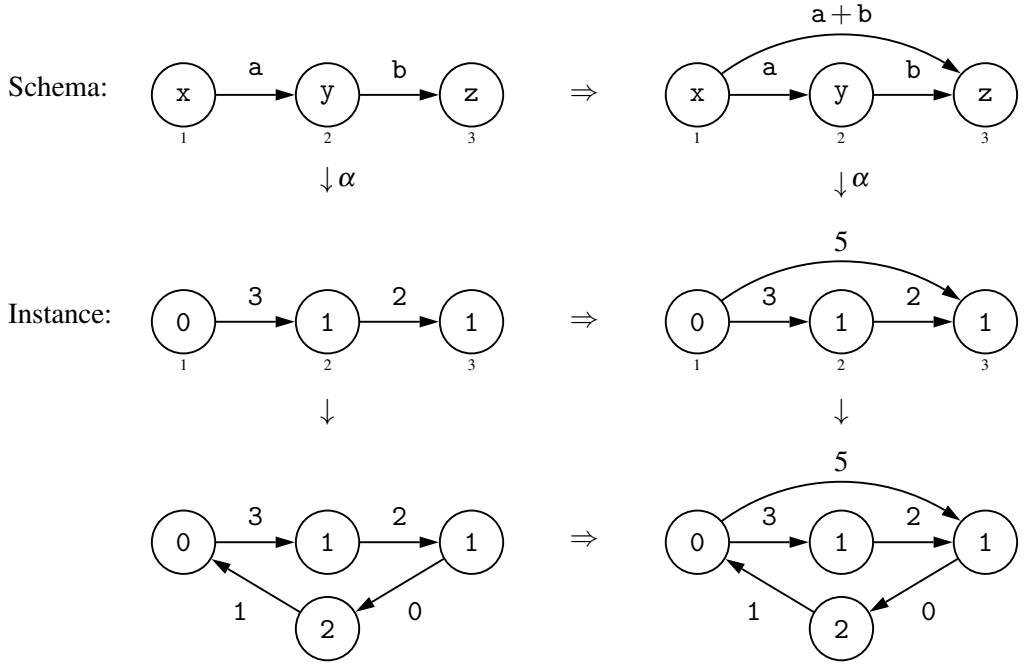

Figure 5: Application of a rule schema using instantiation

4 Graph Programs

We start by discussing an example program for graph colouring.

Example 1 (Computing a 2-colouring). A *colouring* for a graph is an assignment of colours (integers) to nodes such that the source and target of each edge have different colours. A graph is *2-colourable* (or *bipartite*) if it possesses a colouring with at most two colours. The program `2-colouring` in Figure 6 generates a 2-colouring for nonempty, connected input graphs without loops if such a colouring exists—otherwise the input graph is returned. The program consists of five rule-schema declarations, the *macro* `colour` representing the rule-schema set {`colour1`, `colour2`}, and the main command sequence following the key word `main`.

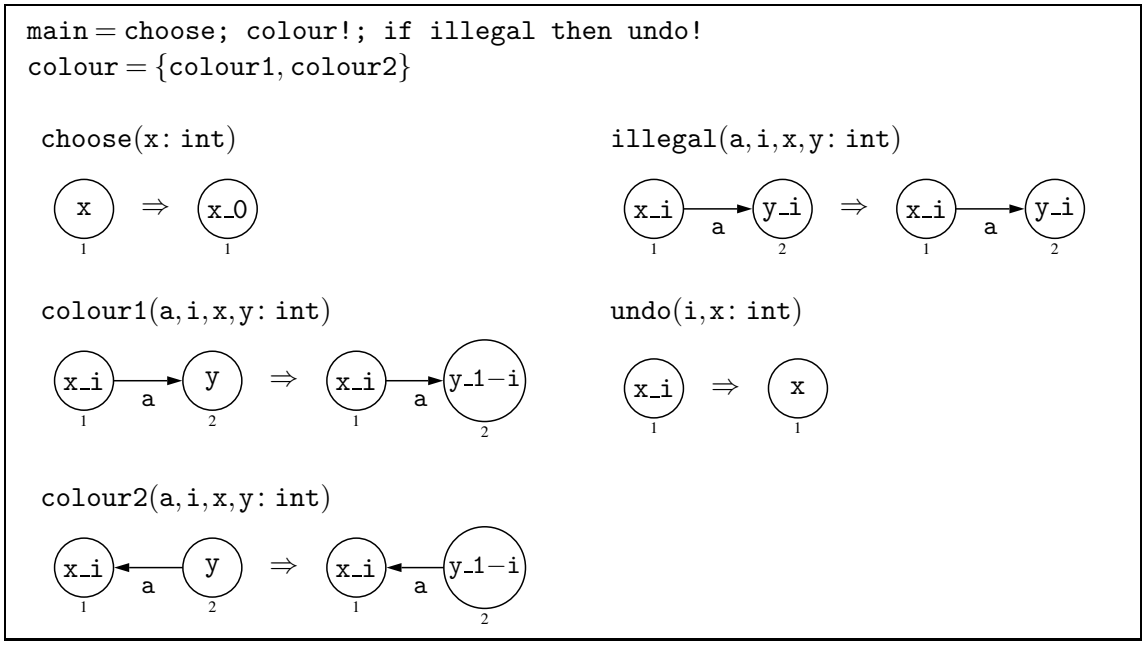

Figure 6: The program `2-colouring`

Given an integer-labelled input graph, the program first uses the rule schema `choose` to pick any node and replace its label x with x_0. The underscore operator allows to add a *tag* to a label, used here to add colours to labels. In general, a tagged label consists of a sequence of expressions joined by underscores. After the first node has been coloured, the command `colour!` applies the rule schemata `colour1` and `colour2` nondeterministically as long as possible to colour all remaining nodes. In each iteration of the loop, an uncoloured node adjacent to an already coloured node v gets the colour in $\{0,1\}$ that is complementary to v's colour. If the input graph is connected, the graph resulting from `colour!` is correctly coloured if and only if the rule schema `illegal` is not applicable. The latter is checked by the if-statement. If `illegal` is applicable, then the input must contain an undirected cycle of odd length and hence is not 2-colourable (see for example [10]). In this case the loop `undo!` removes all tags to return the input graph unmodified. Note that the number of rule-schema applications performed by `2-colouring` is linear in the number of input nodes.

To make `2-colouring` applicable to graphs that are possibly empty or disconnected, we can insert

a nested loop:

```
main = (choose; colour!)!; if illegal then undo!.
```

Now if the input graph is empty, `choose` fails which causes the outer loop to terminate and return the current (empty) graph. On the other hand, if the input consists of several connected components, the body of the outer loop is repeatedly called to colour each component.

Figure 7 shows the abstract syntax of GP programs.[5] A program consists of a number of declarations of conditional rule schemata and macros, and exactly one declaration of a main command sequence. The rule-schema identifiers (category RuleId) occurring in a call of category RuleSetCall refer to declarations of conditional rule schemata in category RuleDecl (see Section 3). Semantically, each rule-schema identifier r stands for the set $\mathrm{I}(r)$ of conditional rules induced by that identifier. A call of the form $\{r_1,\ldots,r_n\}$ stands for the union $\bigcup_{i=1}^{n}\mathrm{I}(r_i)$.

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]
		| ComSeq '!'
		| `skip` | `fail`
RuleSetCall	::=	RuleId | '{' [RuleId {',' RuleId}] '}'
MacroCall	::=	MacroId

Figure 7: Abstract syntax of GP

Macros are a simple means to structure programs and thereby to make them more readable. Every program can be transformed into an equivalent macro-free program by replacing macro calls with their associated command sequences (recursive macros are not allowed). In the next section we use the terms "program" and "command sequence" synonymously, assuming that all macro calls have been replaced.

The commands `skip` and `fail` can be expressed through the other commands (see next section), hence the core of GP includes only the call of a set of conditional rule schemata (RuleSetCall), sequential composition (';'), the if-then-else statement and as-long-as-possible iteration ('!').

5 Semantics of Graph Programs

We present a formal semantics of GP in the style of Plotkin's structural operational semantics [14]. As usual for this approach, inference rules inductively define a small-step transition relation $\to$ on *configurations*. In our setting, a configuration is either a command sequence together with a graph, just a graph or the special element fail:

$$\to \subseteq (\mathrm{ComSeq}\times\mathscr{G})\times((\mathrm{ComSeq}\times\mathscr{G})\cup\mathscr{G}\cup\{\mathrm{fail}\}).$$

[5] Where necessary we use parentheses to disambiguate programs.

Configurations in ComSeq $\times \mathscr{G}$ represent unfinished computations, given by a rest program and a state in the form of a graph, while graphs in $\mathscr{G}$ are proper results of computations. In addition, the element fail represents a failure state. A configuration γ is *terminal* if there is no configuration δ such that $\gamma \to \delta$.

Each inference rule in Figure 8 consists of a premise and a conclusion separated by a horizontal bar. Both parts contain meta-variables for command sequences and graphs, where R stands for a call in category RuleSetCall, C, P, P', Q stand for command sequences in category ComSeq and G, H stand for graphs in $\mathscr{G}$. Given a rule-set call R, let $\mathrm{I}(R) = \bigcup\{\mathrm{I}(r) \mid r \text{ is a rule-schema identifier in } R\}$ (see Section 3 for the definition of $\mathrm{I}(r)$). The *domain* of $\Rightarrow_{\mathrm{I}(R)}$, denoted by $\mathrm{Dom}(\Rightarrow_{\mathrm{I}(R)})$, is the set of all graphs G in $\mathscr{G}$ such that $G \Rightarrow_{\mathrm{I}(R)} H$ for some graph H. Meta-variables are considered to be universally quantified. For example, the rule $[\mathrm{Call}_1]$ should be read as: "For all R in RuleSetCall and all G, H in $\mathscr{G}$, $G \Rightarrow_{\mathrm{I}(R)} H$ implies $\langle R, G\rangle \to H$."

Figure 8 shows the inference rules for the core constructs of GP. We write $\to^+$ and $\to^*$ for the transitive and reflexive-transitive closures of $\to$. A command sequence C *finitely fails* on a graph $G \in \mathscr{G}$ if (1) there does not exist an infinite sequence $\langle C, G\rangle \to \langle C_1, G_1\rangle \to \ldots$ and (2) for each terminal configuration γ such that $\langle C, G\rangle \to^* \gamma$, $\gamma = \text{fail}$. In other words, C finitely fails on G if all computations starting from (C, G) eventually end in the configuration fail.

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

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

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

$$[\mathrm{If}_1]\ \frac{\langle C, G\rangle \to^+ H}{\langle \texttt{if } C \texttt{ then } P \texttt{ else } Q, G\rangle \to \langle P, G\rangle} \qquad [\mathrm{If}_2]\ \frac{C \text{ finitely fails on } G}{\langle \texttt{if } C \texttt{ then } P \texttt{ else } Q, G\rangle \to \langle Q, G\rangle}$$

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

Figure 8: Inference rules for core commands

The concept of finite failure stems from logic programming where it is used to define *negation as failure* [4]. In the case of GP, we use it to define powerful branching and iteration constructs. In particular, our definition of the if-then-else command allows to "hide" destructive tests.

Example 2 (Recognizing series-parallel graphs). A graph is *series-parallel* if it reduces to a graph consisting of two nodes and an edge between them by the following two operations [1, 5]: (1) Replace a pair of parallel edges by an edge from their source to their target. (2) Given a node v with exactly one incoming edge e_1 and exactly one outgoing edge e_2 such that the source of e_1 and the target of e_2 are distinct, replace e_1, e_2 and v by an edge from the source of e_1 to the target of e_2.

Suppose that we want to check whether a connected, integer-labelled graph G is series-parallel and, depending on the result, execute either a program P or a program Q on G. We can do this with the program

$$\texttt{main} = \texttt{if } \{\texttt{par}, \texttt{seq}\}!; \texttt{base then } P \texttt{ else } Q$$

whose rule schemata `par`, `seq` and `base` are shown in Figure 9. The subprogram $\{\texttt{par}, \texttt{seq}\}!$ applies

as long as possible the operations (1) and (2) to the input graph G, then the rule schema `base` checks if the resulting graph consists of two nodes connected by an edge. Graph G is series-parallel if and only if `base` is applicable to the reduced graph. (Note that $\{\texttt{par}, \texttt{seq}\}!$ preserves connectedness and that, by the dangling condition, `base` is applicable only if the images of its left-hand nodes have degree one.) It is important to note that by the inference rules [If$_1$] and [If$_2$], the main program executes P or Q *on the input graph* G whereas the graph resulting from the test is discarded.

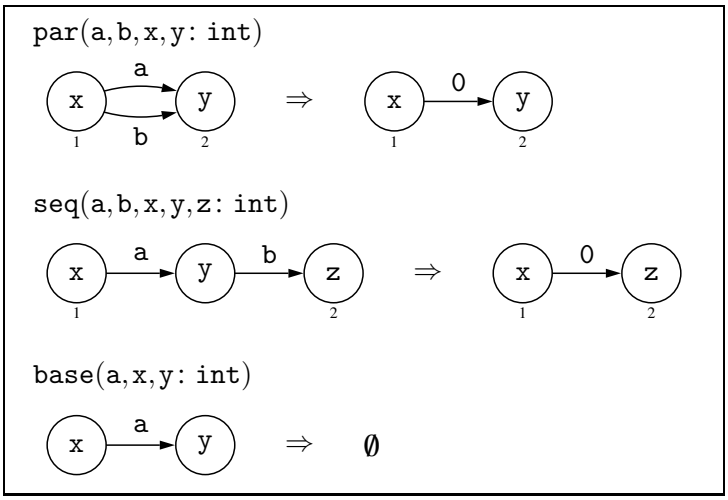

Figure 9: Rule schemata for recognizing series-parallel graphs

The meaning of the remaining GP commands is defined in terms of the meaning of the core commands, see Figure 10. We refer to these commands as *derived* commands.

[Skip] $\langle \texttt{skip}, G\rangle \to \langle \texttt{r}, G\rangle$
where `r` is an identifier for the rule schema $\emptyset \Rightarrow \emptyset$

[[Fail] $\langle \texttt{fail}, G\rangle \to \langle \{\}, G\rangle$

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

Figure 10: Inference rules for derived commands

We can now summarise the meaning of GP programs by a semantic function $[\![\_]\!]$ which assigns to each program P the function $[\![P]\!]$ mapping an input graph G to the set of all possible results of running P on G. The result set may contain, besides proper results in the form of graphs, the special value $\perp$ which indicates a nonterminating or stuck computation. The *semantic function* $[\![\_]\!]\colon \mathrm{ComSeq} \to (\mathscr{G} \to 2^{\mathscr{G}\cup\{\perp\}})$ is defined by[6]

$$[\![P]\!]G = \{H \in \mathscr{G} \mid \langle P, G\rangle \xrightarrow{+} H\} \cup \{\perp \mid P \text{ can diverge or get stuck from } G\}$$

where P *can diverge from* G if there is an infinite sequence $\langle P, G\rangle \to \langle P_1, G_1\rangle \to \langle P_2, G_2\rangle \to \ldots$, and P *can get stuck from* G if there is a terminal configuration $\langle Q, H\rangle$ such that $\langle P, G\rangle \to^* \langle Q, H\rangle$.

[6] We write $[\![P]\!]G$ for the application of $[\![P]\!]$ to a graph G.

Note that $[\![P]\!]G = \emptyset$ if and only if P finitely fails on G. In Example 2, for instance, we have $[\![\{\texttt{par}, \texttt{seq}\}!; \texttt{base}]\!]G = \emptyset$ for every connected graph G containing a cycle. This is because the graph resulting from $\{\texttt{par}, \texttt{seq}\}!$ is still connected and cyclic, so the rule schema `base` is not applicable.

A program can get stuck only in two situations: either it contains a subprogram `if` C `then` P `else` Q where C both can diverge from some graph and cannot produce a proper result from that graph, or it contains a subprogram $B!$ where the loop's body B possesses the said property of C. The evaluation of these subprograms will get stuck because the inference rules for branching and iteration are not applicable.

6 Conclusion

GP is an experimental rule-based language for high-level problem solving in the domain of graphs, freeing programmers from handling low-level data structures. The hallmark of GP is syntactic and semantic simplicity. Conditional rule schemata for graph transformation allow to express application conditions and computations on labels, in addition to structural changes. The semantics of rule schemata is orthogonal to the semantics of control constructs, making it easy to change the format of rules or graphs.

The operational semantics of programs describes the effect of GP's control constructs in a natural way and captures the nondeterminism of the language. In particular, powerful branching and iteration commands have been defined using the concept of finite failure. Destructive tests on the current graph can be hidden in the condition of the branching command, and nested loops can be coded since arbitrary subprograms can be iterated as long as possible.

Future extensions of GP may include recursive procedures for writing complex algorithms (see [19]), and a type concept for restricting the shape of graphs. Our goal is to support formal reasoning on graph programs by developing static analyses for properties such as termination and confluence (uniqueness of results), and a calculus and tool support for program verification.

References

[1] Jørgen Bang-Jensen and Gregory Gutin. *Digraphs: Theory, Algorithms and Applications*. Springer-Verlag, 2000.

[2] Peter Borovanský, Claude Kirchner, Hélène Kirchner, and Pierre-Etienne Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285(2):155–185, 2002.

[3] Martin Bravenboer, Arthur van Dam, Karina Olmos, and Eelco Visser. Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae*, 69(1–2):123–178, 2006.

[4] Keith L. Clark. Negation as failure. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[5] R. J. Duffin. Topology of series-parallel networks. *Journal of Mathematical Analysis and Applications*, 10:303–318, 1965.

[6] Claudia Ermel, Michael Rudolf, and Gabi Taentzer. The AGG approach: Language and environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, chapter 14, pages 551–603. World Scientific, 1999.

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

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

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

[10] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Addison Wesley, 2006.

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

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

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

[14] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, 2004.

[15] Detlef Plump. The graph programming language GP. In *Proc. Algebraic Informatics (CAI 2009)*, volume 5725 of *Lecture Notes in Computer Science*, pages 99–122. Springer-Verlag, 2009.

[16] Detlef Plump and Sandra Steinert. Towards graph programs for graph algorithms. In *Proc. International Conference on Graph Transformation (ICGT 2004)*, volume 3256 of *Lecture Notes in Computer Science*, pages 128–143. Springer-Verlag, 2004.

[17] Andy Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen*. Deutscher Universitäts-Verlag, 1991. In German.

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

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