A Small-Step Operational Semantics for GP 2

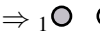
Brian Courtehoute and Detlef Plump
Department of Computer Science, University of York, York, UK
{bc956,detlef.plump}@york.ac.uk

The operational semantics of a programming language is said to be small-step if each transition step is an atomic computation step in the language. A semantics with this property faithfully corresponds to the implementation of the language. The previous semantics of the graph programming language GP 2 is not fully small-step because the loop and branching commands are defined in big-step style. In this paper, we present a truly small-step operational semantics for GP 2 which, in particular, accurately models diverging computations. To obtain small-step definitions of all commands, we equip the transition relation with a stack of host graphs and associated operations. We prove that the new semantics is non-blocking in that every computation either diverges or eventually produces a result graph or the failure state. We also show the finite nondeterminism property, viz. that each configuration has only a finite number of direct successors. The previous semantics of GP 2 is neither non-blocking nor does it have the finite nondeterminism property. We also show that, for a program and a graph that terminate, both semantics are equivalent, and that the old semantics can be simulated with the new one.

1 Introduction

GP 2 is a nondeterministic programming language based on graph transformation rules. The previous semantics of GP 2 is defined by both small-step and big-step inference rules [17]. An operational semantics is *small-step* if atomic computation steps in the language correspond to transition steps, meaning that the language can be implemented by translating the transition steps into corresponding code. In this paper, we present a truly small-step operational semantics for GP 2 which, in particular, accurately models diverging computations.

While the previous semantics (Figure 3) has small-step elements, the branching and loop constructs are not small-step. This can lead to the semantic transition sequence *blocking* or *getting stuck* [14], i.e. reaching a configuration which is neither a graph nor the failure state, such that no inference rule is applicable.

To illustrate this situation, consider the program `P = try (r1!) then skip else skip`, with the rule `r1` :   , applied to the host graph . The statement `r1!` means that the rule `r1` is called until it is no longer applicable. The `try` statement attempts to evaluate `r1!` but will neither branch to the `then` nor the `else` part because the loop `r1!` diverges on . In the previous semantics, `try` statements are handled with the following inference rules :

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

The premises of these inference rules are that the conditional part C of a `try` statement applied to host graph G results in either a graph H or failure, which determines whether P or Q is called. If $\langle C, G\rangle$ diverges (does not terminate) however, neither rule applies. Since there are no other `try` rules, the transition sequence gets stuck.

The new semantics we introduce in this paper handles `try` statements with the following rules:

B. Hoffmann and M. Minas (Eds.): Twelfth International Workshop on Graph Computation Models (GCM 2021)
EPTCS 350, 2021, pp. 89–110, doi:10.4204/EPTCS.350.6

$$[\text{try}_1]\ \langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q, S\rangle \to \langle \text{TRY}(C,P,Q), \text{push}(\text{top}(S), S)\rangle$$

$$[\text{try}_2]\ \frac{\langle C, S\rangle \to \langle C', S'\rangle}{\langle \text{TRY}(C,P,Q), S\rangle \to \langle \text{TRY}(C',P,Q), S'\rangle} \qquad [\text{try}_3]\ \frac{\langle C, S\rangle \to S'}{\langle \text{TRY}(C,P,Q), S\rangle \to \langle P, \text{pop2}(S')\rangle}$$

$$[\text{try}_4]\ \frac{\langle C, S\rangle \to \text{fail}}{\langle \text{TRY}(C,P,Q), S\rangle \to \langle Q, \text{pop}(S)\rangle}$$

Here S and S' are stacks of graphs. The rule $[\text{try}_1]$ duplicates the top of the stack, and the TRY construct signals that the copy operation has happened. Repeated applications of the inference rule $[\text{try}_2]$ model the evaluation of the condition in a small-step fashion. If the condition loops, $[\text{try}_2]$ can be applied indefinitely, and we get an infinite transition sequence.

Intuitively, `P` should loop, which is what happens in the implementation of GP 2. In the previous semantics however, `P` gets stuck because `r1!` diverges, which means that we cannot apply either of the inference rules $[\text{try}'_1]$ or $[\text{try}'_2]$ to resolve the `try` statement.

The previous semantics tries to remedy this issue in the *semantic function* which associates to a program `P` and host graph G the set $[\texttt{P}]G$ of all possible outcomes of the execution of `P` on G. These outcomes can be a graph, the element fail, or $\perp$ which represents an infinite transition sequence. The previous semantic function uses $\perp$ as an outcome if the transition sequence gets stuck. However, there are problems with this approach.

Consider the program `P = try Loop then skip else skip`, where `Loop = {r1,r2}!`, `r1` is as previously defined, and `r2` : ○ ⇒ ∅. The command `{r1,r2}` is a *rule set call*, meaning that rules `r1` and `r2` are selected nondeterministically. When `P` is executed on the host graph ○, an application of `r2` causes the loop to terminate since it removes the marked node which is necessary for either rule to be applicable. Hence `r1` may be applied a number of times, and then `r2` is applied once. But it should also be possible that `r2` is never called, resulting in a diverging computation. Hence the set of outcomes we want is $\{\perp, \emptyset, ○, ○○, ○○○, \dots\}$. According to the previous semantics, however, the execution of `P` on ○ cannot get stuck since `Loop` *can* always transition to a graph; and by the rules $[\text{try}'_1]$ and $[\text{try}'_2]$, the execution cannot diverge either. So $\perp \notin [\texttt{P}]○ = \{\emptyset, ○, ○○, ○○○, \dots\}$. Now, the new semantics indeed corresponds exactly to our intuition of the operational behaviour of GP 2 programs. Moreover, we conjecture that the implementation is sound with respect to the new semantics, in that the behaviour of the implementation is covered by the new semantics.

This may also lead to two programs being semantically equivalent, even though they should not be. Programs P and P' are *semantically equivalent* if $[P] = [P']$, i.e. they have the same outcomes for all host graphs. Consider the program `P = try ({r3,r2}!) then skip else skip`, where `r3` : $_1○ \Rightarrow {}_1○$. It can diverge but is semantically equivalent to `Q = try r2 then skip else skip` since the previous semantics cannot detect that divergence. For instance, $[\texttt{P}]○ = [\texttt{Q}]○ = \{\emptyset\}$, but $[\texttt{P}]○$ should include $\perp$.

The aforementioned issues can also happen with `if` statements, which work similarly to `try` statements, except that the changes the condition made to the host graph are reversed, even if the evaluation of the condition succeeds. Nested loops such as `Loop!` can get stuck as well since their inference rules also assume that the loop body either results in a graph or fails.

Diverging computations not being modelled properly entails an undesirable property, namely *infinite nondeterminism*, i.e. there can be infinitely many configurations reachable in a single transition step. Consider the program `P = try Loop then skip else skip`, where `Loop = {r1,r2}!`, and the rules are as previously defined. We have $[\texttt{Loop}]○ = \{\emptyset, ○, ○○, ○○○, \dots, \perp\}$. In a transition sequence starting with $\langle \texttt{P}, ○\rangle$, since the `try` statement is resolved within a single step, it only takes one step to transition to either of the graphs in the set $\{\emptyset, ○, ○○, ○○○, \dots\}$, of which there are infinitely many.

The semantics we introduce in this paper is truly small-step and as such, it accurately models looping computations with diverging transition sequences. When starting with a valid GP 2 program, it cannot get stuck, which is a property we call *non-blocking*. As a consequence of the small-step approach, we get *finite nondeterminism*, meaning we can only reach a finite number of configurations within a single transition step.

In Section 2, we give a brief overview of the rule-based graph programming language GP 2 along with the previous semantics. We propose the new semantics in Section 3 and give examples of transition sequences. In Section 4 we prove several properties of the new semantics, including non-blocking as well as finite nondeterminism, and define the semantic function along with semantic equivalence. In Section 5, we compare the new and previous semantics by showing the new semantic function is an extension of the previous one, and that they are equivalent excluding divergence.

2 The Graph Programming Language GP 2

This section provides a brief introduction to GP 2 [16], a nondeterministic graph programming language based on transformation rules. We show the abstract syntax of GP 2 programs below, and refer to [3] for the full syntax. The language is implemented by a compiler generating C code [4].

GP 2 programs transform input graphs into output graphs, where graphs are labelled and directed and may contain parallel edges and loops.

The principal programming construct in GP 2 are conditional graph transformation rules labelled with expressions. For example, Figure 1 shows a program recognising graphs that contain cycles and the declaration of its rules. The rule `delete` which has three formal parameters, a left-hand graph and a right-hand graph which are specified graphically, and a textual condition starting with the keyword `where`. The small numbers attached to nodes are identifiers, all other text in the graphs are labels.

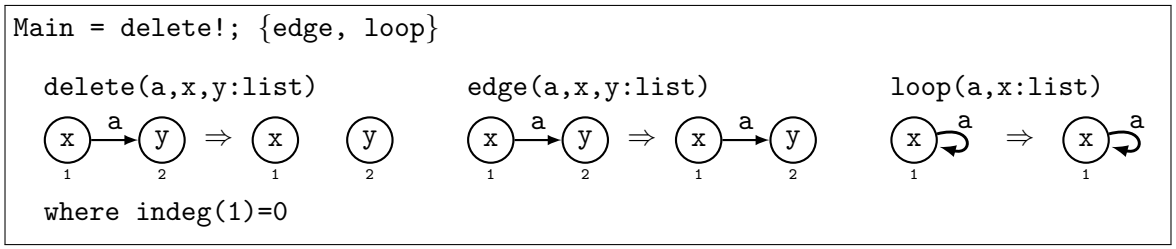


Figure 1: GP 2 program recognising cyclic graphs

GP 2 labels consist of an expression and an optional mark (explained below). Expressions are of type `int`, `char`, `string`, `atom` or `list`, where `atom` is the union of `int` and `string`, and `list` is the type of a (possibly empty) list of atoms. Lists of length one are equated with their entries and hence every expression can be considered as a list.

The concatenation of two lists x and y is written x:y, the empty list is denoted by `empty`. Character strings are enclosed in double quotes. Composite arithmetic expressions such as `n * n` must not occur in the left-hand graph, and all variables occurring in the right-hand graph or the condition must also occur in the left-hand graph.

Besides carrying list expressions, nodes and edges can be *marked*. For example, one of the nodes in rule r1 in the introduction is marked by a grey shading. Marks are convenient to highlight items in

```
Program         ::= Declaration { Declaration }
Declaration     ::= MainDecl | ProcedureDecl | RuleDecl
MainDecl        ::= Main '=' CommandSeq
ProcedureDecl   ::= ProcedureID '=' [ '[' LocalDecl ']' ] CommandSeq
LocalDecl       ::= ( RuleDecl | ProcedureDecl ) { LocalDecl }
CommandSeq      ::= Command {';'Command}
Command         ::= Block
                  | if Block then Block [ else Block ]
                  | try Block [ then Block ] [ else Block ]
Block           ::= '(' CommandSeq ')' ['!']
                  | SimpleCommand
                  | Block or Block
SimpleCommand   ::= RuleSetCall ['!']
                  | ProcedureCall ['!']
                  | break
                  | skip
                  | fail
RuleSetCall     ::= RuleID | '{' [ RuleID { ',' RuleID } ] '}'
ProcedureCall   ::= ProcedureID
```

Figure 2: GP 2 Program Syntax

input or output graphs, and to record visited items during a graph traversal. For instance, a graph can be checked for connectedness by propagating marks along edges as long as possible and subsequently testing whether any unmarked nodes remain. Note that conventional graph algorithms are often described by using marks as a visual aid [7].

Additionally, nodes in rules and host graphs can be *rooted*. If such a node appears in the left-hand side of a rule, it can only be matched with a root node in the host graph. Their use restricts matching to the neighbourhoods of root nodes, which can greatly increase efficiency [6].

We do not elaborate any further on features such as marks or roots because the GP 2 semantics does not depend on them.

Rules operate on *host graphs* which are labelled with constant values (lists containing integer and string constants). Applying a rule $L \Rightarrow R$ to a host graph G works roughly as follows: (1) Replace the variables in L and R with constant values and evaluate the expressions in L and R, to obtain an instantiated rule $\hat{L} \Rightarrow \hat{R}$. (2) Choose a subgraph S of G isomorphic to $\hat{L}$ such that the dangling condition and the rule's application condition are satisfied (see below). (3) Replace S with $\hat{R}$ as follows: numbered nodes stay in place (possibly relabelled), edges and unnumbered nodes of $\hat{L}$ are deleted, and edges and unnumbered nodes of $\hat{R}$ are inserted.

In this construction, the *dangling condition* requires that nodes in S corresponding to unnumbered nodes in $\hat{L}$ (which should be deleted) must not be incident with edges outside S. The rule's application condition is evaluated after variables have been replaced with the corresponding values of $\hat{L}$, and node identifiers of L with the corresponding identifiers of S. For example, the term `indeg(1)=0` in the condition of `delete` in Figure 1 forbids the node $g(1)$ to have incoming edges, where $g(1)$ is the node in S corresponding to `1`.

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

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

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

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

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

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

$$[\text{alap}'_3]\ \frac{\langle P, G\rangle \rightsquigarrow^* \langle \texttt{break}, H\rangle}{\langle P!, G\rangle \rightsquigarrow H} \qquad [\text{break}']\ \langle \texttt{break};P, G\rangle \rightsquigarrow \langle \texttt{break}, G\rangle$$

(a) Inference rules for core commands

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

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

$$[\text{if}'_3]\ \langle \texttt{if}\ C\ \texttt{then}\ P, G\rangle \rightsquigarrow \langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else skip}, G\rangle$$

$$[\text{try}'_3]\ \langle \texttt{try}\ C\ \texttt{then}\ P, G\rangle \rightsquigarrow \langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else skip}, G\rangle$$

$$[\text{try}'_4]\ \langle \texttt{try}\ C\ \texttt{else}\ P, G\rangle \rightsquigarrow \langle \texttt{try}\ C\ \texttt{then skip else}\ P, G\rangle$$

$$[\text{try}'_5]\ \langle \texttt{try}\ C, G\rangle \rightsquigarrow \langle \texttt{try}\ C\ \texttt{then skip else skip}, G\rangle$$

(b) Inference rules for derived commands

Figure 3: Previous GP 2 Semantics

Formally, GP 2 is based on a form of attributed graph transformation according to the so-called double-pushout approach [11, 9]. The grammar in Figure 2 gives the abstract syntax of GP 2 programs. A program consists of declarations of conditional rules and procedures, and exactly one declaration of a main command sequence. The category RuleID refers to declarations of conditional rules in RuleDecl (whose syntax is omitted). Procedures must be non-recursive, they can be seen as macros with local declarations.

The call of a rule set $\{r_1, \ldots, r_n\}$ nondeterministically applies one of the rules whose left-hand graph matches a subgraph of the host graph such that the dangling condition and the rule's application condition are satisfied. The call *fails* if none of the rules is applicable to the host graph.

The command `if` C `then` P `else` Q is executed on a host graph G by first executing C on G. If this results in a graph, P is executed on the original graph G; otherwise, if C fails, Q is executed on G. The `try` command has a similar effect, except that P is executed on the result of C's execution in case C succeeds.

The loop command $P!$ executes the body P repeatedly until it fails. When this is the case, $P!$ terminates with the graph on which the body was entered for the last time. The `break` command inside a loop terminates that loop with the current graph and transfers control to the command following the loop.

A program P `or` Q non-deterministically chooses to execute either P or Q, which can be simulated by a rule-set call and the other commands [16]. The commands `skip` and `fail` can also be expressed by the other commands: `skip` is equivalent to an application of the rule $\emptyset \Rightarrow \emptyset$ (where $\emptyset$ is the empty graph) and `fail` is equivalent to an application of $\{\}$ (the empty rule set).

Like Plotkin's structural operational semantics [15], the previous GP 2 semantics is given by inference rules. The rules in Figure 3 define the transition relation $\rightsquigarrow$ over the following set:

$$(\text{ComSeq} \times \mathscr{G}) \times ((\text{ComSeq} \times \mathscr{G}) \cup \mathscr{G} \cup \{\text{fail}\}).$$

Here $\mathscr{G}$ is the set of all GP 2 host graphs and ComSeq is the set of command sequences as defined in the syntax (Figure 2), where we assume that procedure IDs have been eliminated by macro expansion. This means that procedure IDs have been replaced with their defining command sequence, and name clashes arising from local declarations have been resolved by renaming.

The element fail represents the program resulting in a failure state. The inference rules contain universally quantified variables, namely host graphs G and H, command sequences in ComSeq C, P, P', and Q, and rule set call R. The transitive closure of $\rightsquigarrow$ is denoted by $\rightsquigarrow^+$, and the reflexive transitive closure by $\rightsquigarrow^*$.

In general, the execution of a program on a host graph may result in another graph, fail, or diverge. Also, executions can get *stuck* in that they reach a non-terminal configuration (neither a graph nor fail) to which no inference rule is applicable. Let $\mathscr{G}$ be the set of all host graphs and $\mathscr{G}^{\oplus} = \mathscr{G} \cup \{\bot, \text{fail}\}$. These outcomes are described by the semantic function $[\_] : \text{ComSeq} \to (\mathscr{G} \to 2^{\mathscr{G}^{\oplus}})$ which, for a command sequence P and a host graph G, is defined as

$$[P]G = \{X \in \mathscr{G} \cup \{\text{fail}\} \mid \langle P, G\rangle \rightsquigarrow^+ X\} \cup \{\bot \mid P \text{ can diverge or get stuck from } G\}.$$

By divergence we mean non-termination, that is the existence of an infinite transition sequence starting in $\langle P, G\rangle$.

3 The Small-Step Semantics

In this section, we introduce an improved semantics defined by inference rules, and give examples of transition sequences.

Due to additional constructs, the new semantics needs to distinguish between command sequences that are valid GP 2 programs, and command sequences that are intermediary. The former are members of CommandSeq from the syntax in Figure 2, and are called *command sequences*. They have to satisfy the context conditions specified in Appendix A.6 of Bak's thesis [3]. The following condition is particularly relevant to this paper: "A `break` must be enclosed within a loop. If a `break` is in the condition of a branching statement[1], the enclosing loop must be within the same condition." This constraint is not specific to graph programs: Java, C, and Python have similar restrictions on the use of `break` statements.

We define *extended command sequences* (set ExtComSeq) to be command sequences with additional auxiliary constructs ITE and TRY. They do not follow context conditions since we may want a `break` outside of a loop in an intermediary transition step. The ITE and TRY statements serve to advance the command sequence in the condition in a small-step fashion, as well as to maintain the stack of host graphs. When we enter an ITE or TRY statement, the top of the stack (and current host graph) is duplicated in order to keep a backup. When exiting these statements we either pop the top, modified graph, or the second graph on the stack which is the unmodified backup copy depending on the outcome of the condition. The stack structure is needed because `if` and `try` statements may be nested. Whenever we enter an ITE or TRY construct, we push a graph, and whenever we exit one, we pop a graph. This ensures that the stack always contains enough graphs to pop and that the current host graph is always on top.

The rules in Figure 4 inductively define a transition relation $\to$ over the following set:

$$(\text{ExtComSeq} \times \mathscr{S}) \times ((\text{ExtComSeq} \times \mathscr{S}) \cup \mathscr{S} \cup \{\text{fail}\}),$$

where $\mathscr{S}$ is the set of all stacks of GP 2 host graphs (explained below). We call an element of the set $(\text{ExtComSeq} \times \mathscr{S}) \cup \mathscr{S} \cup \{\text{fail}\}$ an *extended configuration*, whereas $(\text{CommandSeq} \times \mathscr{S}) \cup \mathscr{S} \cup \{\text{fail}\}$ is the set of *configurations*. A configuration (or extended configuration) C is *terminal* if $C = \text{fail}$ or $C = S$ for some graph stack S.

The set $\mathscr{S}$ is the set of all non-empty stacks of GP 2 host graphs where the top element is the current host graph, and where the other elements are backup copies to revert to or discard after the resolution of conditions of branching statements. Such a stack $S = [G_1, G_2, G_3, \ldots, G_n]$ is a finite ordered list of GP 2 host graphs with unary operations $\text{top}(S) = G_1$, $\text{pop}(S) = [G_2, G_3, \ldots, G_n]$ and $\text{pop2}(S) = [G_1, G_3, \ldots, G_n]$, as well as the binary operation $\text{push}(G, S) = [G, G_1, G_2, \ldots, G_n]$, where G is a GP 2 host graph.

Most of the inference rules in Figure 4 have a horizontal bar. These rules consist of a *premise* above the bar and a *conclusion* below. The conclusion defines a transition step provided that the premise holds. A rule without a bar is called an *axiom* and can be applied to a configuration without any precondition.

There are several universally quantified meta-variables within the inference rules. P, P', Q, Q', C, and C' stand for extended command sequences in ExtComSeq, S stands for a graph stack in $\mathscr{S}$, G represents a host graph, and R represents a rule set. We denote the transitive closure of $\to$ by $\to^+$, and the reflexive transitive closure by $\to^*$.

The inference rules inductively define the transition relation $\to$. The rules $[\text{call}_1]$ and $[\text{call}_2]$ are base cases. Their premises are GP 2 derivations. Which of the two premises is satisfied depends on whether

[1] By branching statement, we mean an `if`, `try`, ITE, or TRY statement.

$$[\text{call}_1]\ \frac{\text{top}(S) \Rightarrow_R G}{\langle R, S\rangle \to \text{push}(G, \text{pop}(S))} \qquad [\text{call}_2]\ \frac{\text{top}(S) \not\Rightarrow_R}{\langle R, S\rangle \to \text{fail}}$$

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

$$[\text{seq}_3]\ \frac{\langle P, S\rangle \to \text{fail}}{\langle P; Q, S\rangle \to \text{fail}} \qquad [\text{break}]\ \langle \texttt{break}; P, S\rangle \to \langle \texttt{break}, S\rangle$$

$$[\text{alap}_1]\ \langle P!, S\rangle \to \langle \texttt{try}\ P\ \texttt{then}\ P!\ \texttt{else}\ \texttt{skip}, S\rangle \qquad [\text{alap}_2]\ \langle \text{TRY}(\texttt{break}, P!, \texttt{skip}), S\rangle \to \text{pop2}(S)$$

$$[\text{if}_1]\ \langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q, S\rangle \to \langle \text{ITE}(C, P, Q), \text{push}(\text{top}(S), S)\rangle$$

$$[\text{try}_1]\ \langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q, S\rangle \to \langle \text{TRY}(C, P, Q), \text{push}(\text{top}(S), S)\rangle$$

$$[\text{if}_2]\ \frac{\langle C, S\rangle \to \langle C', S'\rangle}{\langle \text{ITE}(C, P, Q), S\rangle \to \langle \text{ITE}(C', P, Q), S'\rangle} \qquad [\text{try}_2]\ \frac{\langle C, S\rangle \to \langle C', S'\rangle}{\langle \text{TRY}(C, P, Q), S\rangle \to \langle \text{TRY}(C', P, Q), S'\rangle}$$

$$[\text{if}_3]\ \frac{\langle C, S\rangle \to S'}{\langle \text{ITE}(C, P, Q), S\rangle \to \langle P, \text{pop}(S')\rangle} \qquad [\text{try}_3]\ \frac{\langle C, S\rangle \to S'}{\langle \text{TRY}(C, P, Q), S\rangle \to \langle P, \text{pop2}(S')\rangle}$$

$$[\text{if}_4]\ \frac{\langle C, S\rangle \to \text{fail}}{\langle \text{ITE}(C, P, Q), S\rangle \to \langle Q, \text{pop}(S)\rangle} \qquad [\text{try}_4]\ \frac{\langle C, S\rangle \to \text{fail}}{\langle \text{TRY}(C, P, Q), S\rangle \to \langle Q, \text{pop}(S)\rangle}$$

(a) Inference rules for core commands

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

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

$$[\text{if}_5]\quad \langle \texttt{if}\ C\ \texttt{then}\ P, S\rangle \to \langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ \texttt{skip}, S\rangle$$

$$[\text{try}_5]\quad \langle \texttt{try}\ C\ \texttt{then}\ P, S\rangle \to \langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ \texttt{skip}, S\rangle$$

$$[\text{try}_6]\quad \langle \texttt{try}\ C\ \texttt{else}\ P, S\rangle \to \langle \texttt{try}\ C\ \texttt{then}\ \texttt{skip}\ \texttt{else}\ P, S\rangle$$

$$[\text{try}_7]\quad \langle \texttt{try}\ C, S\rangle \to \langle \texttt{try}\ C\ \texttt{then}\ \texttt{skip}\ \texttt{else}\ \texttt{skip}, S\rangle$$

(b) Inference rules for derived commands

Figure 4: Improved GP 2 Semantics

$\text{top}(S) \Rightarrow_R G$ or $\text{top}(S) \not\Rightarrow_R$, i.e. whether a rule in the rule set can be applied to the current host graph or not. The `if` and `try` statements are modelled by the $[\text{if}_i]$ and $[\text{try}_i]$ rules.

Sequential composition of commands is covered by $[\text{seq}_1]$, $[\text{seq}_2]$, and $[\text{seq}_3]$, covering the cases of whether the first command called on a host graph results in a configuration, a graph stack, or fail.

Loops are semantically described as a try statement in $[\text{alap}_1]$. Calling a command sequence as long as possible is modelled by trying to apply the command sequence, and if it succeeds, keep applying it as long as possible. Breaking from a loop is handled by [break], which makes sure commands following the break are discarded, and $[\text{alap}_2]$, which terminates the loop if there is an isolated break in the TRY condition.

Figure 4a shows the inference rules for the core commands of GP 2, while Figure 4b gives the inference rules for derived commands such as `or`, `skip`, and `fail`, as well as some `if` and `try` statements with omitted `then` and `else` clauses. These commands are referred to as *derived* commands because they can be defined by the core commands (see [16] for the case of the previous semantics).

Let us look at a couple of examples of transition sequences in Figure 5, the first to illustrate loops, and the second to illustrate `if` and `try` statements. For each transition, we note the applied inference rule as a subscript. If the conclusion of $[\text{rule}_1]$ is used as a premise for $[\text{rule}_2]$, we denote it by $\frac{[\text{rule}_1]}{[\text{rule}_2]}$.

Example 3.1. Consider the program `P=r!` and the rule `r` : $_1 \circ\!\!\rightarrow\!\!\circ \Rightarrow_1 \circ$. Let us examine a transition sequence of `P` applied to the graph $\circ\!\!\rightarrow\!\!\circ\!\!\rightarrow\!\!\circ$, as seen in Figure 5a.

We start by applying $[\text{alap}_1]$ which turns the loop into a `try` statement. Unlike in the previous semantics, we model a loop by trying to apply its body, and if it is successful, we call the loop again.

The inference rule $[\text{try}_1]$ transforms the `try` statement into the auxiliary TRY construct, which advances the program in a small-step fashion, unlike the previous semantics. There is a similar ITE construct which models `if` statements. The top of the graph stack is duplicated since the changes made by the condition of the `try` may be discarded.

We then apply `r` to the current host graph (top of the stack) so $[\text{call}_1]$ can be applied. This serves as a premise for $[\text{try}_3]$, which ends the TRY statement, pops the second element of the stack, and moves on to the `then` part which is the original loop.

We repeat this process until `r` is no longer applicable to the host graph. At this point, $[\text{call}_2]$ serves as the premise for $[\text{try}_4]$ which exits the TRY statement. This time, the condition results in fail, so we move on to the `else` part which is `skip` and the loop terminates.

Now consider program `P'=try(if (r1;r1) then (r1;r1))` and the rule `r1` : $_1 \circ\!\!\rightarrow\!\!\circ \Rightarrow_1 \circ$. A transition sequence of `P'` applied to host graph $\circ\!\!\leftarrow\!\!\circ\!\!\rightarrow\!\!\circ$ can be found in Figure 5b.

Since the `try` statement does not have a `then` or `else` part, we first apply $[\text{try}_7]$, which adds `skip` as both the `then` and `else` parts.

The inference rule $[\text{try}_1]$ turns the `try` statement into the auxiliary TRY statement and duplicates the top of the stack. For most of the remaining transition sequence, we apply $[\text{try}_2]$ under various premises to advance the condition.

Since the `if` has no `else` part, $[\text{if}_5]$ completes it with a `skip`. The `if` statement is then turned into the auxiliary ITE statement, duplicating the top of the stack once again.

The rule `r1` is applied to the host graph which advances the concatenation with $[\text{seq}_2]$, the ITE with $[\text{if}_2]$, and the TRY with $[\text{try}_2]$. Calling `r1` a second time resolves the ITE, and the top of the stack is popped since changes made by the conditions of `if` statements are reversed.

We keep applying the condition of the TRY, until we resolve it with $[\text{try}_3]$. This time the second graph on the stack is popped since changes made by the condition of a `try` that did not result in fail are

$\langle$ `r!`, [O→O→O]$\rangle$

$\to_{[\text{alap}_1]}$ $\langle$ `try r then r! else skip`, [O→O→O]$\rangle$

$\to_{[\text{try}_1]}$ $\langle$TRY (`r`, `r!`, `skip`), [O→O→O, O→O→O]$\rangle$

$\to_{\substack{[\text{call}_1]\\[\text{try}_3]}}$ $\langle$ `r!`, [O→O]$\rangle$

$\to_{[\text{alap}_1]}$ $\langle$ `try r then r! else skip`, [O→O]$\rangle$

$\to_{[\text{try}_1]}$ $\langle$TRY (`r`, `r!`, `skip`), [O→O, O→O]$\rangle$

$\to_{\substack{[\text{call}_1]\\[\text{try}_3]}}$ $\langle$ `r!`, [O]$\rangle$

$\to_{[\text{alap}_1]}$ $\langle$ `try r then r! else skip`, [O]$\rangle$

$\to_{[\text{try}_1]}$ $\langle$TRY (`r`, `r!`, `skip`), [O, O]$\rangle$

$\to_{\substack{[\text{call}_2]\\[\text{try}_4]}}$ $\langle$ `skip`, [O]$\rangle$

$\to_{[\text{skip}]}$ [O]

(a) Transition sequence of program `P` applied to graph O→O→O.

$\langle$ `try(if(r1;r1) then(r1;r1))`, [O←O→O]$\rangle$

$\to_{[\text{try}_7]}$ $\langle$ `try (if (r1;r1) then (r1;r1)) then skip else skip`, [O←O→O]$\rangle$

$\to_{[\text{try}_1]}$ $\langle$TRY(`if (r1;r1) then (r1;r1)`, `skip`, `skip`), [O←O→O, O←O→O]$\rangle$

$\to_{\substack{[\text{if}_5]\\[\text{try}_2]}}$ $\langle$TRY(`if (r1;r1) then (r1;r1) else skip`, `skip`, `skip`), [O←O→O, O←O→O]$\rangle$

$\to_{\substack{[\text{if}_1]\\[\text{try}_2]}}$ $\langle$TRY(ITE(`r1;r1`, `r1;r1`, `skip`), `skip`, `skip`), [O←O→O, O←O→O, O←O→O]$\rangle$

$\to_{\substack{[\text{call}_1]\\[\text{seq}_2]\\[\text{if}_2]\\[\text{try}_2]}}$ $\langle$TRY(ITE(`r1`, `r1;r1`, `skip`), `skip`, `skip`), [O→O, O←O→O, O←O→O]$\rangle$

$\to_{\substack{[\text{call}_1]\\[\text{if}_3]\\[\text{try}_2]}}$ $\langle$TRY(`r1;r1`, `skip`, `skip`), [O←O→O, O←O→O]$\rangle$

$\to_{\substack{[\text{call}_1]\\[\text{try}_2]}}$ $\langle$TRY(`r1`, `skip`, `skip`), [O→O, O←O→O]$\rangle$

$\to_{\substack{[\text{call}_1]\\[\text{try}_3]}}$ $\langle$ `skip`, [O]$\rangle$

$\to_{[\text{skip}]}$ [O]

(b) Transition sequence of program `P'` applied to graph O←O→O.

Figure 5: Examples of transition sequences

preserved. □

4 Properties of the Semantics

In this section, we show that the semantics is non-blocking, i.e. if a transition sequence ends in an extended configuration, we can always apply an inference rule (Proposition 4.3). Note that we can only guarantee the non-blocking property for extended configurations that are part of a transition sequence originating in a valid GP 2 program. We call those *reachable* extended configurations. This is reasonable because there can be no other types of configurations in a transition sequence modelling a GP 2 program.

Furthermore, we will describe the outcomes of a transition sequence starting with a valid GP 2 program (Proposition 4.5), and show that we have finite nondeterminism (Proposition 4.7), i.e. there are only finitely many one-step transitions starting from any configuration, and what it means for the semantic function.

Let us first look at a lemma that guarantees we can make a transition step from extended configurations that do not contain a `break`, which is the first step towards showing the non-blocking property.

Lemma 4.1 (Progress from Extended Configurations)**.** Let $\langle P, S\rangle$ be an extended configuration. Then one of the following applies:

- $\langle P, S\rangle \to \langle P', S'\rangle$ for some extended configuration $\langle P', S'\rangle$.
- $\langle P, S\rangle \to S'$ for some graph stack $S' \in \mathscr{S}$.
- $\langle P, S\rangle \to$ fail.
- P is not a command sequence and contains a `break`.

Proof. We shall prove this lemma by going through what P could be according to the syntax and the semantics.

Case 1: P is a rule set call. Then either $\mathrm{top}(S) \Rightarrow_P G$ or $\mathrm{top}(S) \not\Rightarrow_P$. So either $[\mathrm{call}_1]$ or $[\mathrm{call}_2]$ can be applied.

Case 2: P is a loop. If P is a loop, $[\mathrm{alap}_1]$ can be applied.

Case 3: P is fail*,* `skip` *or an* `or` *statement.* Then [fail], [skip], or $[\mathrm{or}_1]$ can be applied respectively.

Case 4: P is of the form `if` P_1 `then` P_2 `else` P_3 *or* `try` P_1 `then` P_2 `else` P_3. Then $[\mathrm{if}_1]$ or $[\mathrm{try}_1]$ can be applied. If any then-clause or else-clause is omitted as specified by the syntax, $[\mathrm{if}_5]$, $[\mathrm{try}_5]$, $[\mathrm{try}_6]$, or $[\mathrm{try}_7]$ can be applied.

Case 5: P is of the form ITE(P_1, P_2, P_3) *or TRY*(P_1, P_2, P_3). If P contains a `break`, the fourth point of the lemma is satisfied, as containing ITE or TRY statements makes P not a command sequence. So for the remainder of this case, assume P does not contain a `break`. If P_1 is a sequential composition, let $P_1 = P_1'; P_1''$ where P_1' is not a sequential composition. Otherwise let $P_1 = P_1'$. We shall show the lemma's statement by induction on how many ITE or TRY statements are nested in P_1' via the first sequential component of the condition.

- For the base case, assume P_1' is not an ITE or TRY statement. Then P_1' is not a sequential composition and covered by cases 1 to 4 (P_1' cannot be `break` since P contains no `break`).
- Now for the induction step, assume that P_1' is an ITE or TRY statement. Then P_1' does derive either a configuration $\langle P_1''', S'\rangle$, a graph stack S' or fail by the induction hypothesis. Hence one of $[\mathrm{if}_2]$, $[\mathrm{if}_3]$, $[\mathrm{if}_4]$, $[\mathrm{try}_2]$, $[\mathrm{try}_3]$, or $[\mathrm{try}_4]$ can be applied to $\langle P, S\rangle$.

Case 6: P is a sequential composition. Then we can decompose P into $P = P_1;P_2$ where P_1 is not a sequential composition. We can apply $[\text{seq}_1]$, $[\text{seq}_2]$, or $[\text{seq}_3]$ since $\langle P_1, S\rangle \to \langle P_1', S'\rangle$, $\langle P_1, S\rangle \to S'$, or $\langle P_1, S\rangle \to$ fail respectively by cases 1 to 5.

Case 7: P contains a `break`.

If P is not a command sequence, the final point of the lemma is satisfied. Otherwise, P satisfies context conditions, meaning it must be enclosed within a loop, so either case 2 or one of the other previous cases is applicable.

□

Lemma 4.1 has a case where the extended command sequence contains a `break`. This is because for a transition sequence not to get stuck on a `break`, we need to start with a command sequence where the `break` is within a loop, which we cannot guarantee if we consider a single transition step like in Lemma 4.1. In order to deal with this case, we prove that we can construct a transition sequence that leads to a state with no `break` in the following lemma. However, we need to restrict it to extended configurations reachable from a valid GP 2 program. We say that an extended configuration C is *reachable* if there is a configuration $\langle P, [G]\rangle$ such that $\langle P, [G]\rangle \to^* C$. This will still allow us to work towards non-blocking, since we only care about transition sequences that start with valid GP 2 programs.

Lemma 4.2 (Removing the `break` Statement)**.** Let $\langle P, S\rangle$ be an extended configuration that is reachable and non-terminal. Suppose that P contains `break`. Then one of the following applies.

- There is an extended configuration $\langle P', S'\rangle$ containing no `break` statement such that $\langle P, S\rangle \to^* \langle P', S'\rangle$.
- There is a graph stack S' such that $\langle P, S\rangle \to^+ S'$.
- $\langle P, S\rangle \to^+$ fail

Proof. First assume that $\langle P, S\rangle$ satisfies context conditions, i.e. the `break` is contained within a loop, and if the `break` is in the condition of an `if` or `try` statement, the enclosing loop must be in the same condition.

We will apply various inference rules to construct a transition sequence starting in $\langle P, S\rangle$. Remember that whenever we apply such an inference rule, it results in either a non-terminal extended configuration, a graph stack, or fail. If it results in a graph stack or fail, the second or third point of the lemma is satisfied. So at each step of the transition sequence we construct, we only need to consider the case where an inference rule results in a non-terminal extended command sequence.

If there are multiple loops with `break` statements, they are either in different sequential composition components, or nested. So let us show this lemma by induction on nesting and sequential composition.

As a base case, assume P contains a single loop with a `break`, and want to show we can apply a sequence of inference rules that ultimately removes the `break`. So P is of the form $Q_0;Q_1!;Q_2$, where Q_1 contains a `break`, and neither Q_0 nor Q_2 do. (What follows also applies if P is of the form $Q_1!;Q_2$, $Q_0;Q_1!$, or $Q_1!$.) We can repeatedly apply Lemma 4.1 to transition to $Q_1!;Q_2$. Then we apply $[\text{alap}_1]$ followed by $[\text{try}_1]$ to get $\text{TRY}(Q_1, Q_1!, \texttt{skip})$. We can then use Lemma 4.1 repeatedly as a premise for $[\text{try}_2]$ until we get $\text{TRY}(Q_3;Q_4, Q_1!, \texttt{skip})$, where Q_3 contains `break` and is not a sequential composition. If Q_3 is a `break`, we can apply $[\text{try}_2]$ under the premise of [break], followed by $[\text{alap}_2]$ to get rid of the `break`. We know Q_3 cannot be a loop since we assumed $Q_1!$ is the enclosing loop of the `break`. So Q_3 is either an `or`, `if`, or `try` statement. If it is an `or` statement, we can apply $[\text{or}_1]$ or $[\text{or}_2]$ to either remove the `break` or lead to $\text{TRY}(\texttt{break};Q_5, Q_1!, \texttt{skip})$. Similarly, if Q_3 is an `if` or

`try` statement, the `break` must be in the `then` or `else` part due to context conditions, and we can use inference rules to either remove the `break` or lead to TRY(`break;`$Q_5, Q_1!,$ `skip`). We can now apply [try$_2$] under the premise of [break] to get TRY(`break`$, Q_1!,$ `skip`). To this, we can apply [alap$_2$], which gets rid of the `break`.

For the induction step, let us first consider the case of nesting. Assume that P is of the form $Q_0;(Q_1;Q_2;Q_3)!;\ Q_4$, where Q_2 satisfies the lemma statement, and either Q_1 or Q_2 contain a single `break`. We can use the same arguments as in the base case in addition to [seq$_1$] under the premise of the induction hypothesis to get rid of the `break`.

Now consider sequential composition. As an induction step, assume that P is of the form $Q_0;Q_1!;Q_2;Q_3!;Q_4$, where one of Q_1 or Q_3 satisfies the lemma statement, and the other contains a single `break`. Again, we can use the arguments from the base case as well as the induction hypothesis in conjunction with [seq$_1$] to remove the `break`.

Finally, assume that $\langle P,S\rangle$ does not satisfy context conditions, i.e. either there is a `break` without an enclosing loop, or there is a `break` in the condition of a branching statement whose enclosing loop is not within that condition. Since $\langle P,S\rangle$ is reachable, the latter cannot be the case: transitions steps cannot separate a `break` from its enclosing loop in a way that they stop being within the same condition of a branching statement (loops can only be removed by inference rules, they cannot be "moved"). So suppose there is a `break` without an enclosing loop. This must be because [alap$_1$] is applied earlier in the transition sequence, so it must be within the condition of a `try` or TRY. So we can use the same arguments as earlier in the proof, except that we need not argue that some of the inference rule, such as [alap$_1$] or [try$_1$] need to be applied. □

Now that we have Lemmata 4.1 and 4.2, we can prove that the non-blocking property holds.

Proposition 4.3 (Non-Blocking Property)**.** Let $\langle P,S\rangle$ be an extended configuration that is reachable and non-terminal. Then there is a transition step $\langle P,S\rangle \to C$ for some extended configuration C.

Proof. If P does not contain a `break`, this proposition follows from Lemma 4.1. Otherwise, it follows from Lemma 4.2. □

Let us now introduce a lemma that makes various statements about the size of host graph stacks in order to ensure that the inference rules are well-defined. Since we defined stacks to be nonempty, we want to make sure that if a transition sequence starts with a nonempty stack, it cannot lead to an empty stack, which the following lemma shows. Furthermore, when a transition sequence terminates in a graph stack, we want that stack to only contain one host graph.

For this lemma, we want to start from a valid GP 2 program, not extended command sequences in general (since they may contain auxiliary constructs like ITE and TRY). So we consider configurations in CommandSeq $\times\ \mathscr{S}$. These follow the context conditions on where the `break` statement can appear as specified in [3].

Lemma 4.4 (Stack Size)**.** Let $\langle P,[G]\rangle$ be a configuration in CommandSeq $\times\ \mathscr{S}$.

(a) If $\langle P,[G]\rangle \to^* \langle P',S\rangle$, where $\langle P',S\rangle$ is an extended configuration, then $|S|\geq 1$.

(b) If $\langle P,[G]\rangle \to^+ S$, where S is a graph stack, then $|S|=1$.

Proof. The statement in (a), is satisfied for zero transition steps. So let us examine the inference rules that contain push, pop, and pop2. The rule [call$_1$] contains both push and pop, but preserves the size of the stack. The rules that push a graph onto the stack are [if$_1$] and [try$_1$] which are exactly the rules that

introduce an ITE or a TRY. The rules that pop a graph from the stack are $[\text{alap}_2]$, $[\text{if}_3]$, $[\text{if}_4]$, $[\text{try}_3]$, and $[\text{try}_4]$. These are exactly rules that remove an ITE or TRY from the extended command sequence. Since $\langle P, [G]\rangle$ contains no ITE or TRY statements and only one host graph, we have $|S| = \#(P') + 1$, where # counts the combined number of ITE and TRY statements in an extended command sequence. Since $|S| = \#(P') + 1$, we have $|S| \geq 1$.

Now in case (b), we can break down the transition sequence into $\langle P, [G]\rangle \to^* \langle P', S'\rangle \to S$. Like in the proof of (a), the formula $|S'| = \#(P') + 1$ applies. Let us examine which inference rules can be applied in the final step of the transition. It can only be either [skip], $[\text{call}_1]$, or $[\text{alap}_2]$. To apply [skip], P' must be `skip` and $\#(\texttt{skip}) = 0$, so $|S| = |S'| = 1$. To apply $[\text{call}_1]$, P' must be rule set call, and hence cannot contain ITE or TRY, so $|S| = |S'| = 1$. To apply $[\text{alap}_2]$, P' must be of the form $\text{TRY}(\texttt{break}, P''!, \texttt{skip})$, where P'' is an extended command sequence. We know P'' cannot contain an ITE or TRY statement because they can only be nested in their first argument. Indeed, if an extended command sequence already starts with an ITE or TRY, no inference rule allows for said ITE or TRY statement to be nested within another one. So the only way to nest statements is via the rule $[\text{try}_2]$, which modifies the first argument. But the first argument of P' is `break`, which contains no ITE or TRY statements. So $\#(P') = 1$ and $|S'| = 2$. Since we apply $[\text{alap}_2]$, we have $S = \text{pop2}(S')$, so $|S| = |S'| - 1 = 1$. □

We also want to make sure that if we call pop2 on a stack to pop its second element, the stack does indeed contain at least two elements. More precisely, under the premise of Lemma 4.4, if $\langle P, [G]\rangle \to^+ \langle P', \text{pop2}(S)\rangle$ (an extended configuration) or $\langle P, [G]\rangle \to^+ \text{pop2}(S)$ (a graph stack), then $|S| \geq 2$. This follows directly from Lemma 4.4 since $|\text{pop2}(S)| = |S| - 1$.

Let us now use Lemmata 4.1, 4.2, and 4.4 is to describe what the possible outcomes of a transition sequence starting in a valid GP 2 program are.

Theorem 4.5 (Outcomes of Transition Sequences)**.** Let $\langle P, [G]\rangle$ be a configuration. Then one of the following applies:

- There is an infinite transition sequence $\langle P, [G]\rangle \to \langle P_1, S_1\rangle \to \langle P_2, S_2\rangle \to \dots$ where $\langle P_i, S_i\rangle$ is an extended configuration for all $i \geq 1$.
- $\langle P, [G]\rangle \to^+ [G']$ for some host graph G'.
- $\langle P, [G]\rangle \to^+$ fail.

Proof. Lemma 4.4 guarantees that if a transition sequence starts in $\langle P, [G]\rangle$ and ends in a stack, that stack only contains one graph. So for this proposition, it is enough to show that transition sequences end in a stack in the relevant cases.

In order to get rid of a potential `break` statement in P, we can apply Lemma 4.2 to $\langle P, [G]\rangle$. If we get a graph stack or fail, we fulfil the second or third case of this proposition. Otherwise we get an extended configuration $\langle P', S\rangle$ that contains no `break`.

Since there is now no `break` in either $\langle P, [G]\rangle$ or $\langle P', S\rangle$, we can apply the first, second, and third cases of Lemma 4.1 either indefinitely to get an infinite transition sequence, or until we get a graph stack or fail. □

Now that we know the possible outcomes of a transition sequence, we can define the *semantic function* $[\![\_]\!] : \text{CommandSeq} \to (\mathscr{G} \to 2^{\mathscr{G}^\oplus})$, where $\mathscr{G}$ is the set of host graphs, $[\mathscr{G}]$ the set of stacks consisting of exactly one host graph (which we can identify with single host graphs), and $\mathscr{G}^\oplus = [\mathscr{G}] \cup \{\text{fail}, \bot\}$. The symbol $\bot$ is used to represent an infinite transition sequence, i.e. divergence. The function is defined as

$$[\![P]\!]G = \{X \in [\mathscr{G}] \cup \{\text{fail}\} \mid \langle P, G\rangle \to^+ X\} \cup \{\bot \mid P \text{ can diverge from } G\}.$$

This functions differs from the one presented in [17] and Section 2 since $\bot$ is only used when P diverges, because we know by Proposition 4.3 that P cannot get stuck. We will show in Lemma 5.5 that every infinite or stuck transition sequence in the previous semantics corresponds to an infinite transition sequence in the new semantics.

Let us now examine the property of *finite nondeterminism* as specified by Apt in Section 4.1 of [1], i.e. the set of elements reachable from a configuration in one transition step is finite. A related concept is *bounded nondeterminism*, where the cardinality of the aforementioned set is finite and depends on the program only (and not on the size of the current state). An example for a language with bounded nondeterminsim is Dijkstra's language of guarded commands [18]. Many references [8, 10, 18, 19] equate the concepts of finite and bounded nondeterminism and call it "bounded nondeterminism". However, rule-based languages generally have unbounded nondeterminism because they rely on nondeterministic rule matching. This also applies to GP 2 which the following example illustrates.

Example 4.6. Consider the rule $\mathtt{r} : {}_1\bullet\leftarrow\bullet\Rightarrow{}_1\bullet$ and the *comb graph* G_4 as shown in Figure 6. There

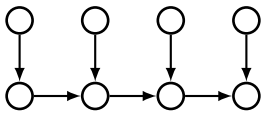


Figure 6: The comb graph G_4

are four possible matches for the left-hand side of rule r in graph G_4, so applying the rule can result in four different non-isomorphic graphs, which is a finite amount. When applying r to comb graph G_k, we get k non-isomorphic graphs, which depends on the size of the host graph and hence is not bounded. □

We now show that GP 2 does have finite nondeterminism.

Proposition 4.7 (Finite Nondeterminism)**.** Let $\gamma \in \text{ExtComSeq} \times \mathscr{S}$ be an extended configuration, and $T_\gamma = \{\gamma' \mid \gamma \to \gamma' \in \text{ExtCommSeq} \times \mathscr{S}\}$. Then $|T_\gamma|$ is finite.

Proof. The only inference rules that cause nondeterminism are $[\text{or}_1]$, $[\text{or}_2]$, and $[\text{call}_1]$. If the rules $[\text{or}_1]$ and $[\text{or}_2]$ are applicable to γ then there are exactly two configurations reachable from γ. In $[\text{call}_1]$, the nondeterminism comes from several GP 2 rules being called non-deterministically as part of a rule set, as well as from all the ways these rules can be matched in the host graph. Since rule sets and host graphs are finite, the number of configurations reachable from γ in one step via the inference rule $[\text{call}_1]$ is finite as well. □

Reynolds [18] defines this kind of nondeterminism using the semantic function instead of the set of configurations reachable in one step. The following corollary shows that this semantics fulfils that definition as well.

Corollary 4.8. Let $P \in \text{CommandSeq}$ and $G \in \mathscr{G}$ such that $[\![P]\!]G$ is infinite. Then $\bot \in [\![P]\!]G$.

Proof. Let $\gamma_0 = \langle P, [G]\rangle$. Then $T^*_{\gamma_0} = \{\gamma \mid \gamma_0 \to^* \gamma \in \text{ExtCommSeq} \times \mathscr{S}\}$ is infinite as well since it contains all elements of $[\![P]\!]G$ except perhaps fail or $\bot$. The set $T^*_{\gamma_0}$ can be seen as a tree whose nodes are configurations and whose edges are defined by transition relations. Since T_γ is finite for all configurations γ by Proposition 4.7, each node in the tree only has finitely many adjacent nodes. By König's Lemma [12], the tree contains an infinite path. Since every node of the tree is reachable from the root γ_0, there is an infinite path starting from γ_0. By definition of the tree, this means there is an infinite transition sequence starting with γ_0. By definition of the semantic function, we can conclude that $\bot \in [\![P]\!]G$. □

5 Comparison to the Previous Semantics

In this section, we show that the new semantics is a conservative extension of the previous one, i.e. their behaviour is equivalent on converging configurations, and if a configuration diverges in the previous semantics, it also diverges in the new one.

When we mention graph stacks in this section, we allow them to be empty. We use the notation $[G_1, G_2, \ldots, G_k, S]$ (where G_i are graphs, S is a graph stack, and $k > 0$) to denote a stack whose top k elements are $G_1, G_2, \ldots, G_k$, and whose remaining elements are the elements of S.

Lemma 5.1 (Simulating Finite Old Transition Sequences)**.** Let $P \in \text{CommandSeq}$, $G \in \mathscr{G}$, and $X \in \{\langle P', G'\rangle, G', \text{fail}\}$, where $P' \in \text{CommandSeq}$ and $G' \in \mathscr{G}$. If $\langle P, G\rangle \rightsquigarrow^* X$, then, for any graph stack S, there is a transition sequence

- $\langle P, [G,S]\rangle \to^* \langle P', [G',S]\rangle$ if $X = \langle P', G'\rangle$.
- $\langle P, [G,S]\rangle \to^* [G']$ if $X = G'$.
- $\langle P, [G,S]\rangle \to^* \text{fail}$ if $X = \text{fail}$.

Proof. We shall prove this lemma by induction on the number of `if`, `try`, and `!` statements in P combined.

If P has no `if`, `try`, or `!` statements, none of the [if], [try], and [alap] inference rules are applicable. All other rules behave identically in both semantics when identifying the tops graph of the stacks in the new rules with the graphs in the previous rules. Hence the base case is satisfied.

As the induction hypothesis, assume the lemma holds for P containing k `if`, `try`, or `!` statements. Now consider the case where P contains $k+1$ of them. Let $\langle P_1, G_1\rangle \rightsquigarrow \langle P_2, G_2\rangle$ be a derivation step of $\langle P, G\rangle \rightsquigarrow^* X$ that uses an [if], [try], or [alap] rule (possibly as a premise for another rule such as $[\text{seq}_1]$). If such a step does not exist, the lemma holds by the same argument used in the base case. Consider the [if], [try], or [alap] rule [r] that relates to the `if`, `try`, or `!` statement enclosing all others resolved in the same step. Then no rule where [r] is a premise (or the premise of a premise) is an [if], [try], or [alap] rule. Since those are identical in both semantics, we only need to show that the part of P resolved by [r] is resolved in a way that fulfils the lemma statement.

If $[\text{r}] = [\text{if}_1]$, we have $\langle \texttt{if } C \texttt{ then } P_3 \texttt{ else } Q, G_3\rangle \rightsquigarrow \langle P_3, G_3\rangle$ under the premise of $\langle C, G_3\rangle \rightsquigarrow^+ H$. Since P contains $k+1$ `if`, `try`, or `!` statements, C contains at most k of them. So by the induction hypothesis, there is a transition sequence $\langle C, [G_3,S]\rangle \to^* [H,S]$ (for any graph stack S). We can decompose this transition sequence into $\langle C, [G_3,S]\rangle \to^l \langle C_4, S_4\rangle \to [H,S]$ where $l \geq 0$, and S_4 is a graph stack. This fulfills the premise of $[\text{if}_2]$ l times, and then the premise of $[\text{if}_3]$ once. So for any graph stack S', the premises are fulfilled by choosing $S = [G_3, S']$, and we have

$$\begin{aligned}\langle \texttt{if } C \texttt{ then } P_3 \texttt{ else } Q, [G_3,S']\rangle \quad &\to_{[\text{if}_1]} \langle \text{ITE}(C,P_3,Q), [G_3,G_3,S']\rangle \to^l_{[\text{if}_2]} \langle \text{ITE}(C_4,P_3,Q), S_4\rangle \\ &\to_{[\text{if}_3]} \langle P_2, [G_3,S']\rangle.\end{aligned}$$

If $[\text{r}] = [\text{if}_2]$, we have $\langle \texttt{if } C \texttt{ then } P_3 \texttt{ else } Q, G_3\rangle \rightsquigarrow \langle Q, G_3\rangle$ under the premise of $\langle C, G_3\rangle \rightsquigarrow^+ \text{fail}$. Again, we can use the induction hypothesis to conclude $\langle C, [G_3,S]\rangle \to^* \text{fail}$ (for any graph stack S), which is a sequence of $l \geq 0$ transitions $\langle C, [G_3,S]\rangle \to^l \langle C_4, S_4\rangle$ between extended configurations, followed by 1 transition $\langle C_4, S_4\rangle \to \text{fail}$. This fulfills the premise of $[\text{if}_2]$ l times, and then the premise of $[\text{if}_4]$ once. So for any graph stack S', the premises are fulfilled by choosing $S = [G_3, S']$, and we have

$$\begin{aligned}\langle \texttt{if } C \texttt{ then } P_3 \texttt{ else } Q, [G_3,S']\rangle \ &\to_{[\text{if}_1]} \langle \text{ITE}(C,P_3,Q), [G_3,G_3,S']\rangle \to^l_{[\text{if}_2]} \langle \text{ITE}(C_4,P_3,Q), S_4\rangle \\ &\to_{[\text{if}_4]} \langle Q, [G_3,S']\rangle.\end{aligned}$$

If $[\mathrm{r}] = [\mathrm{try}_1]$, we can use the same arguments as when $[\mathrm{r}] = [\mathrm{if}_1]$ to get a transition sequence $\langle \texttt{try}\ C\ \texttt{then}\ P_3\ \texttt{else}\ Q, [G_3, S']\rangle \to^+ \langle P_3, [G_4, S']\rangle$ for any graph stack S'.

If $[\mathrm{r}] = [\mathrm{try}_2]$, we can use the same arguments as when $[\mathrm{r}] = [\mathrm{if}_2]$ to get a transition sequence $\langle \texttt{try}\ C\ \texttt{then}\ P_3\ \texttt{else}\ Q, [G_3, S']\rangle \to^+ \langle Q, [G_3, S']\rangle$ for any graph stack S'.

If $[\mathrm{r}] = [\mathrm{alap}_1]$, we have $\langle P_3!, G_3\rangle \rightsquigarrow \langle P_3!, H\rangle$ under the premise of $\langle P_3, G_3\rangle \rightsquigarrow^+ H$. Since P contains $k+1$ `if`, `try`, or `!` statements, P_3 contains at most k of them. So by the induction hypothesis, we can conclude $\langle P_3, [G_3, S]\rangle \to^* [H, S]$ for any graph stack S. We can decompose this transition sequence into $\langle P_3, [G_3, S]\rangle \to^l \langle P_4, S_4\rangle \to [H, S]$, where $l \geq 0$, and S_4 is a graph stack. These derivations fulfil the premise of $[\mathrm{try}_2]$ l times and then the premise of $[\mathrm{try}_3]$ once. So for any graph stack S', the premises are fulfilled by choosing $S = [G_3, S']$, and we have

$$\begin{aligned}\langle P_3!, [G_3, S']\rangle &\to_{[\mathrm{alap}_1]} \langle \texttt{try}\ P_3\ \texttt{then}\ P!\ \texttt{else}\ \texttt{skip}, [G_3, S']\rangle \to_{[\mathrm{try}_1]} \langle \mathrm{TRY}(P_3, P_3!, \texttt{skip}), [G_3, G_3, S']\rangle \\ &\to^l_{[\mathrm{try}_2]} \langle \mathrm{TRY}(P_4, P_3!, \texttt{skip}), S_4\rangle \to_{[\mathrm{try}_3]} \langle P_3!, [H, S']\rangle.\end{aligned}$$

If $[\mathrm{r}] = [\mathrm{alap}_2]$, we have $\langle P_3!, G_3\rangle \rightsquigarrow G_3$ under the premise of $\langle P_3, G_3\rangle \rightsquigarrow^+ \mathrm{fail}$. Again, we can use the induction hypothesis to conclude $\langle P_3, [G_3, S]\rangle \to^* \mathrm{fail}$ for any graph stack S. We can decompose this transition sequence into $\langle P_3, [G_3, S]\rangle \to^l \langle P_4, [G_4, S]\rangle \to \mathrm{fail}$, where $l \geq 0$, and G_4 is a graph. Let us argue that the graph stack right before the fail is $[G_4, S]$. The stack ends in S because P_3 is a command sequence and hence does not contain ITE or TRY statements, which are the only constructs that could pop and hence modify S. Any `if`, `try`, and `!` statements in P_3 must resolve before the configuration that leads to fail in one step because fail is not reachable in one step from `if`, `try`, `!`, ITE, and TRY statements. Since they all resolved, any push has a corresponding pop, meaning the number of graphs in the stack before S remains 1. The derivations fulfil the premise of $[\mathrm{try}_2]$ l times and then the premise of $[\mathrm{try}_4]$ once. So for any graph stack S', the premises are fulfilled by choosing $S = [G_3, S']$, and we have

$$\begin{aligned}\langle P_3!, [G_3, S']\rangle &\to_{[\mathrm{alap}_1]} \langle \texttt{try}\ P_3\ \texttt{then}\ P!\ \texttt{else}\ \texttt{skip}, [G_3, S']\rangle \to_{[\mathrm{try}_1]} \langle \mathrm{TRY}(P_3, P_3!, \texttt{skip}), [G_3, G_3, S']\rangle \\ &\to^l_{[\mathrm{try}_2]} \langle \mathrm{TRY}(P_4, P_3!, \texttt{skip}), [G_4, G_3, S']\rangle \to_{[\mathrm{try}_4]} \langle \texttt{skip}, [G_3, S']\rangle \to_{[\mathrm{skip}]} [G_3, S'].\end{aligned}$$

If $[\mathrm{r}] = [\mathrm{alap}_3]$, we have $\langle P_3!, G_3\rangle \rightsquigarrow H$ under the premise of $\langle P_3, G_3\rangle \rightsquigarrow^* \langle \texttt{break}, H\rangle$. Again, we can use the induction hypothesis to conclude $\langle P_3, [G_3, S]\rangle \to^* \langle \texttt{break}, [H, S]\rangle$ (for any graph stack S). Let $l \geq 0$ be the number of steps in that transition sequence. These derivations fulfil the premise of $[\mathrm{try}_2]$ l times. So for any graph stack S', the premises are fulfilled by choosing $S = [G_3, S']$, and we have

$$\begin{aligned}\langle P_3!, [G_3, S']\rangle &\to_{[\mathrm{alap}_1]} \langle \texttt{try}\ P_3\ \texttt{then}\ P_3!\ \texttt{else}\ \texttt{skip}, [G_3, S']\rangle \to_{[\mathrm{try}_1]} \langle \mathrm{TRY}(P_3, P_3!, \texttt{skip}), [G_3, G_3, S']\rangle \\ &\to^l_{[\mathrm{try}_2]} \langle \mathrm{TRY}(\texttt{break}, P_3!, \texttt{skip}), [H, G_3, S']\rangle \to_{[\mathrm{alap}_2]} [H, S'].\end{aligned}$$

□

Lemma 5.2 (Configurations That Get Stuck)**.** Let $\langle P, G\rangle$ be a configuration to which no inference rule of the previous semantics is applicable. Then P starts with an `if`, `try`, or `!` statement such that the condition (or the body in the case of `!`) diverges or gets stuck in the previous semantics.

Proof. For this proof, whenever we say a rule is applicable, we mean it is either applicable, or it can be used as a premise for a [seq] rule.

Let us first show that, if P does not start with an `if`, `try`, or `!` statement, we can apply an inference rule to $\langle P, G\rangle$. If P starts with a rule set, that rule set is either applicable to the host graph or not, so either $[\mathrm{call}_1]$ or $[\mathrm{call}_2]$ can be applied. If P starts with a `break`, `skip`, or `fail`, then [break], [skip], or [fail] can be applied respectively. If P starts with an `or` statement, either $[\mathrm{or}_1]$ or $[\mathrm{or}_2]$ can be applied.

Now assume that P starts with an `if`, `try`, or `!` statement with a condition or body C. If $\langle C, G\rangle$ neither converges nor gets stuck, there is a transition sequence $\langle C, G\rangle \rightsquigarrow \mathrm{fail}$, or $\langle C, G\rangle \rightsquigarrow H$ for some graph H. Hence one of $[\mathrm{if}_1]$, $[\mathrm{if}_2]$, $[\mathrm{try}_1]$, $[\mathrm{try}_2]$, $[\mathrm{alap}_1]$, or $[\mathrm{alap}_2]$ is applicable. □

Lemma 5.3 (Loop-Free Command Sequences Do Not Get Stuck)**.** For every loop-free command sequence P and graph G, no transition sequence starting with $\langle P, G\rangle$ gets stuck in the previous semantics.

Proof. We prove this lemma by structural induction. For a base case, consider programs consisting of a single rule set R. Then on any graph G, R is either applicable or not. So to $\langle R, G\rangle$ we can apply either $[\text{call}_1]$ or $[\text{call}_2]$, leading to a graph or fail. They are both transition sequences that end in a terminal state, and hence do not get stuck.

Another base case is `skip` or `fail`, which always lead to a graph or fail in a single step, and hence cannot lead to stuck transition sequences.

The `break` statement cannot be in P since context conditions require it to have an enclosing loop.

For the induction step, assume that every proper subprogram of P cannot get stuck, and show that P cannot get stuck either.

Assume $P = P_1; P_2$. By the induction hypothesis, for any graph G, no transition sequence starting with $\langle P_1, G\rangle$ gets stuck, i.e. they can all be extended to either $\langle P_1, G\rangle \rightsquigarrow^+$ fail or $\langle P_1, G\rangle \rightsquigarrow^+ H$ for some graph H. This fulfils the premise of $[\text{seq}_1]$ some number of times, and then the premise of either $[\text{seq}_2]$ or $[\text{seq}_3]$ once. So each transition sequence staring with $\langle P, G\rangle$ must be of the form $\langle P, G\rangle \rightsquigarrow^+_{[\text{seq}]}$ fail or $\langle P, G\rangle \rightsquigarrow^+_{[\text{seq}]} H$.

Assume $P =$ `if` C `then` P_1 `else` P_2. Then by induction hypothesis, for any graph G, no transition sequence starting with $\langle C, G\rangle$ can get stuck, i.e. they can all be extended to either $\langle C, G\rangle \rightsquigarrow^+$ fail or $\langle C, G\rangle \rightsquigarrow^+ H$ for some graph H. This satisfies the premise of either $[\text{if}_1]$ or $[\text{if}_2]$. So each transition sequence staring with $\langle P, G\rangle$ must be of the form $\langle P, G\rangle \rightsquigarrow_{[\text{if}_1]} \langle P_1, G\rangle$ or $\langle P, G\rangle \rightsquigarrow_{[\text{if}_2]} \langle P_2, G\rangle$. Any continuation of these sequences cannot get stuck because P_1 and P_2 satisfy the induction hypothesis.

The case $P =$ `try` C `then` P_1 `else` P_2 is analogous to the previous one.

If P is an `if` or `try` with omitted `then` or `else` clauses, one of $[\text{if}_3]$, $[\text{try}_3]$, $[\text{try}_4]$, or $[\text{try}_5]$ can be applied, and then the arguments used in the `if` and `try` cases can be applied.

Assume $P = P_1$ `or` P_2. Then for any graph G, any transition sequence starting with $\langle P, G\rangle$ starts with either $\langle P, G\rangle \rightsquigarrow_{[\text{or}_1]} P_1$ or $\langle P, G\rangle \rightsquigarrow_{[\text{or}_2]} P_2$. Any continuation of these sequences cannot get stuck because P_1 and P_2 satisfy the induction hypothesis. □

Lemma 5.4 (Non-Nested Loops Do Not Get Stuck)**.** For every loop-free command sequence P and graph G, no transition sequence starting with $\langle P!, G\rangle$ gets stuck in the previous semantics.

Proof. It is enough to show that either $\langle P, G\rangle \rightsquigarrow^+ H$, $\langle P, G\rangle \rightsquigarrow^+$ fail, or $\langle P, G\rangle \rightsquigarrow^* \langle$`break`$, H\rangle$. Because then, one of $[\text{alap}_1]$, $[\text{alap}_2]$, or $[\text{alap}_3]$ is applicable. And if $[\text{alap}_1]$ was applicable, we get $\langle P!, G\rangle \rightsquigarrow \langle P!, H\rangle$, the same arguments can be used on $\langle P!, H\rangle$, and hence on all its successors.

First of all, since P contains no `!`, $\langle P, G\rangle$ cannot diverge. If P does not contain a `break`, $\langle P, G\rangle$ cannot get stuck by Lemma 5.3. If P does contain a `break`, that is never called, the arguments of the proof of Lemma 5.3 still apply, and $\langle P, G\rangle$ does not get stuck. If P contains a `break` that is called, that means there is a transition sequence $\langle P, G\rangle \rightsquigarrow^* \langle$`break`$, H\rangle$, or $\langle P, G\rangle \rightsquigarrow^* \langle$`break`$; Q, H\rangle$, to which we can apply [break] to get the former.

So either $\langle P, G\rangle \rightsquigarrow^* \langle$`break`$, H\rangle$, or $\langle P, G\rangle$ neither diverges nor gets stuck, which means $\langle P, G\rangle$ must resolve to either a graph or fail. □

Lemma 5.5 (Simulating Old Transition Sequences That Are Infinite or Stuck)**.** Assume there is an infinite transition sequence $\langle P, G\rangle \rightsquigarrow \ldots$, or a stuck transition sequence $\langle P, G\rangle \rightsquigarrow^* \langle P', G'\rangle$. Then for any graph stack S, there is an infinite transition sequence $\langle P, [G, S]\rangle \to \ldots$.

Proof. First assume there is an infinite transition sequence $\langle P,G\rangle \rightsquigarrow \langle P_1,G_1\rangle \rightsquigarrow \dots$. To each step $\langle P_i,G_i\rangle \rightsquigarrow \langle P_{i+1},G_{i+1}\rangle$ in that transition sequence, we can apply Lemma 5.1 to get a transition sequence $\langle P_i,[G_i,S_i]\rangle \to^* \langle P_{i+1},[G_{i+1},S_{i+1}]\rangle$ for any graph stack S_i and some S_{i+1}. We can concatenate these into an infinite transition sequence.

Now assume there is a stuck transition sequence $\langle P,G\rangle \rightsquigarrow^* \langle P',G'\rangle$. We can apply Lemma 5.1 to each step in that sequence to get $\langle P,[G,S]\rangle \to^* \langle P',[G',S]\rangle$ for each graph stack S.

We claim that for any command sequence P' such that $\langle P',G'\rangle$ is stuck with respect to $\rightsquigarrow$, there is a diverging transition sequence $\langle P',[G',S]\rangle \to \dots$ for any graph stack S. This is enough to prove the Lemma. Let us show this by induction on the combined number of `if`, `try`, and `!` statements in P'.

If there are no such statements, then $\langle P',G'\rangle$ cannot get stuck by Lemma 5.2, so there has to be at least one, and P' starts with it. If that one statement is an `if` or a `try`, then $\langle P',G'\rangle$ cannot get stuck by Lemma 5.3. If it is a `!` statement, $\langle P',G'\rangle$ cannot get stuck by Lemma 5.4. So there have to be at least two `if`, `try`, or `!` statements.

So for our base case, assume P' contains exactly two `if`, `try`, or `!` statements. By Lemma 5.3, P' must contain a `!` statement. If the two statements are not nested, we can apply Lemmata 5.3 and/or 5.4 sequentially to conclude that $\langle P',G'\rangle$ does not get stuck. So the two statements must be nested, and they must be the start of P' by Lemma 5.2. Assume the "inner" statement is not a `!` statement. Then the "outer" statement must be the `!` statement. By Lemma 5.4, $\langle P',G'\rangle$ does not get stuck. So the "inner" statement must be a `!` statement $Q!$. The loop $Q!$ cannot resolve to a graph or fail because then, the "outer" statement could be resolved and $\langle P',G'\rangle$ would not be stuck. By Lemma 5.4, $Q!$ cannot get stuck either. So $Q!$ must diverge, and so must the condition or body C of the starting statement of P'. So there is an infinite transition sequence $\langle C,G'\rangle \rightsquigarrow \dots$, and hence by Lemma 5.1 to each step in that transition sequence, we get $\langle C,[G',S]\rangle \to \dots$ for any graph stack S. This serves as a premise for $[\text{if}_2]$ or $[\text{try}_2]$, which we can use to get an infinite transition sequence $\langle P',G'\rangle \to \dots$.

Now for the induction step, assume we get infinite transition sequences for programs with at most k `if`, `try`, and `!` statements. Assume P' has $k+1$ such statements. By Lemma 5.2, since $\langle P',G'\rangle$ is stuck, P' must start with one of those statements. So we can apply either $[\text{if}_1]$, $[\text{try}_1]$, or $[\text{alap}_1]$ followed by $[\text{try}_1]$ to get $\langle P',[G',S']\rangle \to^+ \langle P'',[G',G',S']\rangle$ for any graph stack S', where P'' starts with either $\text{ITE}(C,Q,Q')$ or $\text{TRY}(C,Q,Q')$. Now C has at least one less `if`, `try`, or `!` statement than P', so we can apply the induction hypothesis to get an infinite transition sequence $\langle C,[G',G',S']\rangle \to \dots$, which can serve as premises for infinitely many applications of $[\text{if}_2]$ or $[\text{try}_2]$ (or these inference rules provide the premises for $[\text{seq}_1]$). Hence we have an infinite transition sequence $\langle P',[G',S']\rangle \to^+ \langle P'',[G',G',S']\rangle \to \dots$. □

Lemma 5.6 (Simulating Finite New Transition Sequences)**.** Let $P \in \text{CommandSeq}$, $G \in \mathscr{G}$, S a graph stack, and $X \in \{\langle P', [G',S']\rangle, [G',S'], \text{fail}\}$, where $P' \in \text{CommandSeq}$, S' is a graph stack, and $G' \in \mathscr{G}$. If $\langle P, [G,S]\rangle \to^* X$, then there is a transition sequence

- $\langle P, G\rangle \rightsquigarrow^* \langle P', G'\rangle$ if $X = \langle P', [G',S']\rangle$.
- $\langle P, G\rangle \rightsquigarrow^* G'$ if $X = [G',S']$.
- $\langle P, G\rangle \rightsquigarrow^* \text{fail}$ if $X = \text{fail}$.

Proof. Let us show this lemma by induction on the combined number of `if`, `try`, and `!` statements in P. If P has no such statements, no step in $\langle P, [G,S]\rangle \to^* X$ uses [if], [try], or [alap] rules. So no pushing or popping occurs and only the top of the stack is modified. The applied rules behave identically to those in the previous semantics when identifying the top of the stacks in the new rules with the host graphs in the previous rules. Hence the lemma is satisfied in the base case.

Now assume that the lemma holds for command sequences with k `if`, `try`, and `!` statements, and assume P has $k+1$ of them. By the previous paragraph, we can simulate transition steps that do not involve [if], [try], or [alap] rules. So let $\langle Q, [H, S']\rangle$ be a configuration (we will handle the case where $\langle Q, [H, S']\rangle$ is an extended configuration, but not a configuration later) in the sequence $\langle P, [G, S]\rangle \to^* X$ that uses such a rule to get the next state. Let [r] be that rule. It can only be [if$_1$], [try$_1$], or [alap$_1$] since $\langle Q, [H, S']\rangle$ is a configuration and hence does not contain ITE or TRY.

If [r] = [if$_1$], then we get $\langle Q, [H, S']\rangle \to \langle \mathrm{ITE}(C, Q_1, Q_2); Q_3, [H, H, S']\rangle \to^+ \langle Q_4; Q_3, [H, S']\rangle$, where Q_1, Q_2, and Q_3 are command sequences, and where $Q_4 \in \{Q_1, Q_2\}$ (since this is part of a transition sequence that ends in X, a configuration, graph, or fail, we know that the `if` eventually resolves). So either $\langle C, [H, H, S']\rangle \to^* [H', H, S']$ or $\langle C, [H, H, S']\rangle \to^*$ fail. We can apply the induction hypothesis to C, which has at most k `if`, `try`, and `!` statements, to get that either $\langle C, H\rangle \rightsquigarrow^* H'$ or $\langle C, H\rangle \rightsquigarrow^*$ fail. We can use this as a premise for either [if$_1$] or [if$_2$] to get $\langle Q, H\rangle \rightsquigarrow \langle Q_4; Q_3, H\rangle$, where $Q_4 \in \{Q_1, Q_2\}$, which simulates the sequence outlined at the beginning of this case.

If [r] = [try$_1$], then we get either $\langle Q, [H, S']\rangle \to \langle \mathrm{TRY}(C, Q_1, Q_2); Q_3, [H, H, S']\rangle \to^+ \langle Q_1; Q_3, [H', S']\rangle$ or $\langle Q, [H, S']\rangle \to \langle \mathrm{TRY}(C, Q_1, Q_2); Q_3, [H, H, S']\rangle \to^+ \langle Q_2; Q_3, [H, S']\rangle$, where Q_1, Q_2, and Q_3 are command sequences, and H' some graph. We can use the same arguments as in the previous case to get a transition sequence $\langle Q, H\rangle \rightsquigarrow \langle Q_1; Q_3, H'\rangle$ or $\langle Q, H\rangle \rightsquigarrow \langle Q_2; Q_3, H\rangle$.

If [r] = [alap$_1$], we have $Q = Q_1!; Q_2$ so either $\langle Q_1!; Q_2, [H, S']\rangle \to^* \langle Q_1!; Q_2, [H', S']\rangle$, $\langle Q_1!; Q_2, [H, S']\rangle \to^* \langle Q_2, [H, S']\rangle$, or $\langle Q_1!; Q_2, [H, S']\rangle \to^* \langle \mathrm{TRY}(\texttt{break}, Q_1!, \texttt{skip}); Q_2, [H', H, S']\rangle \to \langle Q_2, [H', S']\rangle$. We can conclude that either $\langle Q_1, [H, S']\rangle \to^* [H', S']$, $\langle Q_1, [H, S']\rangle \to^*$ fail, or $\langle Q_1, [H, S']\rangle \to^* \langle \texttt{break}, [H', S']\rangle$. By induction hypothesis, we get that either $\langle Q_1, H\rangle \rightsquigarrow^* H'$, $\langle Q_1, H\rangle \rightsquigarrow^*$ fail, or $\langle Q_1, H\rangle \rightsquigarrow^* \langle \texttt{break}, H'\rangle$. These can be used as premises of [alap$_1$], [alap$_2$], or [alap$_3$] to conclude that either $\langle Q_1!; Q_2, H\rangle \rightsquigarrow \langle Q_1!; Q_2, H'\rangle$, $\langle Q_1!; Q_2, H\rangle \rightsquigarrow \langle Q_2, H\rangle$, or $\langle Q_1!; Q_2, H\rangle \rightsquigarrow \langle Q_2, H'\rangle$.

Finally, if $\langle Q, [H, S']\rangle$ is an extended configuration, but not a configuration, Q must contain an ITE, TRY, or a `break` that does not satisfy the context conditions. All of these must originate in an `if`, `try`, or `!` statement, and are hence covered by the previous part of the proof. □

Note that the first point of Lemma 5.6 only applies to transition sequences between command sequences (they do not contain TRY or ITE constructs). So diverging transition sequences in the new semantics where all command sequences contain ITE or TRY after some step cannot be simulated with diverging transition sequences in the previous semantics.

Theorem 5.7. Let $P \in \mathrm{CommandSeq}$ and $G \in \mathscr{G}$. Then

(a) $[P]G \subseteq [\![P]\!]G$ and

(b) $[P]G \setminus \{\bot\} = [\![P]\!]G \setminus \{\bot\}$.

Proof. Lemma 5.1 guarantees that (a) holds for graphs and fail, and Lemma 5.5 guarantees that (a) holds for $\bot$. Furthermore, (b) follows from (a) and Lemma 5.6. □

6 Conclusion

We have introduced a new operational semantics for the graph programming language GP 2. Unlike the previous semantics, this one is entirely small-step and non-blocking. As a consequence, it accurately models computations which intuitively should diverge (and do so in the implementation). In particular, the new semantic function correctly lists $\bot$ as an outcome when there is a computation in which the

condition of a branching statement or the body of a loop diverges. We also obtain finite nondeterminism, meaning that for every configuration there are only finitely many choices for the next transition step.

Furthermore, we have shown that the new semantic function is an extension of the previous one, and that they are equivalent excluding divergence.

In future work, the new semantics should serve as a solid underpinning for setting up a time and space complexity theory for GP 2. Its small-step nature is crucial to defining atomic computation steps. Such a theory could possibly be automated akin to the resource analysis in [13].

Another aspect of the GP 2 semantics is that it is orthogonal to the definition of the transformation rules. The inference rules that depend on the domain of graph transformation only need the definition of a rule application ($[\text{call}_1]$) and the information when such an application fails ($[\text{call}_2]$). Hence this semantics could be used as a foundation for GP 2-like programming languages over other rule-based domains, such as string rewriting [5] or term rewriting [2].

Acknowledgements. We are grateful to the anonymous referees whose comments helped to improve the presentation of this paper.

References

[1] Krzysztof R. Apt (1984): *Ten Years of Hoare's Logic: A Survey. Part II: Nondeterminism*. *Theoretical Computer Science* 28, pp. 83–109, doi:10.1016/0304-3975(83)90066-X.

[2] Franz Baader & Tobias Nipkow (1998): *Term Rewriting and All That*. Cambridge University Press, doi:10.1017/cbo9781139172752.

[3] Christopher Bak (2015): *GP 2: Efficient Implementation of a Graph Programming Language*. Ph.D. thesis, Department of Computer Science, University of York, UK. Available at `https://etheses.whiterose.ac.uk/12586/`.

[4] Christopher Bak & Detlef Plump (2016): *Compiling Graph Programs to C*. In: *Proc. International Conference on Graph Transformation (ICGT 2016)*, *Lecture Notes in Computer Science* 9761, Springer, pp. 102–117, doi:10.1007/978-3-319-40530-8_7.

[5] Ronald V. Book & Friedrich Otto (1993): *String-Rewriting Systems*. Springer, doi:10.1007/978-1-4613-9771-7.

[6] Graham Campbell, Brian Courtehoute & Detlef Plump (2020): *Fast Rule-Based Graph Programs*. *ArXiv e-prints* arXiv:2012.11394. Available at `https://arxiv.org/abs/2012.11394`. 47 pages.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein (2009): *Introduction to Algorithms*, 3 edition. The MIT Press.

[8] Edsger W. Dijkstra (1997): *A Discipline of Programming*, 1st edition. Prentice Hall PTR.

[9] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange & Gabriele Taentzer (2006): *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science, Springer, doi:10.1007/3-540-31188-2_3.

[10] Wan Fokkink & Thuy Vu (2003): *Structural Operational Semantics and Bounded Nondeterminism*. *Acta Informatica* 39, pp. 501–516, doi:10.1007/s00236-003-0111-1.

[11] Ivaylo Hristakiev & Detlef Plump (2016): *Attributed Graph Transformation via Rule Schemata: Church-Rosser Theorem*. In: *Software Technologies: Applications and Foundations – STAF 2016 Collocated Workshops, Revised Selected Papers*, *Lecture Notes in Computer Science* 9946, Springer, pp. 145–160, doi:10.1007/978-3-319-50230-4_11.

[12] Dénes König (1927): *Über eine Schlussweise aus dem Endlichen ins Unendliche*. *Acta Sci. Math.(Szeged)* 3(2-3), pp. 121–130.

[13] Georg Moser & Manuel Schneckenreither (2020): *Automated amortised resource analysis for term rewrite systems*. *Science of Computer Programming* 185, p. 102306, doi:10.1016/j.scico.2019.102306.

[14] Hanne Riis Nielson & Flemming Nielson (2007): *Semantics with Applications: An Appetizer*. Springer, doi:10.1007/978-1-84628-692-6.

[15] 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.

[16] Detlef Plump (2012): *The Design of GP 2*. In: *Proc. 10th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2011)*, *Electronic Proceedings in Theoretical Computer Science* 82, pp. 1–16, doi:10.4204/EPTCS.82.1.

[17] Detlef Plump (2017): *From Imperative to Rule-based Graph Programs*. *Journal of Logical and Algebraic Methods in Programming* 88, pp. 154–173, doi:10.1016/j.jlamp.2016.12.001.

[18] John C Reynolds (1998): *Theories of Programming Languages*. Cambridge University Press, doi:10.1017/cbo9780511626364.

[19] H. Søndergaard & P. Sestoft (1992): *Non-determinism in Functional Languages*. *The Computer Journal* 35(5), pp. 514–523, doi:10.1093/comjnl/35.5.514.