The Graph Programming Language GP

Detlef Plump

Department of Computer Science
The University of York, UK

Abstract. 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. This paper gives an overview on the GP project. We introduce the language by discussing a sequence of small programming case studies, formally explain conditional rule schemata which are the building blocks of programs, and present a semantics for GP in the style of structural operational semantics. A special feature of the semantics is how it uses the notion of *finitely failing* programs to define powerful branching and iteration commands. We also describe GP's prototype implementation.

1 Introduction

This paper gives an overview on 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 [19]) and thereby frees programmers from handling low-level data structures for graphs. The prototype implementation of GP compiles graph programs into bytecode for an abstract machine, and comes with a graphical editor for programs and graphs.

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 [9]. A major goal for the development of GP is to obtain a practical graph-transformation language that comes with a concise formal semantics, to facilitate program verification and other formal reasoning on programs.

There exist a number of graph-transformation languages and tools, such as PROGRES [24], AGG [6], Fujaba [15], GROOVE [21] and GrGen [8]. But to the best of our knowledge, PROGRES has been the only graph-transformation language with a complete formal semantics so far. The semantics given by Schürr in his dissertation [23], however, reflects the complexity of PROGRES and is in our opinion too complicated to be used for formal reasoning.

For GP, we adopt Plotkin's method of structural operational semantics [18] to define the meaning of programs. This approach is well established for imperative programming languages [16] 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 [3]. 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 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 [19]) to arbitrary programs: the body of a loop can no longer be applied if it finitely fails on the current graph.

The prototype implementation of GP is faithful to the semantics in that it uses backtracking to compute results for input graphs. Hence, for terminating programs a result will be found whenever one exists. In contrast, most other graph-transformation languages (except PROGRES) lack this completeness because their implementations have no backtracking mechanism. The GP system even provides users with the option to generate all possible results of a terminating program.

The rest of this paper is structured as follows. The next section is a brief summary of the graph-transformation formalism underlying GP, the so-called double-pushout approach with relabelling. Section 3 introduces conditional rule schemata and explains their use by interpreting them as sets of conditional rules. In Section 4, graph programs are gently introduced by discussing seven small case studies of problem solving with GP. The section also defines an abstract syntax for graph programs. Section 5 presents a formal semantics for GP in the style of structural operational semantics and discusses some consequences of the semantics. A brief description of the current implementation of GP is given in Section 6. In Section 7, we conclude and mention some topics for future work. Finally, the Appendix defines the natural pushouts on which the double-pushout approach with relabelling is based.

This overview paper is in parts based on the papers [19,20,13].

2 Graph Transformation

We briefly review the model of graph transformation underlying GP, the double-pushout approach with relabelling [10]. Our presentation is tailored to GP in that we consider graphs over a fixed label alphabet and rules in which only the interface graph may contain unlabelled nodes.

GP programs operate on graphs labelled with sequences of integers and strings (the reason for using sequences will be explained in Section 4). Let $\mathbb{Z}$ be the set of integers and Char be a finite set which represents the characters that can be typed on a keyboard. We fix the label alphabet $\mathcal{L} = (\mathbb{Z} \cup \mathrm{Char}^*)^+$ of all nonempty sequences over integers and character strings.

A *partially labelled graph* (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, $l_G \colon V_G \to \mathcal{L}$ is the partial node labelling function and $m_G \colon E_G \to \mathcal{L}$ is the (total) edge labelling function. Given a node v, we write $l_G(v) =\perp$ if $l_G(v)$ is undefined. Graph G is *totally labelled* if l_G is a total function. The set of all totally labelled graphs is denoted by $\mathcal{G}$.

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. More precisely, we have $s_H \circ g_E = g_V \circ s_G$, $t_H \circ g_E = g_V \circ t_G$, $m_H \circ g_E = m_G$, 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), and an *isomorphism* if it is injective, surjective and satisfies $l_H(g_V(v)) = \perp$ for all nodes v with $l_V(v) = \perp$. In this case G and H are *isomorphic*, denoted by $G \cong H$. The *composition* $h \circ g \colon G \to M$ of two morphisms $g \colon G \to H$ and $h \colon H \to M$ is defined componentwise: $h \circ g = \langle h_V \circ g_V,\, h_E \circ g_E \rangle$.

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. We call L the *left-hand side*, R the *right-hand side* and K 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 nodes that are unlabelled in K.

Given graphs G, H in $\mathcal{G}$, a rule $r = (L \leftarrow K \rightarrow R)$, and an injective graph morphism $g \colon L \to G$, a *direct derivation* from G to H by r and g consists of two natural pushouts as in Figure 1. (See the appendix for the definition of natural pushouts.) We write $G \Rightarrow_{r,g} H$ or just $G \Rightarrow_r H$ if there exists such a direct derivation, and $G \Rightarrow_{\mathcal{R}} H$, where $\mathcal{R}$ is a set of rules, if there is some $r \in \mathcal{R}$ such that $G \Rightarrow_r H$.

$$\begin{tikzcd}
L \arrow[d, "g"'] & K \arrow[l] \arrow[r] \arrow[d] & R \arrow[d] \\
G & D \arrow[l] \arrow[r] & H
\end{tikzcd}$$

Fig. 1. A double-pushout

In [10] it is shown that, given r, G and g as above, there exists a direct derivation as in Figure 1 if and only if g satisfies the *dangling condition*: no node in $g(L) - g(K)$ is incident to an edge in $G - g(L)$. In this case D and H are determined uniquely up to isomorphism and can be constructed from G as follows:

1. Remove all nodes and edges in $g(L)-g(K)$. For each $v \in V_K$ with $l_K(v) = \bot$, define $l_D(g_V(v)) = \bot$. The resulting graph is D.
2. Add disjointly to D all nodes and edges from $R - K$, while 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 $v \in V_K$ with $l_K(v) = \bot$, define $l_H(g_V(v)) = l_R(v)$. The resulting graph is H.

To define conditional rules, we follow [5] and equip rules with predicates that constrain the morphisms by which rules can be applied. A *conditional rule* $q = (r, P)$ consists of a rule r and a predicate P on graph morphisms. We require that P is invariant under isomorphic codomains: for a morphism $g\colon L \to G$ and an isomorphism $i\colon G \to G'$, we have $P(g)$ if and only if $P(i \circ g)$. Given a direct derivation $G \Rightarrow_{r,g} H$ such that $P(g)$, we write $G \Rightarrow_{q,g} H$ or just $G \Rightarrow_q H$. For a set of conditional rules $\mathcal{R}$, we write $G \Rightarrow_{\mathcal{R}} H$ if there is some $q \in \mathcal{R}$ such that $G \Rightarrow_q H$.

3 Conditional Rule Schemata

Conditional rule schemata are the "building blocks" of GP, as programs are essentially declarations of such schemata together with a command sequence for controlling their application. Rule schemata generalise rules in that labels may be expressions with parameters of type integer or string. In this section, we give an abstract 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.

In the GP programming system [13], conditional rule schemata are constructed with a graphical editor. We give grammars in Extended Backus-Naur Form for the textual components of such schemata. Figure 3 shows the grammar for node and edge labels in the left and right graph of a rule schema (categories LeftLabel and RightLabel), Figure 4 shows the grammar for conditions (category BoolExp).[1]

Labels can be sequences of expressions separated by underscores, as is demonstrated by examples in Section 4. We require that labels in the left graph must

[1] The grammars are ambiguous, we use parentheses to disambiguate expressions where necessary.

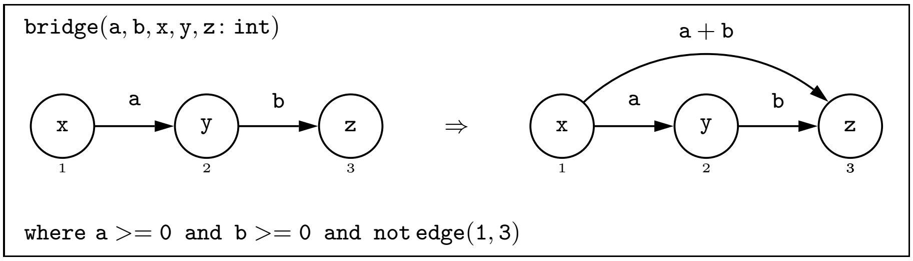

bridge(a, b, x, y, z: int)
a
b
x
y
z
1
2
3
⇒
a + b
where a >= 0 and b >= 0 and not edge(1, 3)

Fig. 2. A conditional rule schema

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 the type of a variable identifier is determined by its declaration and arithmetical operators expect arguments of type `int`.

LeftLabel	::= SimpleExp [’_’ LeftLabel]
RightLabel	::= Exp [’_’ RightLabel]
SimpleExp	::= [’-’] Num | String | VarId
Exp	::= SimpleExp | Exp ArithOp Exp
ArithOp	::= ’+’ | ’-’ | ’*’ | ’/’
Num	::= Digit {Digit}
String	::= ’ ” ’ {Char} ’ ” ’

Fig. 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`

Fig. 4. Syntax of conditions

The condition of a rule schema is a Boolean expression built from expressions of category Exp and the special predicate `edge`, where relational operators have arguments of type `int`. Again, all variable identifiers occurring in the condition must 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 $\mathrm{Var}(D)$, $\mathrm{type}(x) = \mathtt{int}$ implies $\alpha(x) \in \mathbb{Z}$, and $\mathrm{type}(x) = \mathtt{string}$ implies $\alpha(x) \in \mathrm{Char}^*$.

Given a label l of category RightLabel occuring in D and an assignment α, the value $l^\alpha \in \mathcal{L}$ 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 $\oplus$.[2] If l has the form e_m with e in Exp and m in RightLabel, then $l^\alpha = e^\alpha m^\alpha$. 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`, then 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 $G \in \mathcal{G}$. Then for each Boolean subexpression b of c, the value $b^{\alpha,g}$ in $\mathbb{B} = \{\mathtt{tt}, \mathtt{ff}\}$ 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 $b^{\alpha,g} = \mathtt{tt}$ if and only if $e_1^\alpha \bowtie_\mathbb{Z} e_2^\alpha$ where $\bowtie_\mathbb{Z}$ is the relation on integers represented by $\bowtie$. If b has the form $\mathtt{not}\, b_1$ with b_1 in BoolExp, then $b^{\alpha,g} = \mathtt{tt}$ if and only if $b_1^{\alpha,g} = \mathtt{ff}$. 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 $\mathtt{edge}(v, w)$ where v, w are identifiers of interface nodes in D. We then have

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

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 $G \in \mathcal{G}$ and $c^{\alpha,g} = \mathtt{tt}$.

Now 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 $\Rightarrow_r$.

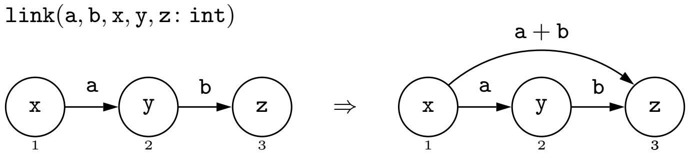
[2] For simplicity, we consider division by zero as an implementation-level issue.

4 Graph Programs

We discuss a number of example programs to familiarize the reader with the features of GP and their use in solving graph problems. At the end of the section, we define the abstract syntax of GP programs.

Example 1 (Transitive closure)
A *transitive closure* of a graph is obtained by inserting an edge between all distinct nodes v and w such that there is a directed path from v to w but no edge. The program `trans_closure` in Figure 5 generates a transitive closure of an integer-labelled input graph by applying the rule schema `link` as long as possible, using the iteration operator '`!`'. In general, arbitrary command sequences can be iterated.

```
main = link!

link(a, b, x, y, z: int)
```



```
where not edge(1, 3)
```

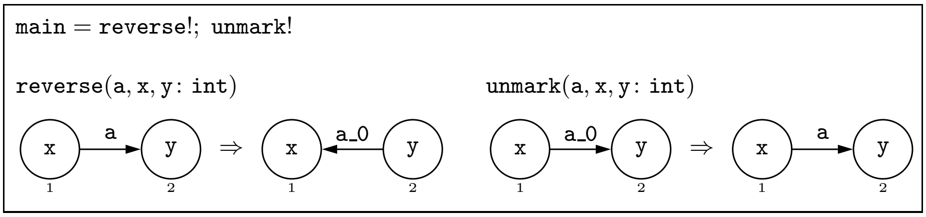
Fig. 5. The program `trans_closure`

The keyword `main` starts the main command sequence of a program to distinguish it from macros (see Example 5). Note that the condition `not edge(1, 3)` of `link` prevents the creation of edges between nodes that are already linked. Without this condition, `trans_closure` could generate parallel edges between nodes 1 and 3 ad infinitum.

By our definition of transitive closure, we can choose any label for the edge created by `link`. Using `a + b` implies that `trans_closure` may produce different results for a given input graph if there are different paths between two nodes. If we want to generate a unique transitive closure, we can replace `a + b` with a constant such as `0`.

Example 2 (Inverse)
The *inverse* of a graph is obtained by reversing the directions of all edges. The program `inverse` in Figure 6 computes the inverse of an integer-labelled input graph in two stages, using the sequential composition of the loops `reverse!` and `unmark!`. The first loop reverses each edge and replaces its label x with the *tagged* label x`_0`, then the second loop removes all tags. In general, arbitrary subprograms can be joined by the semicolon operator.

The underscore operator allows to add a *tag* to a label, used here to mark an edge as having been reversed. In general, a tagged label is a sequence of



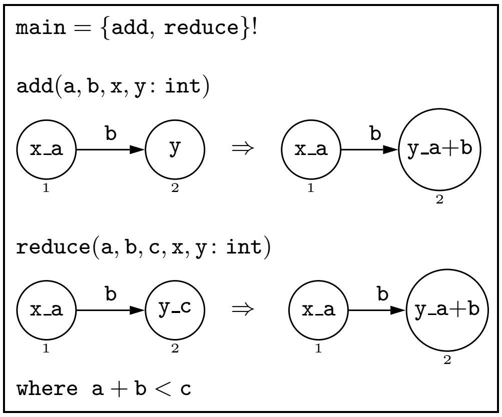
```
main = reverse!; unmark!

reverse(a, x, y: int)                    unmark(a, x, y: int)
```

Fig. 6. The program `inverse`

expressions joined by underscores. Here, we need to mark reversed edges as otherwise the loop `reverse!` would not terminate. Note that the rule schema `reverse` can only be applied to edges with untagged labels.

Example 3 (Shortest distances). Given a graph G whose edge labels are integers, the *distance* of a directed path from a node v to a node w is the sum of the edge labels on that path. If all edge labels in G are nonnegative, then the *shortest distance* from v to w is the minimum of the distances of all paths from v to w.

The program `distances` in Figure 7 expects an integer-labelled input graph where exactly one node v has a tagged label of the form x_`0` and where all edge labels are nonnegative. It adds to each node w that is distinct and reachable from v a tag with the shortest distance from v to w.

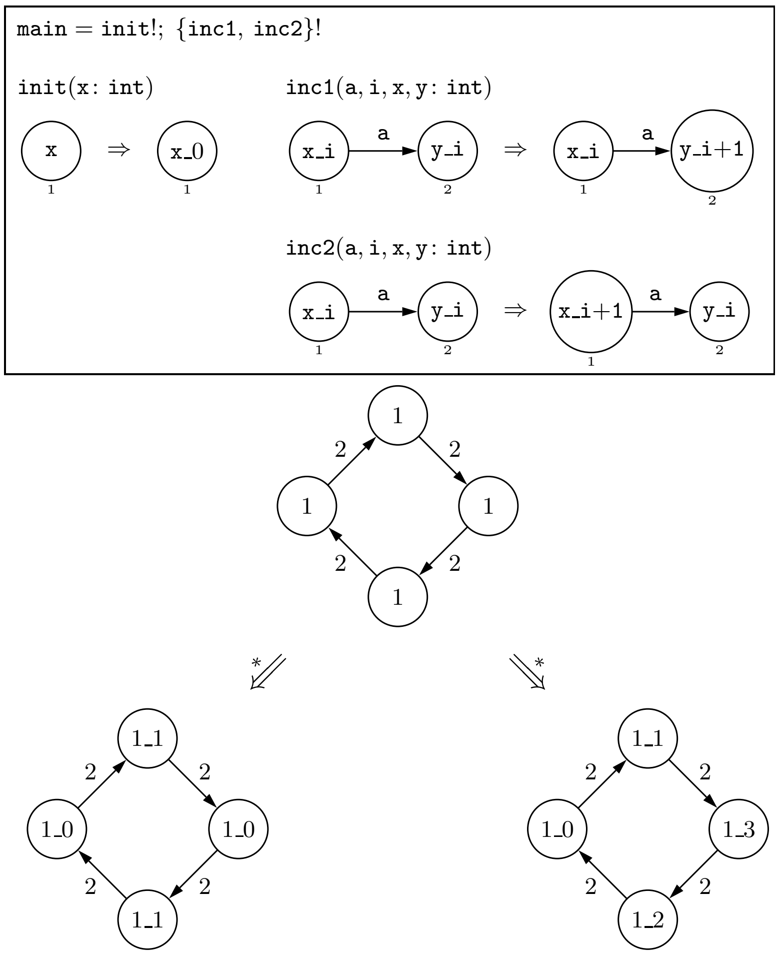


```
main = {add, reduce}!

add(a, b, x, y: int)

reduce(a, b, c, x, y: int)

where a + b < c
```

Fig. 7. The program `distances`

In each iteration of the program's loop, one of the rule schemata `add` and `reduce` is applied to the current graph. If both rule schemata are applicable, one of them is chosen nondeterministically. An equivalent, slightly more deterministic solution is to separate the phases of addition and reduction: `main = add!; reduce!`. A refined version of the program `distances` which implements Dijkstra's shortest-path algorithm can be found in [19].

Example 4 (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. The program `colouring` in Figure 8 produces a colouring for every integer-labelled input graph without loops, recording colours as tags. (Checking for loops would be a straightforward extension which we omit for simplicity.)



```
main = init!; {inc1, inc2}!
```

Fig. 8. The program `colouring` and two of its derivations

The program initially colours each node with zero and then repeatedly increments either the source or the target colour of an edge with the same colour at both ends. Note that this process is highly nondeterministic: Figure 8 shows two different colourings produced for the same input graph, where one is optimal in that it uses only two colours while the other uses four colours. (The problem to generate a colouring with a minimal number of colours is NP-complete [7] and requires a more involved program.)

It is easy to see that whenever `colouring` terminates, the resulting graph is a correctly coloured version of the input graph. For, the output cannot contain an edge with the same colour at both nodes as then `inc1` or `inc2` would have been applied at least one more time. It is less obvious though that the program does terminate for every input graph.

To see that `colouring` always terminates, consider graphs whose node labels are of the form n_i, with $n, i \in \mathbb{Z}$. Given a node v, we denote the tag of its label by $\mathrm{tag}(v)$. Now observe that if G is a graph with $\mathrm{tag}(v) = 0$ for each node v, then for every derivation $G \Rightarrow^*_{\{\mathtt{inc1},\mathtt{inc2}\}} H$ there is some $0 \leq k < V_H$ such that $\mathrm{tag}(V_H) = \{0, 1, \ldots, k\}$ (where some tags may occur repeatedly in H). Thus, by assigning to every graph M the integer $\#M = \sum_{v \in V_M} \mathrm{tag}(v)$, we obtain

$$\#H < 1 + 2 + \cdots + |V_H| = 1 + 2 + \cdots + |V_G|.$$

Since $\#H$ equals the number of rule schema applications in $G \Rightarrow^* H$, it follows that every derivation with `inc1` and `inc2` starting from G must eventually terminate. Moreover, as the upper bound for $\#H$ is quadratic in $|V_G|$, `colouring` always performs at most a quadratic number of rule schema applications.

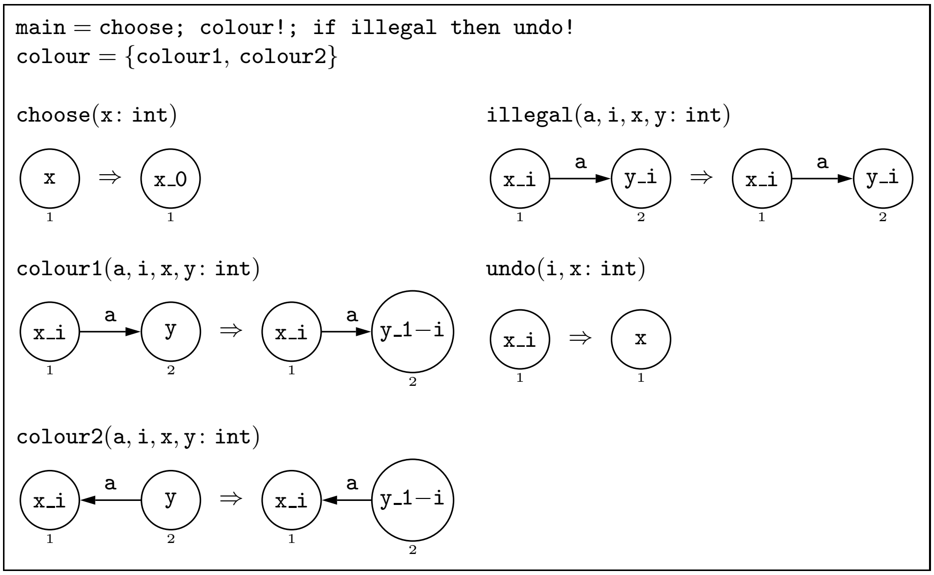
Example 5 (2-Colouring). A graph is *2-colourable* (or *bipartite*) if it possesses a colouring with at most two colours. The program `2-colouring` in Figure 9 generates a 2-colouring for a nonempty and connected input graph if such a colouring exists—otherwise the input graph is returned. The program uses the *macro* `colour` to represent the rule-schema set {`colour1`, `colour2`}.

Given an integer-labelled input graph, first the rule schema `choose` colours an arbitrary node by replacing its label x with x_0. Then the loop `colour!` applies the rule schemata `colour1` and `colour2` 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 [12]). 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.

We can extend `2-colouring`'s applicability to graphs that are possibly empty or disconnected by inserting 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.



```
main = choose; colour!; if illegal then undo!
colour = {colour1, colour2}
```

Fig. 9. The program `2-colouring`

Example 6 (Series-parallel graphs)
The class of *series-parallel* graphs is 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 called *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.

It is known [2,4] that a graph is series-parallel if and only if it reduces to a graph consisting of two nodes connected by an edge 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.

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

```
main = if reduce!; base then P else Q
reduce = {serial, parallel}
```

whose rule schemata `serial`, `parallel` and `base` are shown in Figure 10.

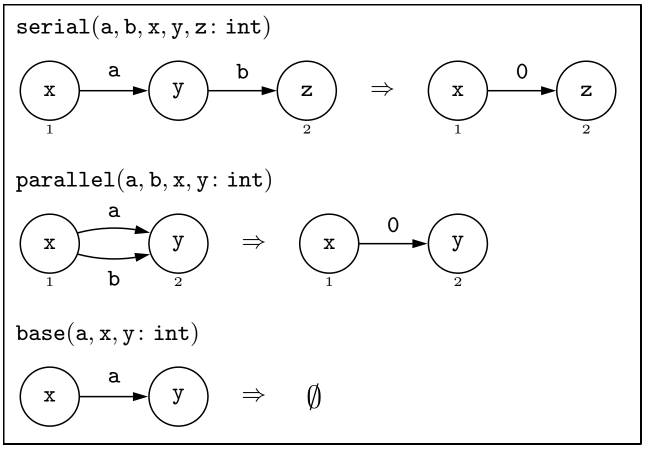


Fig. 10. Rule schemata for recognizing series-parallel graphs

The subprogram `reduce!` applies as long as possible the operations (a) and (b) 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 `reduce!` preserves connectedness and that, by the dangling condition, `base` is applicable only if the images of its left-hand nodes have degree one.) If `base` is applicable, then program P is executed, otherwise program Q. It is important to note that P or Q is executed *on the input graph* G whereas the graph resulting from the test is discarded. The precise semantics of the branching command is given in the next section.

To make the above program usable for possibly disconnected graphs, we can add an if-statement which checks whether the application of `base` has resulted in a nonempty graph:

```
main = if (reduce!; base; if nonempty then fail) then P else Q.
```

Here `nonempty` is a rule schema whose left-hand side is a single interface node, labelled with an integer variable. If `nonempty` is applicable, then the graph resulting from `reduce!` is disconnected and hence the input graph is not series-parallel. In this case `fail` causes the test of the outer if-statement to fail, with the consequence that program Q is executed on the input graph.

Example 7 (Sierpinski triangles). A *Sierpinski triangle* is a self-similar geometric structure which can be recursively defined [17]. Figure 11 shows a Sierpinski triangle of generation three, composed of three second-generation triangles, each of which consists of three triangles of generation one. The triangle and its geometric layout have been generated with the GP programming system [26,13].

The program in Figure 12 expects as input a graph consisting of a single node labelled with the generation number of the Sierpinski triangle to be produced.

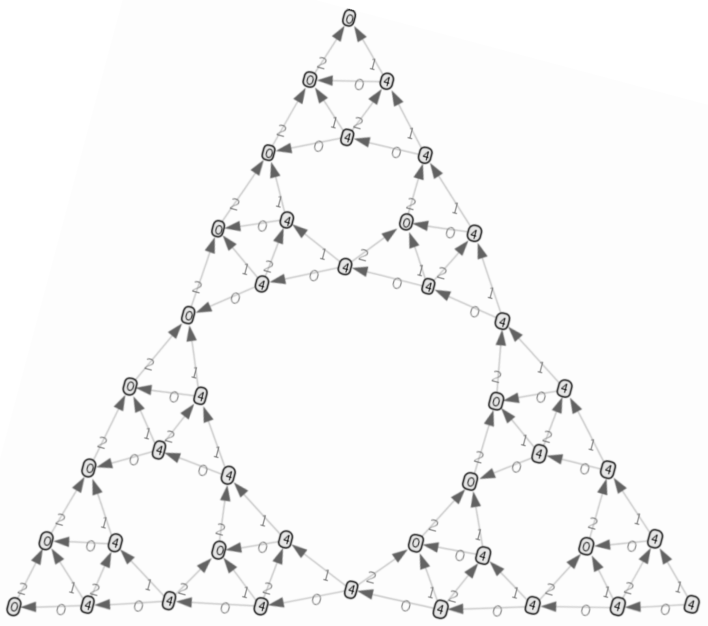


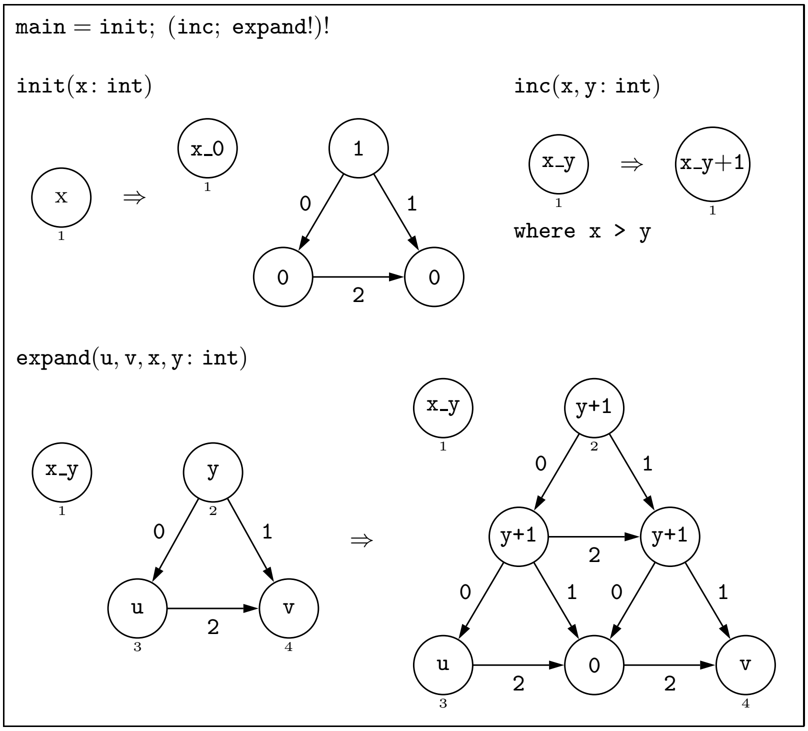
Fig. 11. A Sierpinski triangle (third generation)

The rule schema `init` creates the Sierpinski triangle of generation 0 and turns the input node into a unique "control node" with the tagged label x_0 in order to hold the required generation number x together with the current generation number.

After initialisation, the nested loop (`inc`; `expand`!)! is executed. In each iteration of the outer loop, `inc` increases the current generation number if it is smaller than the required number. The latter is checked by the condition `where x > y`. If the test is successful, the inner loop `expand!` performs a Sierpinski step on each triangle whose top node is labelled with the current generation number: the triangle is replaced by four triangles such that the top nodes of the three outer triangles are labelled with the next higher generation number. The test `x > y` fails when the required generation number has been reached. In this case the application of `inc` fails, causing the outer loop to terminate and return the current graph which is the Sierpinski triangle of the requested generation.

Figure 13 shows the abstract syntax of GP programs.[3] 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,

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



```
main = init; (inc; expand!)!
```

Fig. 12. The program `sierpinski`

```
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
```

Fig. 13. Abstract syntax of GP

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 $\bigcup_{i=1}^{n} \mathrm{I}(r_i)$.

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

This section presents a formal semantics of GP in the style of Plotkin's structural operational semantics [18]. As usual for this approach, inference rules inductively define a small-step transition relation $\rightarrow$ on *configurations*. In our setting, a configuration is either a command sequence together with a graph, just a graph or the special element fail:

$$\rightarrow \;\subseteq\; (\mathrm{ComSeq} \times \mathcal{G}) \times ((\mathrm{ComSeq} \times \mathcal{G}) \cup \mathcal{G} \cup \{\mathrm{fail}\}).$$

Configurations in ComSeq $\times\, \mathcal{G}$ represent unfinished computations, given by a rest program and a state in the form of a graph, while graphs in $\mathcal{G}$ 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 $\gamma \rightarrow \delta$.

Each inference rule in Figure 14 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 $\mathcal{G}$. Given a rule-set call R, let $\mathrm{I}(R) = \bigcup\{\mathrm{I}(r) \mid r \text{ is a rule-schema identifier in } \mathcal{R}\}$ (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 $\mathcal{G}$ such that $G \Rightarrow_{\mathrm{I}(R)} H$ for some graph H. Meta-variables are considered to be universally quantified. For example, the rule [Call$_1$] should be read as: "For all R in RuleSetCall and all G, H in $\mathcal{G}$, $G \Rightarrow_{\mathrm{I}(R)} H$ implies $\langle R,\, G\rangle \rightarrow H$."

Figure 14 shows the inference rules for the core constructs of GP. We write $\rightarrow^+$ and $\rightarrow^*$ for the transitive and reflexive-transitive closures of $\rightarrow$. A command sequence C *finitely fails* on a graph $G \in \mathcal{G}$ if (1) there does not exist an infinite sequence $\langle C,\, G\rangle \rightarrow \langle C_1,\, G_1\rangle \rightarrow \ldots$ and (2) for each terminal configuration γ such that $\langle C,\, G\rangle \rightarrow^* \gamma$, $\gamma = \mathrm{fail}$. In other words, C finitely fails on G if all computations starting from $(C,\, G)$ eventually end in the configuration fail.

The concept of finite failure stems from logic programming where it is used to define *negation as failure* [3]. 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. This is demonstrated by Example 6 in the previous section, where the test of the if-then-else command reduces input graphs as much as possible by the rule schemata `serial` and `parallel`, followed

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

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

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

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

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

Fig. 14. Inference rules for core commands

by an application of `base`. By the inference rules [If$_1$] and [If$_2$], the resulting graph is discarded and program P or Q is executed *on the input graph.*

The meaning of the remaining GP commands is defined in terms of the meaning of the core commands, see Figure 15. We refer to these commands as *derived* commands.

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

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

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

Fig. 15. Inference rules for derived commands

Figure 16 shows a simple example of program evaluation by the transition relation $\to$. It demonstrates that for the same input graph, a program may compute an output graph, reach the failure state or diverge.

We 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. To this end, let the *semantic function* $[\![\_]\!]\colon \mathrm{ComSeq} \to (\mathcal{G} \to 2^{\mathcal{G}\cup\{\perp\}})$ be defined by[4]

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

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 $\langle P,\, G\rangle \to^* \langle Q,\, H\rangle$.

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

```
main = {r1, r2}; {r1, r2}; r1!
r1          r2
```

$\textcircled{1} \Rightarrow \textcircled{1} \qquad \textcircled{1} \Rightarrow \textcircled{2}$

$$\begin{tikzcd}
\langle \texttt{main}, \textcircled{1} \rangle \arrow[r] \arrow[d] & \langle P, \textcircled{2} \rangle \arrow[r] & \text{fail} & \\
\langle P, \textcircled{1} \rangle \arrow[r] \arrow[d] & \langle \texttt{r1!}, \textcircled{1} \rangle \arrow[r] & \langle \texttt{r1!}, \textcircled{1} \rangle \arrow[r] & \dots \\
\langle \texttt{r1!}, \textcircled{2} \rangle \arrow[d] & & & \\
\textcircled{2} & & &
\end{tikzcd}$$

where $P = \{\texttt{r1}, \texttt{r2}\}; \texttt{r1!}$

Fig. 16. Nondeterminism in program evaluation

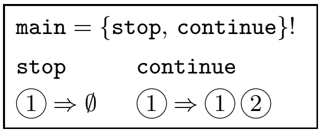
The element fail is not considered as a result of running a program and hence does not occur in result sets. In the current implementation of GP, reaching the failure state triggers backtracking which attempts to find a proper result (see next section). Note that a program P finitely fails on a graph G if and only if $[\![P]\!]G = \emptyset$. In Example 6, for instance, we have $[\![\texttt{reduce!}; \texttt{base}]\!]G = \emptyset$ for every connected graph G containing a cycle. This is because the graph resulting from `reduce!` 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 such subprograms gets stuck because the inference rules for branching resp. iteration are not applicable.

Next we consider programs that produce infinitely many (non-isomorphic) results for some input. A simple example for such a program is given in Figure 17. GP programs showing this behaviour on some input can necessarily diverge from that input. This property is known as *bounded nondeterminism* [22].

Proposition (Bounded nondeterminism). *Let P be a program and G a graph in $\mathcal{G}$. If P cannot diverge from G, then $[\![P]\!]G$ is finite up to isomorphism.*

The reason is that for every configuration γ, the set $\{\delta \mid \gamma \to \delta\}$ is finite up to isomorphism of the graphs in configurations. In particular, the constraints on the syntax of conditional rule schemata ensure that for every rule schema r and every graph G in $\mathcal{G}$, there are up to isomorphism only finitely many graphs H such that $G \Rightarrow_{\mathrm{I}(r)} H$.



```
main = {stop, continue}!
```

$$[\![\texttt{main}]\!]\,\textcircled{1} = \{\bot,\ \emptyset,\ \textcircled{2},\ \textcircled{2}\textcircled{2},\ \textcircled{2}\textcircled{2}\textcircled{2},\ \ldots\}$$

Fig. 17. Infinitely many results for the same input

An important role of a formal semantics is to provide a rigorous notion of program equivalence. We call two programs P and Q *semantically equivalent*, denoted by $P \equiv Q$, if $[\![P]\!] = [\![Q]\!]$. For example, the following equivalences hold for arbitrary programs C, P, P_1, P_2 and Q:

(1) $P;\ \texttt{skip} \equiv P \equiv \texttt{skip}; P$
(2) $\texttt{fail}; P \equiv \texttt{fail}$
(3) $\texttt{if}\ C\ \texttt{then}\ (P_1;\ Q)\ \texttt{else}\ (P_2;\ Q) \equiv (\texttt{if}\ C\ \texttt{then}\ P_1\ \texttt{else}\ P_2);\ Q$
(4) $P! \equiv \texttt{if}\ P\ \texttt{then}\ (P;\ P!)$

On the other hand, there are programs P such that

$$P;\ \texttt{fail} \not\equiv \texttt{fail}.$$

For, if P can diverge from some graph G, then $[\![P;\ \texttt{fail}]\!]G$ contains $\bot$ whereas $[\![\texttt{fail}]\!]G$ is empty.

6 Implementation

This section briefly describes the current implementation of GP, consisting of a graphical editor for programs and graphs, a compiler, and the York Abstract Machine (YAM). Figure 18 shows how these components interact, where GXL is the Graph Exchange Language [27] and YAMG is an internal graph format.

The graphical editor allows graph and program loading, editing and saving, and program execution on a given graph. Figure 19 shows a screenshot of the graphical editor, where the rule schema `expand` of the program `sierpinski` from Example 7 is being edited. The editor is implemented in Java and uses the prefuse data visualisation library [11], which provides automatic graph layout by a force-directed algorithm. The Sierpinski triangle of Figure 11, for example, was generated by this algorithm.

The York abstract machine (YAM) manages the graph on which a GP program operates, by executing low-level graph operations in bytecode format. The current graph is stored in a complex data structure which is designed to make graph interrogation very quick (at the cost of slightly slower graph updates). Typical query operations are "provide a list of all edges whose target is node n" and "provide a list of all nodes whose (possibly tagged) label has value 0 at position 1".

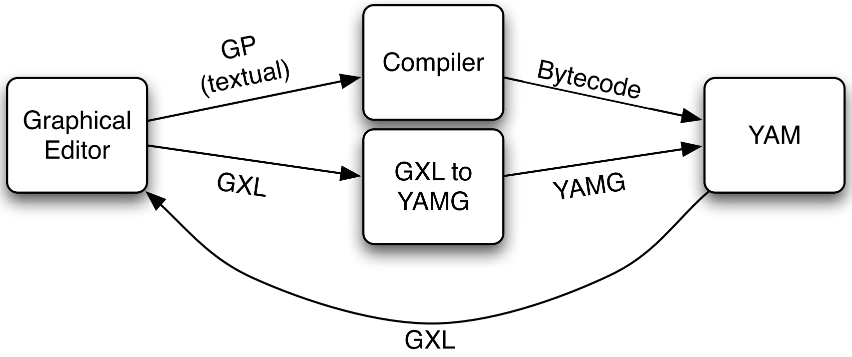
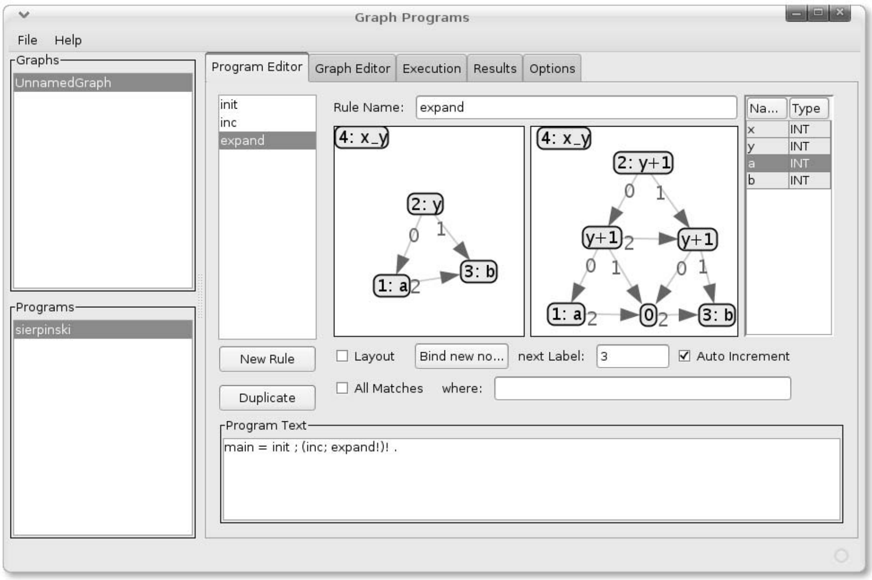


Fig. 18. Components of the GP system



Na...	Type
x	INT
y	INT
a	INT
b	INT

```
main = init ; (inc; expand!)! .
```

Fig. 19. A screenshot of the graphical editor

The YAM is similar to Warren's abstract machine for Prolog [1] in that it handles GP's nondeterminism by backtracking, using a mixed stack of choice points and environment frames. Choice points consist of a record of the number of graph changes at their creation time, a program position to jump to if failure occurs when the choice point is the highest on the stack, and pointers to the previous choice and containing environment. The number of graph changes is recorded so that they can be undone during backtracking: using the stack of graph changes, the graph is recreated as it was at the choice point. Environment frames have a set of registers to store label elements or graph item identities, and an associated function and program position in the bytecode. They also show which environment and program position to return to.

The YAM provides instructions for handling nondeterminism by which the compiler constructs helper functions to implement backtracking. Nondeterministic choice between a set of rule schemata is handled by trying them in textual order until one succeeds. Before each is tried, the failure behaviour is configured to try the next. Nondeterministic choice between graph-item candidates for a match is handled by choosing and saving the first element, and on failure, using the saved previous answer to return and save the next element.

For efficiency reasons, the YAM is implemented in C. See also [14], where a more detailed description of (a slightly older version of) the YAM and its bytecode instructions is given.

The GP compiler is written in Haskell. It converts textually represented GP programs into YAM bytecode by translating each individual rule schema into a sequence of instructions for graph matching and transformation. These sequences are then composed by YAM function calls according to the meaning of GP's control constructs.

The compiler generates code for graph matching by decomposing each rule schema into a searchplan of node lookups, edge lookups (find an edge whose source and target have not been found yet) and extensions (find an edge whose source or target has been found). The choice and order of these search operations is determined by a list of priorities. For example, finding source or target of an edge that has already been found has higher priority (because it is cheaper) than finding an edge between nodes that have already been found. Searchplan generation is a common technique for graph matching and is also used in the implementations of PROGRES [28], Fujaba [15] and GrGen [8].

The semantics of GP assigns to an input graph of a program *all* possible output graphs. This is taken seriously by the implementation in that it provides users with the option to generate all results of a terminating program. (There is no guarantee of completeness for programs that can diverge, because the search for results uses a depth-first strategy.) In contrast, other graph-transformation languages do not fully exploit the nondeterministic nature of graph transformation. For example, AGG [6] makes its nondeterministic choices randomly, with no backtracking. Similarly, Fujaba has no backtracking. PROGRES [24] seems to be the only other graph-transformation language that provides backtracking.

7 Conclusion

We have demonstrated that GP is a 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 operational semantics 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.

The prototype implementation of GP is faithful to the semantics and computes a result for a (terminating) program whenever possible. It even provides the option to generate all possible results.

Future extensions of GP may include recursive procedures for writing complicated algorithms (see [25]) and a type concept for restricting the shape of graphs. A major goal is to support formal reasoning on graph programs. We plan to develop static analyses for properties such as termination and confluence (uniqueness of results), and a calculus and tool support for program verification.

Acknowledgements. I am grateful to Sandra Steinert and Greg Manning for their contributions to the design and implementation of GP [19,25,14,13,20].

References

1. Aït-Kaci, H.: Warren's Abstract Machine: A Tutorial Reconstruction. MIT Press, Cambridge (1991)
2. Bang-Jensen, J., Gutin, G.: Digraphs: Theory, Algorithms and Applications. Springer, Heidelberg (2000)
3. Clark, K.L.: Negation as failure. In: Gallaire, H., Minker, J. (eds.) Logic and Data Bases, pp. 293–322. Plenum Press (1978)
4. Duffin, R.J.: Topology of series-parallel networks. Journal of Mathematical Analysis and Applications 10, 303–318 (1965)
5. Ehrig, H., Habel, A.: Graph grammars with application conditions. In: Rozenberg, G., Salomaa, A. (eds.) The Book of L, pp. 87–100. Springer, Heidelberg (1986)
6. Ermel, C., Rudolf, M., Taentzer, G.: The AGG approach: Language and environment. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) Handbook of Graph Grammars and Computing by Graph Transformation, vol. 2, ch. 14, pp. 551–603. World Scientific, Singapore (1999)
7. Garey, M.R., Johnson, D.S.: Computers and Intractability. W.H. Freeman and Company, New York (1979)
8. Geiß, R., Batz, G.V., Grund, D., Hack, S., Szalkowski, A.: GrGen: A fast SPO-based graph rewriting tool. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 383–397. Springer, Heidelberg (2006)
9. Habel, A., Plump, D.: Computational completeness of programming languages based on graph transformation. In: Honsell, F., Miculan, M. (eds.) FOSSACS 2001. LNCS, vol. 2030, pp. 230–245. Springer, Heidelberg (2001)
10. Habel, A., Plump, D.: Relabelling in graph transformation. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 135–147. Springer, Heidelberg (2002)
11. Heer, J., Card, S.K., Landay, J.A.: Prefuse: A toolkit for interactive information visualization. In: Proc. SIGCHI Conference on Human Factors in Computing Systems (CHI 2005), pp. 421–430. ACM Press, New York (2005)

12. Kleinberg, J., Tardos, É.: Algorithm Design. Addison Wesley, Reading (2006)
13. Manning, G., Plump, D.: The GP programming system. In: Proc. Graph Transformation and Visual Modelling Techniques (GT-VMT 2008). Electronic Communications of the EASST, vol. 10 (2008)
14. Manning, G., Plump, D.: The York abstract machine. In: Proc. Graph Transformation and Visual Modelling Techniques (GT-VMT 2006). Electronic Notes in Theoretical Computer Science, vol. 211, pp. 231–240. Elsevier, Amsterdam (2008)
15. Nickel, U., Niere, J., Zündorf, A.: The FUJABA environment. In: Proc. International Conference on Software Engineering (ICSE 2000), pp. 742–745. ACM Press, New York (2000)
16. Nielson, H.R., Nielson, F.: Semantics with Applications: An Appetizer. Springer, Heidelberg (2007)
17. Peitgen, H.-O., Jürgens, H., Saupe, D.: Chaos and Fractals, 2nd edn. Springer, Heidelberg (2004)
18. Plotkin, G.D.: A structural approach to operational semantics. Journal of Logic and Algebraic Programming 60–61, 17–139 (2004)
19. Plump, D., Steinert, S.: Towards graph programs for graph algorithms. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 128–143. Springer, Heidelberg (2004)
20. Plump, D., Steinert, S.: The semantics of graph programs (submitted for publication, 2009)
21. Rensink, A.: The GROOVE simulator: A tool for state space generation. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 479–485. Springer, Heidelberg (2004)
22. Reynolds, J.C.: Theories of Programming Languages. Cambridge University Press, Cambridge (1998)
23. Schürr, A.: Operationales Spezifizieren mit programmierten Graphersetzungssystemen. Deutscher Universitäts-Verlag (1991) (in German)
24. Schürr, A., Winter, A., Zündorf, A.: The PROGRES approach: Language and environment. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) Handbook of Graph Grammars and Computing by Graph Transformation, vol. 2, ch. 13, pp. 487–550. World Scientific, Singapore (1999)
25. Steinert, S.: The Graph Programming Language GP. PhD thesis, The University of York (2007)
26. Taentzer, G., Biermann, E., Bisztray, D., Bohnet, B., Boneva, I., Boronat, A., Geiger, L., Geiß, R., Horvath, Á., Kniemeyer, O., Mens, T., Ness, B., Plump, D., Vajk, T.: Generation of Sierpinski triangles: A case study for graph transformation tools. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE 2007. LNCS, vol. 5088, pp. 514–539. Springer, Heidelberg (2008)
27. Winter, A.J., Kullbach, B., Riediger, V.: An overview of the GXL graph exchange language. In: Diehl, S. (ed.) Dagstuhl Seminar 2001. LNCS, vol. 2269, pp. 324–336. Springer, Heidelberg (2002)
28. Zündorf, A.: Graph pattern matching in PROGRES. In: Cuny, J., Engels, G., Ehrig, H., Rozenberg, G. (eds.) Graph Grammars 1994. LNCS, vol. 1073, pp. 454–468. Springer, Heidelberg (1996)

Appendix: Natural Pushouts

This appendix defines the natural pushouts on which direct derivations are based (see Section 2) and characterises them in terms of ordinary pushouts. Further properties can be found in [10].

$$\begin{tikzcd}
K \arrow[r, "b"] \arrow[d, "d"'] & R \arrow[d] \\
D \arrow[r] & H
\end{tikzcd}$$

Fig. 20. A pushout diagram

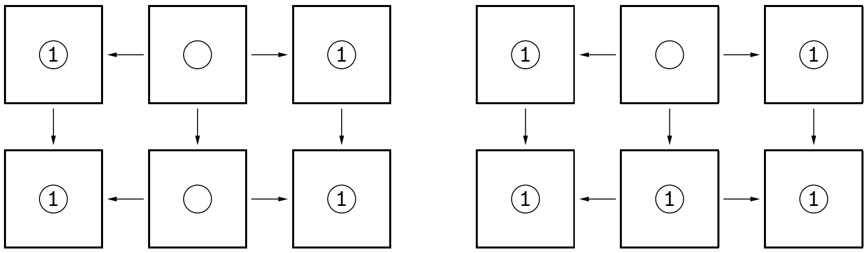


Fig. 21. A natural and a non-natural double-pushout

A diagram of morphisms between partially labelled graphs as in Figure 20 is a *pushout* if the following conditions are satisfied:[5]

- Commutativity: $K \to R \to H = K \to D \to H$.
- Universal property: For every pair of graph morphisms $\langle R \to H', D \to H' \rangle$ such that $K \to R \to H' = K \to D \to H'$, there is a unique morphism $H \to H'$ such that $R \to H' = R \to H \to H'$ and $D \to H' = D \to H \to H'$.

The diagram is a *pullback* if commutativity holds and the following universal property:

- For every pair of graph morphisms $\langle K' \to R, K' \to D \rangle$ such that $K' \to R \to H = K' \to D \to H$, there is a unique morphism $K' \to K$ such that $K' \to R = K' \to K \to R$ and $K' \to D = K' \to K \to D$.

A pushout is *natural* if it is simultaneously a pullback.

[5] Given graph morphisms $f\colon A \to B$ and $g\colon B \to C$, we write $A \to B \to C$ for the composition $g \circ f\colon A \to C$.

Proposition (Characterisation of natural pushouts [10]). *If b is injective, then the pushout in Figure 20 is natural if and only if for all $v \in V_K$,*

$$l_K(v) = \bot \textit{ implies } l_R(b_V(v)) = \bot \textit{ or } l_D(d_V(v)) = \bot.$$

For example, the double-pushout on the left of Figure 21 consists of natural pushouts, the double-pushout on the right consists of non-natural pushouts.