The Design of GP 2

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

This papers defines the syntax and semantics of GP 2, a revised version of the graph programming language GP. New concepts are illustrated and explained with example programs. Changes to the first version of GP include an improved type system for labels, a built-in marking mechanism for nodes and edges, a more powerful `edge` predicate for conditional rule schemata, and functions returning the indegree and outdegree of matched nodes. Moreover, the semantics of the branching and loop statement have been simplified to allow their efficient implementation.

1 Introduction

GP is 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 and has a simple syntax and semantics, to facilitate both understanding by programmers and formal reasoning on programs. The original version of GP (also referred to as GP 1 from now on) is defined in [7, 8] and its protoype implementation is described in [4].

Motivated by case studies in GP programming, the following changes and extensions feature in GP 2:

- There are new types `atom` and `list`, the former representing the union of integers and character strings, the latter lists of atoms. Variables of these types can be declared in rule schemata.
- Rule schemata can *mark* nodes and edges graphically.
- Conditional rule schemata can check, by means of the `edge` predicate, whether there exists an edge with a particular label between two matched nodes.
- The indegree or outdegree of a matched node can be accessed and used in the labels or in the condition of a rule schema.
- The if-then-else statement of GP 1 is complemented by a try-then-else command whose then-part is executed on the graph resulting from the try-part. Also, a new `or` command provides explicit nondeterministic choice between subprograms.
- Failure in evaluating the condition of a branching statement or the body of a loop does no longer enforce backtracking, in order to allow an efficient implementation of branching and looping.

The rest of this paper is organised as follows. In Section 2, the graph transformation approach underlying GP is briefly reviewed, viz. the double-pushout approach with relabelling. Section 3 introduces conditional rule schemata, the building blocks of GP programs. The semantics of conditional rule schemata is defined in Section 4. In Section 5, new features of GP 2 are demonstrated and explained by example programs. A formal operational semantics for GP 2 is presented and discussed in Section 6. Section 7 concludes by summarising GP's revision and addressing topics for future work. The Appendix lists the inference rules of the semantics of Section 6.

2 Graphs and Graph Transformation

Graph transformation in GP is based on the double-pushout approach with relabelling [3]. This framework deals with partially labelled graphs whose definition is recalled below. In this section, we treat the label alphabet as a parameter because in subsequent sections we need different alphabets: graphs in rule schemata are labelled with expressions while graphs on which GP programs operate (also referred to as host graphs) are labelled with lists composed of integers and strings.

A *graph* over a label alphabet $\mathscr{C}$ is a system $G = (V_G, E_G, s_G, t_G, l_G, m_G)$, where V_G and E_G are finite sets of *nodes* (or *vertices*) and *edges*, $s_G, t_G \colon E_G \to V_G$ are the *source* and *target* functions for edges, $l_G \colon V_G \to \mathscr{C}$ is the partial node labelling function and $m_G \colon E_G \to \mathscr{C}$ 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. We write $\mathscr{G}(\mathscr{C}_\perp)$ and $\mathscr{G}(\mathscr{C})$ for the class of graphs resp. totally labelled graphs over $\mathscr{C}$.

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

A *graph morphism* $g \colon G \to H$ between graphs G, H in $\mathscr{G}(\mathscr{C}_\perp)$ 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* (*surjective*) if g_V and g_E are injective (surjective). It is an *isomorphism* if it is injective, surjective and satisfies $l_H(g_V(v)) = \perp$ for all nodes v with $l_G(v) = \perp$. In this case G and H are *isomorphic*, which is denoted by $G \cong H$.

A *rule* $r = \langle L \leftarrow K \rightarrow R \rangle$ consists of two inclusions $K \to L$ and $K \to R$ such that L, R are graphs in $\mathscr{G}(\mathscr{C})$ and K, the *interface* of r, is a graph in $\mathscr{G}(\mathscr{C}_\perp)$. 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.

Definition 1 (Rule application)**.** Let $r = \langle L \leftarrow K \rightarrow R \rangle$ be a rule, G a graph in $\mathscr{G}(\mathscr{C})$, and $g \colon L \to G$ an injective graph morphism satisfying the *dangling condition*: no node in $g(L) - g(K)$ is incident to an edge in $G - g(L)$. We write $G \Rightarrow_{r,g} H$ if H is isomorphic to the graph that is constructed from G as follows:

1. Remove all nodes and edges in $g(L) - g(K)$, obtaining a graph D.
2. Add disjointly to D all nodes and edges from $R - K$, 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 unlabelled node v in K, $l_H(g_V(v))$ becomes $l_R(v)$.

Figure 1 shows an example of a rule application. 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 assume that all edge labels are the same and hence omit them.) The node identifiers 1 and 2 in the rule specify the inclusions of the interface. The middle graph of the lower row is obtained from graph D of Definition 1 by making all nodes unlabelled that are images of unlabelled nodes in K. Then the diagram represents a double-pushout in the category of graphs over $\mathscr{C}_\perp$ (see [3]).

3 Syntax of Rule Schemata

Conditional rule schemata are the principal programming construct in GP. Figure 2 shows an (artificial) example for the declaration of a rule schema containing some of the new features of GP 2.

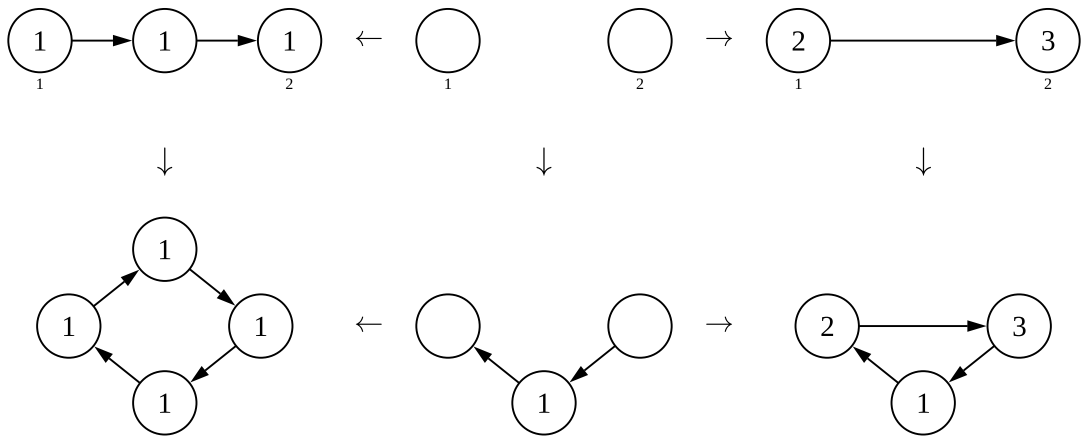


Figure 1: A rule application

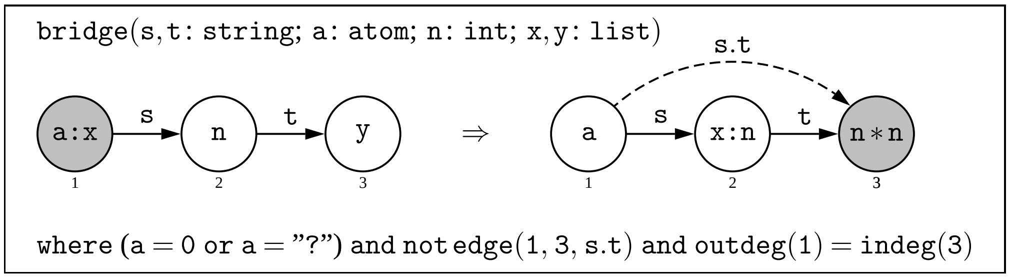

bridge(s,t: string; a: atom; n: int; x,y: list)
s.t
a:x
s
n
t
y
⇒
a
s
x:n
t
n∗n
1
2
3
1
2
3
where (a = 0 or a = "?") and not edge(1, 3, s.t) and outdeg(1) = indeg(3)

Figure 2: Declaration of a conditional rule schema

Besides the types `int` and `string` of GP 1, there are the new types `atom` and `list`. Type `atom` is the union of `int` and `string`, and `list` is the type of a (possibly empty) list of atoms. Given lists `x` and `y`, we write `x:y` for the concatenation of `x` and `y`. The colon replaces the underscore '_' of GP 1 for better readability. Also, the empty list `empty` is now allowed (not to be confused with the empty character string ""). When drawing graphs, we represent the empty list by omitting the word `empty`. (Confusion with unlabelled nodes is not possible as long as we consider graphs on the left or right of a rule schema, or host graphs. This is because these graphs are totally labelled.)

We identify lists of length one with their contents and hence get the syntactic and semantic *subtype* relationships shown in Figure 3. This is why we can form list expressions such as `a:x` and `x:n` in Figure 2, where `x` is a list, `a` an atom and `n` an integer. For the same reason, equations in the condition such as $\mathtt{a} = 0$ or $\mathtt{a} =$ "?" can compare expressions of arbitrary list subtypes.

Expressions in the left-hand side of a rule schema need no longer be constants or variables. Composite expressions such as `a:x` in Figure 2 are allowed if there is no ambiguity in matching individual variables with values in host graph labels. Similarly, the new dot operator '`.`' for string concatenation can be used in left-hand labels. (The exact condition for left-hand expressions is given in Definition 2.)

The new functions `indeg` and `outdeg` access the indegree resp. outdegree of a left-hand node in the host graph. These operators may occur in the labels and the condition of a rule schema. Moreover, the binary `edge` predicate of GP 1 has now an optional third argument specifying the label of a possible edge

$$\begin{tikzcd}
& \texttt{list} \arrow[d, phantom, sloped, "\supseteq"] & & & & (\mathbb{Z} \cup \mathrm{Char}^*)^* \arrow[d, phantom, sloped, "\supseteq"] & \\
& \texttt{atom} \arrow[dl, phantom, sloped, "\supseteq"] \arrow[dr, phantom, sloped, "\supseteq"] & & & & \mathbb{Z} \cup \mathrm{Char}^* \arrow[dl, phantom, sloped, "\supseteq"] \arrow[dr, phantom, sloped, "\supseteq"] & \\
\texttt{int} & & \texttt{string} & & \mathbb{Z} & & \mathrm{Char}^*
\end{tikzcd}$$

Figure 3: Subtype hierarchy for lists

between the given nodes. For example, the subcondition `not edge`$(1,3,$`s.t`$)$ in Figure 2 demands that there must not be an edge from node `1` to node `3` with label `s.t` (where the strings denoted by `s` and `t` are determined by matching the left-hand graph).

Finally, GP 2 allows to mark nodes and edges graphically. For example, the outermost nodes in Figure 2 are marked by a grey shading, and the dashed arrow between nodes 1 and 3 in the right graph represents a marked edge. Marking is formalised below by defining labels as pairs of lists and boolean values, where a boolean value indicates whether a node or edge is marked or not.

Figure 4 and Figure 5 give grammars in Extended Backus-Naur Form defining the abstract syntax of the labels and the condition of a rule schema. These grammars are ambiguous; in examples we use parentheses to disambiguate expressions if necessary. In the next section, the abstract syntax is used in defining the semantics of rule schemata.

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 4: Abstract syntax of rule schema labels

The grammar in Figure 4 defines four syntactic categories of expressions which can occur in a rule schema: Integer, String, Atom and List, where Integer and String are subsets of Atom which in turn is a subset of List. We assume that Node is the set of node identifiers occurring in the rule schema, which must be the same for the left and the right graph ($\{1,2,3\}$ in Figure 2). Moreover, IVariable, SVariable, AVariable and LVariable are the sets of variables of type `int`, `string`, `atom` and `list` that occur in the rule schema. These categories are disjoint since each variable must be declared with a unique type (see Figure 2). The mark components of labels are represented graphically rather than textually.

The values of variables at execution time are determined by graph matching, hence we require that expressions in the left graph of a rule schema must have a simple shape.

Definition 2 (Simple list)**.** An expression $e \in$ List is *simple* if

(1) e contains no arithmetic operators,

(2) e contains at most one occurrence of a list variable, and

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

For example, given the variable declarations of Figure 2, `a:x` and ”`no`”`.s:y:t` are simple expressions whereas `x:y` and `s.t` are not simple.

The syntax of a rule schema condition is defined by the grammar in Figure 5. New features are the predicates `int`, `string` and `atom`, which allow to check whether an expression belongs to a subtype of `list`, and equations between arbitrary list expressions. Also, the `edge` predicate can have a third parameter specifying the list component of an edge label.

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

Figure 5: Abstract syntax of rule-schema condition

Definition 3 (Conditional rule schema)**.** A *rule schema* $\langle L \leftarrow K \rightarrow R\rangle$ consists of two inclusions $K \rightarrow L$ and $K \rightarrow R$ such that L, R are graphs in $\mathscr{G}(\text{Label})$ and K consists of unlabelled nodes only. We require that all list expressions in L are simple and that all variables occurring in R also occur in L. A *conditional* rule schema $\langle L \leftarrow K \rightarrow R, c\rangle$ consists of a rule schema $\langle L \leftarrow K \rightarrow R\rangle$ and a condition $c \in$ Condition such that all variables occurring in c also occur in L.

When a conditional rule schema is declared, as in Figure 2, graph K is implicitly represented by the node identifiers in L and R (which much coincide). Hence nodes without identifiers in L are to be deleted and nodes without identifiers in R are to be created.

The requirement that all variables in R must also occur in L ensures that for a given match of L in a host graph, applying r produces a unique graph (up to isomorphism). Similarly, the evaluation of c has a unique result if all its variables occur in L.

4 Semantics of Rule Schemata

While the left and right graph of a rule schema are labelled with elements from the syntactic category Label, host graphs are labelled with values from the following semantic domain $\mathscr{L}$:

$$\mathscr{L} = (\mathbb{Z} \cup \text{Char}^*)^* \times \mathbb{B}$$

where $\mathbb{B} = \{\text{true}, \text{false}\}$. Hence semantic labels are sequences consisting of integers and character strings[1], paired with boolean values. As in the case of syntactic labels, the individual elements of a

[1]We assume that Char is a fixed set of characters.

sequence are separated by colons, the empty sequence is represented by “white space”, and the boolean value true is represented graphically by shading resp. dashed lines.

The application of a rule schema r with condition c to a graph G in $\mathscr{G}(\mathscr{L})$ proceeds roughly as follows:

1. Match the left graph L of r with a subgraph of G, ignoring labels, by means of a premorphism $g\colon L \to G$.
2. Check whether there is an assignment α of values to variables such that after evaluating the expressions in L, g is label-preserving.
3. Check whether the condition c evaluates to true.
4. Apply the rule $r^{g,\alpha}$, obtained from r by evaluating all expressions in the left and right graph, to G.

For example, Figure 6 shows an application of the rule schema `bridge` of Figure 2. The upper half of the diagram represents the instantiation of `bridge` according to premorphism g and the following assignment α: $\mathtt{a} \mapsto 0$, $\mathtt{x} \mapsto 1:2$, $\mathtt{n} \mapsto 3$, $\mathtt{y} \mapsto 4$, $\mathtt{s} \mapsto$ ”o”, $\mathtt{t} \mapsto$ ”k”. The lower half of the diagram represents the application of the instance $\mathtt{bridge}^{g,\alpha}$ according to g. Note that the application condition of `bridge` (see Figure 2) is satisfied with respect to g and α.

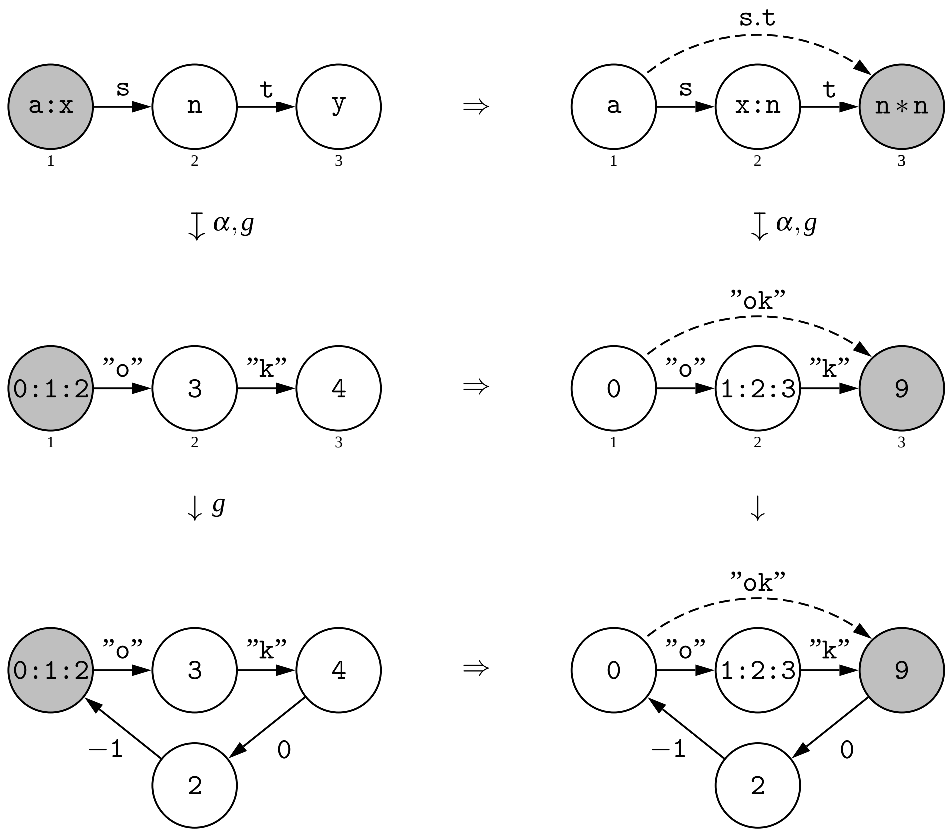


Figure 6: An application of the rule schema `bridge` of Figure 2

In the remainder of this section, we make the above four steps precise. Consider a conditional rule schema $r = \langle L \leftarrow K \to R, c\rangle$. Given graphs G in $\mathscr{G}(\text{Label})$ and H in $\mathscr{G}(\mathscr{L})$, a *premorphism* $g\colon G \to H$

consists of two functions $g_V\colon V_G \to V_H$ and $g_E\colon E_G \to E_H$ that preserve sources and targets: $s_H \circ g_E = g_V \circ s_G$ and $t_H \circ g_E = g_V \circ t_G$.

An *assignment* is a family of mappings $\alpha = (\alpha_X)_{X\in\{\mathrm{I,S,A,L}\}}$ where $\alpha_{\mathrm{I}}\colon \mathrm{IVariable} \to \mathbb{Z}$, $\alpha_{\mathrm{S}}\colon \mathrm{SVariable} \to \mathrm{Char}^*$, $\alpha_{\mathrm{A}}\colon \mathrm{AVariable} \to \mathbb{Z} \cup \mathrm{Char}^*$ and $\alpha_{\mathrm{L}}\colon \mathrm{LVariable} \to \mathscr{L}$. We sometimes omit the subscripts of these mappings as exactly one of them is applicable to a given variable.

Given a premorphism $g\colon L \to G$, an assignment α and a label $l = e\,m$ with $e \in \mathrm{List}$ and $m \in \{\mathtt{true}, \mathtt{false}\}$, the value $l^{g,\alpha} \in \mathscr{L}$ is the pair $\langle e^{g,\alpha}, \mathrm{true}\rangle$ if $m = \mathtt{true}$ and $\langle e^{g,\alpha}, \mathrm{false}\rangle$ otherwise.

The value $e^{g,\alpha} \in (\mathbb{Z} \cup \mathrm{Char}^*)^*$ is inductively defined. If $e = \mathtt{empty}$, then $e^{g,\alpha}$ is the empty sequence. If e has the form $d_1 \ldots d_n$ ($n \geq 1$) with digits $d_1, \ldots, d_n$, or has the form "$c_1 \ldots c_n$" ($n \geq 0$) with characters $c_1, \ldots, c_n$, then $e^{g,\alpha}$ is the unique integer in $\mathbb{Z}$ resp. character string in Char^* represented by e. (Note that the empty character string and $\mathtt{empty}^{g,\alpha}$ are different values.) If e is a variable, then $e^{g,\alpha} = \alpha(e)$. Otherwise, e^{α} is obtained from the values of e's components. If $e = -e_1$ with $e_1 \in \mathrm{Integer}$, then $e^{g,\alpha}$ is the integer opposite to $e_1^{g,\alpha}$. If e has the form $e_1 \oplus e_2$ with $\oplus \in \mathrm{ArithOp}$ and $e_1, e_2 \in \mathrm{Integer}$, then $e^{g,\alpha} = e_1^{g,\alpha} \oplus_{\mathbb{Z}} e_2^{g,\alpha}$ where $\oplus_{\mathbb{Z}}$ is the integer operation represented by $\oplus$.[2] If e has the form $\mathtt{indeg}(n)$ or $\mathtt{outdeg}(n)$, with $n \in \mathrm{Node}$, then $e^{g,\alpha}$ is the indegree resp. outdegree of the node $g_V(n)$ in G. Finally, if $e = e_1.e_2$ with $e_1, e_2 \in \mathrm{String}$ or $e = e_1{:}e_2$ with $e_1, e_2 \in \mathrm{List}$, then $e^{g,\alpha}$ is the concatenation of $e_1^{g,\alpha}$ and $e_2^{g,\alpha}$.

The value $c^{g,\alpha} \in \mathbb{B}$ of the condition c is also inductively defined. If c has the form $\mathtt{int}(e_1)$ with $e_1 \in \mathrm{List}$, then $c^{g,\alpha} = \mathrm{true}$ if and only if $e_1^{g,\alpha} \in \mathbb{Z}$. Similarly, if c has the form $\mathtt{string}(e_1)$ or $\mathtt{atom}(e_1)$, then $c^{g,\alpha} = \mathrm{true}$ if and only if $e_1^{g,\alpha} \in \mathrm{Char}^*$ resp. $e_1^{g,\alpha} \in \mathbb{Z} \cup \mathrm{Char}^*$. If c has the form $e_1 = e_2$ or $e_1 \,!{=}\, e_2$ with $e_1, e_2 \in \mathrm{List}$, then $c^{g,\alpha} = \mathrm{true}$ if and only if $e_1^{g,\alpha} = e_2^{g,\alpha}$ resp. $e_1^{g,\alpha} \neq e_2^{g,\alpha}$. If c has the form $e_1 \bowtie e_2$ with $\bowtie \in \mathrm{RelOp}$ and e_1, e_2 in Integer, then $c^{g,\alpha} = \mathrm{true}$ if and only if $e_1^{g,\alpha} \bowtie_{\mathbb{Z}} e_2^{g,\alpha}$ where $\bowtie_{\mathbb{Z}}$ is the integer relation represented by $\bowtie$.

If c has the form $\mathtt{edge}(m,n)$ with $m, n \in \mathrm{Node}$, then $c^{g,\alpha} = \mathrm{true}$ if and only if there is an edge in G from $g_V(m)$ to $g_V(n)$. Similarly, if c has the form $\mathtt{edge}(m,n,e)$ with $m, n \in \mathrm{Node}$ and $e \in \mathrm{List}$, then $c^{g,\alpha} = \mathrm{true}$ if and only if there is an edge from $g_V(m)$ to $g_V(n)$ with a label whose list component is $e^{g,\alpha}$.

If c has the form $\mathtt{not}\, c_1$ with $c_1 \in \mathrm{Condition}$, then $c^{g,\alpha} = \mathrm{true}$ if and only if $c_1^{g,\alpha} = \mathrm{false}$. Finally, if c has the form $c_1\, \mathtt{and}\, c_2$ with $c_1, c_2 \in \mathrm{Condition}$, then $c^{g,\alpha} = \mathrm{true}$ if and only if $c_1^{g,\alpha} = \mathrm{true} = c_2^{g,\alpha}$, and if c has the form $c_1\, \mathtt{or}\, c_2$, then $c^{g,\alpha} = \mathrm{true}$ if and only $c_1^{g,\alpha} = \mathrm{true}$ or $c_2^{g,\alpha} = \mathrm{true}$.

We call $r^{g,\alpha} = \langle L^{g,\alpha} \leftarrow K \to R^{g,\alpha}\rangle$ the *instance* of r with respect to g and α, where $L^{g,\alpha}$ and $R^{g,\alpha}$ are obtained from L and R by replacing each label l with $l^{g,\alpha}$. Note that $r^{g,\alpha}$ is a graph transformation rule over $\mathscr{L}$, in the sense of Section 2. We can now define the application of conditional rule schemata to graphs in $\mathscr{G}(\mathscr{L})$.

Definition 4 (Rule-schema application)**.** Given a conditional rule schema $r = \langle L \leftarrow K \to R,\, c\rangle$ and graphs G, H in $\mathscr{G}(\mathscr{L})$, we write $G \Rightarrow_{r,g} H$ (or just $G \Rightarrow_r H$) if there are a premorphism $g\colon L \to G$ and an assignment α such that

(1) g is a graph morphism $L^{g,\alpha} \to G$,

(2) $c^{g,\alpha} = \mathrm{true}$, and

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

Here $G \Rightarrow_{r^{g,\alpha},g} H$ denotes the application of $r^{g,\alpha}$ with match g to G, as defined in Section 2. Note that we use $\Rightarrow$ for the application of both rule schemata and rules, to avoid an inflation of symbols. Given a set $\mathscr{R}$ of conditional rule schemata, we write $G \Rightarrow_{\mathscr{R}} H$ if $G \Rightarrow_r H$ for some conditional rule schema r in $\mathscr{R}$.

[2]The effect of dividing by zero is undefined, that is, left to the implementation.

The following proposition shows that given a rule schema r, a premorphism from the left-hand graph of r to G induces at most one instance of r that can be applied with match g.

Proposition. *Given a conditional rule schema $r = \langle L \leftarrow K \rightarrow R, c\rangle$ and a premorphism $g\colon L \rightarrow G$, there exists at most one assignment α such that g is a graph morphism $L^{g,\alpha} \rightarrow G$.*

The proof of this property relies on the fact that the left-hand graph L contains only simple expressions.

5 Programs

The syntax of graph programs is the same as in GP 1, except for the syntax of rule schemata and the new constructs `try_then_else` and `or`. Figure 7 shows the abstract syntax of GP 2 programs. As before, a program consists of a number of declarations of conditional rule schemata and macros, and exactly one declaration of a main command sequence. The identifiers of category RuleId occurring in a RuleSetCall refer to declarations of conditional rule schemata in category RuleDecl (see previous sections).

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

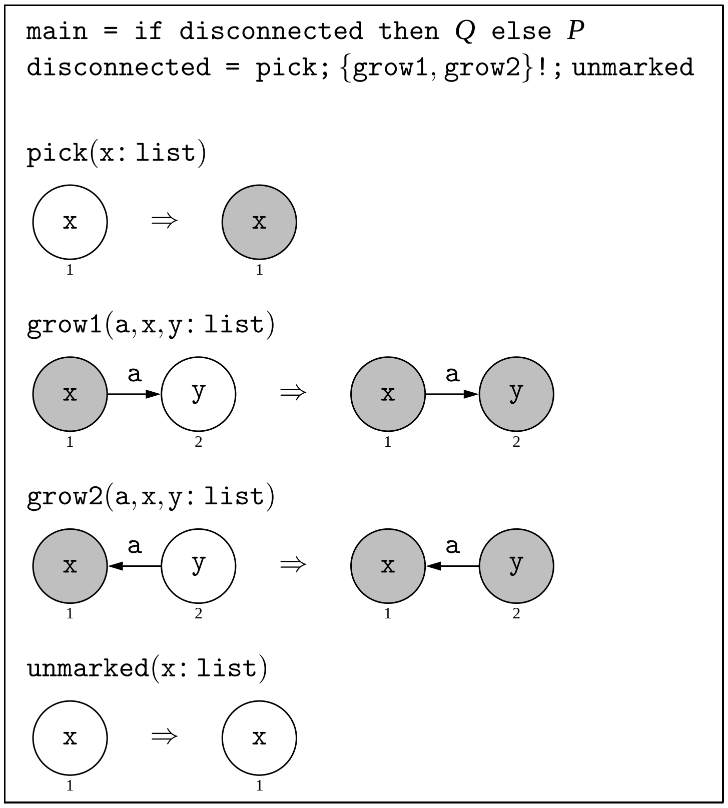
Figure 7: Abstract syntax of programs

In the next section it is shown that the commands `or`, `skip` and `fail` can be expressed through the other commands. Hence the core of GP includes only the call of a set of conditional rule schemata (RuleSetCall), sequential composition (’;’), the if-then-else statement, the try-then-else statement and as-long-as-possible iteration (’!’). Before formally defining the semantics of programs, we discuss some example programs to illustrate the use of the new features of GP 2.

Example 1 (Checking connectedness). A graph is *connected* if there is an undirected path between each two nodes, that is, a sequence of consecutive edges whose directions don’t matter. The program in Figure 8 checks whether an arbitrary input graph G is connected and, depending on the result, executes either program P or program Q on G.

Connectedness is checked by picking some node, marking it, and propagating node marks along edges as long as possible. Then an application of the rule schema `unmarked` tests whether any unmarked nodes are left. If this is the case, then the macro `disconnected` succeeds and program Q is executed, otherwise `disconnected` fails and program P is executed.

It is important to note that P or Q is executed *on the input graph* whereas the graph resulting from the test is discarded. The precise semantics of the branching command is given in Section 6. ☐



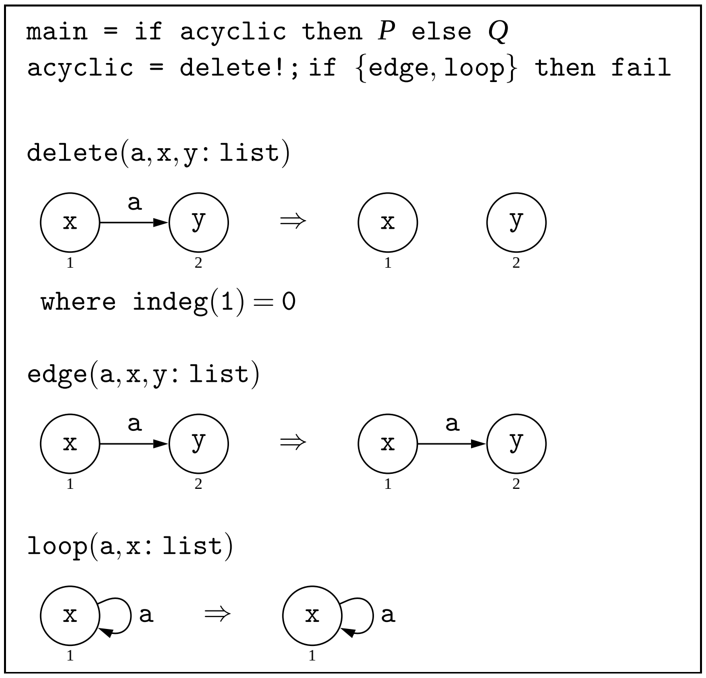
```
main = if disconnected then Q else P
disconnected = pick; {grow1, grow2}!; unmarked

pick(x: list)

grow1(a, x, y: list)

grow2(a, x, y: list)

unmarked(x: list)
```

Figure 8: A program for checking connectedness

Example 2 (Recognising acyclic graphs). A graph is *acyclic* if it does not contain a directed cycle. The program in Figure 9 checks whether an unmarked input graph G is acyclic and, depending on the result, executes either program P or program Q on G.

The absence of cycles is checked by deleting, as long as possible, edges whose source nodes have no incoming edges, and testing subsequently whether any edges remain. This method relies on the following invariant of the rule schema `delete`: for every step $G \Rightarrow_{\texttt{delete}} H$, G is acyclic if and only if H is acyclic. Moreover, a graph to which `delete` is not applicable is acyclic if and only if it does not contain edges. Note that the condition of `delete` uses the new indegree function. ☐

Example 3 (Recognising series-parallel graphs). *Series-parallel* graphs are inductively defined as follows. Every graph G consisting of two nodes connected by an edge is series-parallel, where the edge's source and target are the source and target of G. Given series-parallel graphs G and H, the graphs obtained from the disjoint union $G+H$ by the following two operations are also series-parallel. Serial composition: merge the target of G with the source of H; the source of G becomes the new source and the target of H becomes the new target. Parallel composition: merge the source of G with the source of H, and the target of G with the target of H; sources and targets are preserved.



```
main = if acyclic then P else Q
acyclic = delete!; if {edge, loop} then fail

delete(a, x, y: list)

 where indeg(1) = 0

edge(a, x, y: list)

loop(a, x: list)
```

Figure 9: A program for recognising acyclic graphs

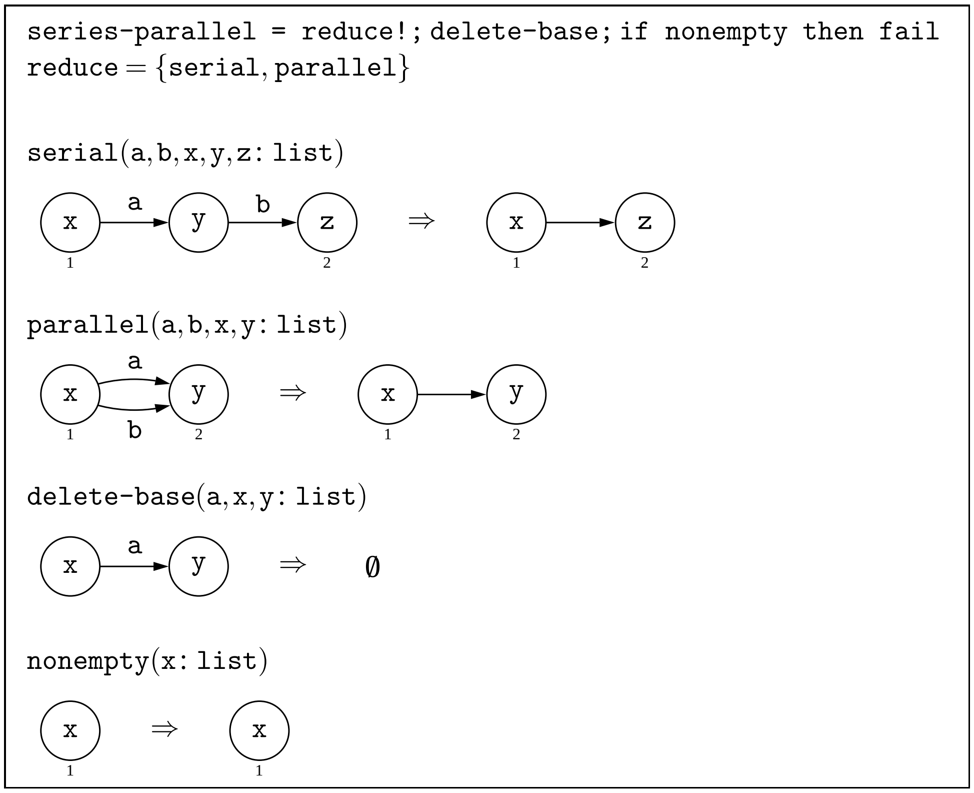
It is known [1, 2] that a graph is series-parallel if and only if it reduces to a graph consisting of two nodes connected by an edge (a *base graph*) by repeated application of the following operations: (a) Given a node with one incoming edge i and one outgoing edge o such that $s(i) \neq t(o)$, replace i, o and the node by an edge from $s(i)$ to $t(o)$. (b) Replace a pair of parallel edges by an edge from their source to their target.

Figure 10 shows a macro which reduces every unmarked series-parallel graph to the empty graph, and fails on every other unmarked graph. The subprogram `reduce!` applies as long as possible the operations (a) and (b) to the input graph G, then the rule schema `delete-base` checks if the result is a base graph whose nodes are not incident to other edges. (The latter is ensured by the dangling condition.) If `delete-base` is not applicable, then the input graph was not reduced to a base graph. In this case the input graph is not series-parallel because every execution of `reduce!` yields the same graph. (This is because the critical pairs of the rule schemata `serial` and `parallel` are strongly joinable [6].)

Finally, after `delete-base` has been applied, the rule schema `nonempty` checks whether the graph resulting from `reduce!` contains nodes other than those of the base graph. The input graph is series-parallel if and only if this is not the case. □

Example 4 (Computing Euler cycles). An *Euler cycle* is a directed cycle of distinct edges that contains all edges and nodes of a graph. A graph is *eulerian* if it contains an Euler cycle. It is known that a graph is eulerian if and only if it is connected and each node has the same indegree as outdegree [1]. Based on this characterisation, the macro `eulerian` of Figure 11 checks whether an unmarked graph is eulerian or not. It does this by using the macro `disconnected` of Figure 8 and the new indegree and outdegree functions.

Given an unmarked eulerian input graph with atomic labels, the program in Figure 12 computes an Euler cycle and numbers its edges. An execution of this program is shown in Figure 13. In the resulting



```
series-parallel = reduce!; delete-base; if nonempty then fail
reduce = {serial, parallel}

serial(a,b,x,y,z: list)

parallel(a,b,x,y: list)

delete-base(a,x,y: list)

nonempty(x: list)
```

Figure 10: A macro for recognising series-parallel graphs

graph, the computed Euler cycle is given by the edges with the labels `1:1`, `1:1:1`, `1:1:2`, `1:1:3`, `1:2`, `1:3` and `1:4`. The graph in the middle of Figure 13 is an intermediate result representing the point in time when the macro `cycle` has been executed for the first time.

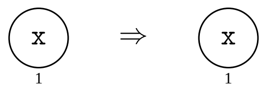
The program uses the command `try_then` to check if the input graph is nonempty. If the input graph is empty, then the empty sequence of edges is an Euler cycle and hence the program returns the empty graph. If the input graph is nonempty, the rule schema `init` picks some node, adds `0` to its label, and marks the node. Then the rule schema `loop` numbers all loops with atomic labels that are incident to the node. Next the rule schema `cycle` numbers a proper (that is, non-loop) cycle starting at this node, by repeatedly applying the rule schema `grow`. Also, at each visited node, `loop` is applied as long as possible to number all incident loops.

When the first proper cycle has been numbered, the subprogram `(next; cycle)!` repeatedly computes a new cycle starting at a node that has already been visited. This cycle is inserted into the current cycle by numbering the new edges with lists that add one position to the list of the edge preceding the new edges. Finally, when all edges of the graph have been numbered, the rule schema `clean-up` removes all auxiliary information in node labels. □

6 Operational Semantics

This section presents a formal semantics for GP 2 in the style of Plotkin's structural operational semantics [5]. As usual for this approach, inference rules inductively define a small-step transition relation $\rightarrow$ on

```
eulerian = if disconnected then fail; if unbalanced then fail
disconnected = ...

unbalanced(x: list)
```



```
where indeg(1) != outdeg(1)
```

Figure 11: A macro for recognising eulerian graphs

configurations. In our setting, a configuration is either a command sequence together with a graph, just a graph or the special element fail:

$$\to \;\subseteq\; (\text{ComSeq} \times \mathscr{G}(\mathscr{L})) \times ((\text{ComSeq} \times \mathscr{G}(\mathscr{L})) \cup \mathscr{G}(\mathscr{L}) \cup \{\text{fail}\}).$$

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

Figure 14 in the Appendix shows the inference rules for the core commands of GP 2. Each rule 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 $\mathscr{G}(\mathscr{L})$. Meta-variables are considered to be universally quantified. For example, the rule [call$_1$] reads: “For all R in RuleSetCall and all G,H in $\mathscr{G}(\mathscr{L})$, $G \Rightarrow_R H$ implies $\langle R,\, G\rangle \to H$.” The transitive and reflexive-transitive closures of $\to$ are written $\to^+$ and $\to^*$, respectively. The notation $G \not\Rightarrow_R$ expresses that for graph G in $\mathscr{G}(\mathscr{L})$ there is no graph H such that $G \Rightarrow_R H$.

The if-then-else command has been designed to “hide” destructive tests. In Example 1, for instance, the test of the if-then-else command produces a graph with marked nodes. By the inference rules [if$_1$] and [if$_2$], this graph is discarded and program P or Q is executed on the input graph. In contrast, a program `try` C `then` P `else` Q passes any graph resulting from its test to P. If test C fails, however, Q is executed on the input graph.

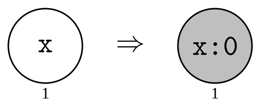
The semantics of the if-then-else command and the as-long-as-possible loop in GP 1 have been modified to allow an efficient implementation. Previously, the conditions of branching commands and the bodies of loops were tested, in the worst case, by trying all possible executions starting from the current graph. This made branching and loop commands impractical for complex tests or large input graphs. In GP 2, the semantics of `if` C `then` P `else` Q, `try` C `then` P `else` Q, and B! do not enforce backtracking when C or B fails. Instead, control is passed to program Q or the loop is terminated, respectively. Note that this change increases the nondeterminism of evaluation in cases where C or B can both succeed and fail on the input graph.

The inference rules for the remaining GP commands are given in Figure 15 of the Appendix. These commands are referred to as *derived* commands because they can be defined by the core commands, as shown below.

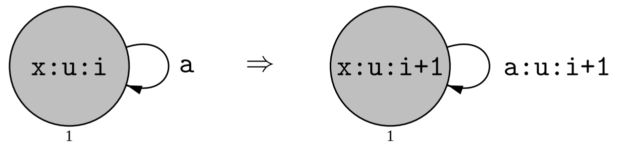
The meaning of GP 2 programs is summarised by the semantic function $[\![\_]\!]$ which assigns to each program P the function $[\![P]\!]$ mapping an input graph G to the set of all possible results of executing P

```
main = try init; loop! then (cycle; (next; cycle)!; clean-up!)
cycle = (grow; loop!)!; unmark
next = first; loop!
```

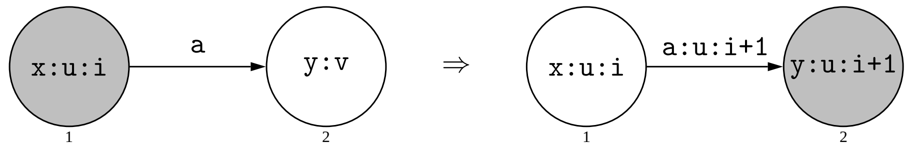
`init(x:atom)`



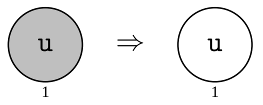
`loop(a,x:atom; u:list; i:int)`



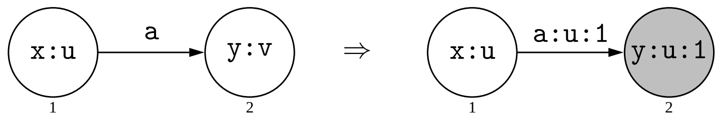
`grow(a,x,y:atom; i:int; u,v:list)`



`unmark(u:list)`

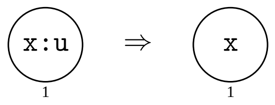


`first(a,x,y:atom; u,v:list)`



where u != empty

`clean-up(x:atom; u:list)`



where u != empty

Figure 12: A program for computing an Euler cycle

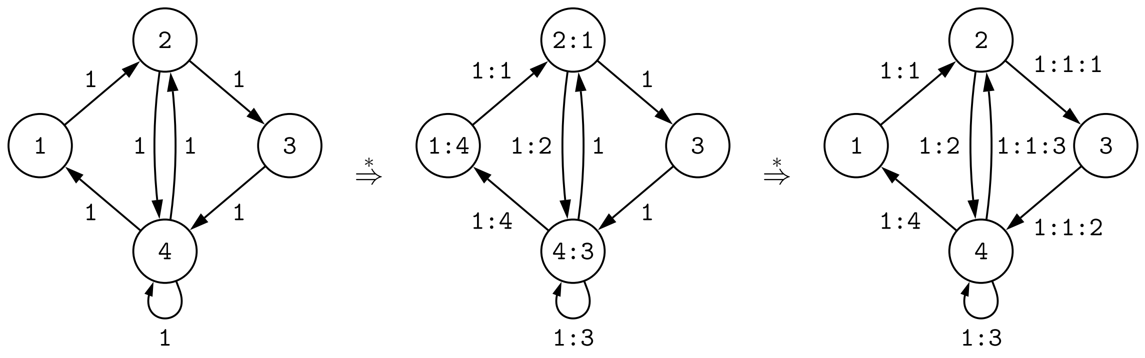


Figure 13: An execution of the program of Figure 12

on G. The application of $[\![P]\!]$ to G is written $[\![P]\!]G$. The result set may contain, besides proper results in the form of graphs, the special values fail and $\perp$. The value fail indicates a failed program run while $\perp$ indicates a run that does not terminate or gets stuck. Program P *can diverge from* G if there is an infinite sequence $\langle P, G\rangle \to \langle P_1, G_1\rangle \to \langle P_2, G_2\rangle \to \ldots$ Also, P *can get stuck from* G if there is a terminal configuration $\langle Q, H\rangle$ such that $\langle P, G\rangle \to^* \langle Q, H\rangle$.

Definition 5 (Semantic function)**.** The *semantic function* $[\![\_]\!]\colon \mathrm{ComSeq} \to (\mathscr{G}(\mathscr{L}) \to 2^{\mathscr{G}(\mathscr{L})\cup\{\mathrm{fail},\perp\}})$ is defined by

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

In the current implementation of GP, reaching the failure state triggers backtracking which then attempts to find a proper result [4]. However, backtracking can be switched off by the user.

A program can get stuck in two situations: (1) it contains a command `if` C `then` P `else` Q or `try` C `then` P `else` Q such that C can diverge from some graph G and can neither produce a proper result from G nor fail from G, or (2) it contains a loop B! whose body B possesses the said property of C. The evaluation of such commands gets stuck because none of the inference rules for if-then-else, try-then-else or iteration is applicable.

The semantic function of Definition 5 suggests a straightforward notion of program equivalence.

Definition 6 (Semantic equivalence)**.** Two programs P and Q are *semantically equivalent*, denoted by $P \equiv Q$, if $[\![P]\!] = [\![Q]\!]$.

For example, it is easy to see that the following equivalences between derived commands and core commands hold (where $\emptyset$ is the empty graph):

- `skip` $\equiv$ `null`, where `null` is the rule schema $\emptyset \Rightarrow \emptyset$;
- `fail` $\equiv \{\}$, where $\{\}$ is the empty set of rule schemata;
- `if` C `then` P $\equiv$ `if` C `then` P `else` `null`, for all programs C and P;
- `try` C `then` P $\equiv$ `try` C `then` P `else` `null`, for all programs C and P.

Less obvious is the following equivalence, showing that `or` is a derived command:

$$P\,\texttt{or}\,Q \equiv \texttt{if remove!; \{create, null\}; zero then}\ P\ \texttt{else}\ Q,$$

for all programs P and Q. Here `remove` is a set of three rule schemata that delete arbitrary edges, loops and isolated nodes, `create` is the rule schema

$$\emptyset \Rightarrow ⓪$$

and `zero` is the rule schema

$$\textcircled{0} \Rightarrow \textcircled{0}.$$

The following non-equivalence may be surprising, too:

$$\texttt{try}\, C \,\texttt{then}\, P \,\texttt{else}\, Q \not\equiv \texttt{if}\, C \,\texttt{then}\, C; P \,\texttt{else}\, Q.$$

To witness, choose $C = \texttt{skip or fail}$, $P = \texttt{skip}$ and $Q = \texttt{skip}$. Then the `try`-program is equivalent to `skip` and hence cannot fail, but the `if`-program can fail.

7 Conclusion

GP allows high-level problem solving in the domain of graphs, by supporting rule-based programming and freeing programmers from dealing with low-level data structures for graphs. The language has a simple syntax and semantics, to facilitate both understanding by programmers and formal reasoning on programs.

The revised language GP 2 has an improved type system, including list variables and subtypes, a new concept of marking nodes and edges graphically, new built-in functions for accessing the indegree and the outdegree of nodes, a more powerful `edge` predicate for conditions, new commands `try-then-else` and `or`, and a simplified semantics of branching and looping to enable an efficient implementation.

Topics for future work include the implementation of GP 2, tool support for Hoare-style program verification [9], and static analyses for properties such as termination and confluence.

Acknowledgements. Parts of this paper were written while visiting Annegret Habel in Oldenburg and Berthold Hoffmann in Bremen in the autumn of 2011. I am grateful for their hospitality. Thanks go also to the Plasma research group in York for helpful comments, especially to Colin Runciman for proposing the concept of shaded nodes.

References

[1] Jørgen Bang-Jensen & Gregory Gutin (2009): *Digraphs: Theory, Algorithms and Applications*, second edition. Springer-Verlag.

[2] R. J. Duffin (1965): *Topology of Series-Parallel Networks*. *Journal of Mathematical Analysis and Applications* 10(2), pp. 303–318, doi:10.1016/0022-247X(65)90125-3.

[3] Annegret Habel & Detlef Plump (2002): *Relabelling in Graph Transformation*. In: *Proc. International Conference on Graph Transformation (ICGT 2002), Lecture Notes in Computer Science* 2505, Springer-Verlag, pp. 135–147, doi:10.1007/3-540-45832-8_12.

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

[5] Gordon D. Plotkin (2004): *A Structural Approach to Operational Semantics*. *Journal of Logic and Algebraic Programming* 60–61, pp. 17–139, doi:10.1016/j.jlap.2004.05.001.

[6] Detlef Plump (2005): *Confluence of Graph Transformation Revisited*. In Aart Middeldorp, Vincent van Oostrom, Femke van Raamsdonk & Roel de Vrijer, editors: *Processes, Terms and Cycles: Steps on the Road to Infinity: Essays Dedicated to Jan Willem Klop on the Occasion of His 60th Birthday, Lecture Notes in Computer Science* 3838, Springer-Verlag, pp. 280–308, doi:10.1007/11601548.

[7] Detlef Plump (2009): *The Graph Programming Language GP*. In: *Proc. International Conference on Algebraic Informatics (CAI 2009)*, *Lecture Notes in Computer Science* 5725, Springer-Verlag, pp. 99–122, doi:10.1007/978-3-642-03564-7_6.

[8] Detlef Plump & Sandra Steinert (2010): *The Semantics of Graph Programs*. In: *Proc. Rule-Based Programming (RULE 2009)*, *Electronic Proceedings in Theoretical Computer Science* 21, pp. 27–38, doi:10.4204/EPTCS.21.3.

[9] Christopher M. Poskitt & Detlef Plump (2012): *Hoare-Style Verification of Graph Programs*. *Fundamenta Informaticae*. To appear.

Appendix: Semantic Inference Rules

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

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

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

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

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

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

Figure 14: Inference rules for core commands

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

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

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

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

Figure 15: Inference rules for derived commands