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.

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 transition steps are atomic computation steps in the language, meaning that the implementation of the language faithfully corresponds to the semantics. 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 \( P = \text{try} (r1!) \) then skip else skip, with the rule \( r1 : 1 \Rightarrow 1 \; O \). applied to the host graph \( O \). 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 \) means that the rule \( r1 \) is called until it is no longer applicable.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 \( O \). In the previous semantics, try statements are handled with the following inference rules:

\[
\begin{align*}
\text{(try)}_1: & \quad (C, G) \xrightarrow{+} H \\
\text{(try \; C \; then \; P \; else \; Q, \; G)} & \quad \sim \quad (P, H)
\end{align*}
\]

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 \( (C, G) \) 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:

\[
\begin{align*}
\text{(try)}_1: & \quad \langle \text{try \; C \; then \; P \; else \; Q, \; S} \rangle \rightarrow \langle \text{TRY}(C, P, Q), \; \text{push}(\text{top}(S), \; S) \rangle
\end{align*}
\]
Here $S$ and $S'$ are stacks of graphs. The rule [try1] duplicates the top of the stack, and the TRY construct signals that the copy operation has happened. Repeated applications of the inference rule [try2] model the evaluation of the condition in a small-step fashion. If the condition loops, [try2] can be applied indefinitely, and we get an infinite transition sequence.

In the implementation of GP2, $P$ indeed loops. In the previous semantics however, $P$ gets stuck because $r1!$ diverges, which means that we cannot apply either of the inference rules [try1] or [try2] 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 $[P]G$ of all possible outcomes of the execution of $P$ on $G$. These outcomes can be a graph, the element fail, or $\bot$ which represents an infinite transition sequence. The previous semantic function uses $\bot$ as an outcome if the transition sequence gets stuck. However, there are problems with this approach.

Consider the program $P = \text{try } Loop \text{ then skip else skip}$, where $Loop = \{r1,r2\}!$. $r1$ is as previously defined, and $r2 : \emptyset \Rightarrow \emptyset$. 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 $\emptyset$, 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 $\{\bot, \emptyset, \emptyset, \emptyset, \emptyset, \ldots\}$. According to the previous semantics, however, the execution of $P$ on $\emptyset$ cannot get stuck since $Loop$ can always transition to a graph; and by the rules [try1] and [try2], the execution cannot diverge either. So $\bot \not\in [P]G = \{\emptyset, \emptyset, \emptyset, \emptyset, \ldots\}$.

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 $P = \text{try } \{r3,r2\}! \text{ then skip else skip}$, where $r3 : \emptyset \Rightarrow \emptyset$. It can diverge but is semantically equivalent to $Q = \text{try } r2 \text{ then skip else skip}$ since the previous semantics cannot detect that divergence. For instance, $[P]G = [Q]G = \{\emptyset\}$, but $[P]G$ should include $\bot$.

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 $\text{Loop}!$ 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 $P = \text{try } Loop \text{ then skip else skip}$, where $Loop = \{r1,r2\}!$, and the rules are as previously defined. We have $[Loop]G = \{\emptyset, \emptyset, \emptyset, \emptyset, \ldots, \bot\}$. In a transition sequence starting with $(P,G)$, 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 $\{\emptyset, \emptyset, \emptyset, \emptyset, \ldots\}$, 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 GP2 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.

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.

```
Main = delete!; {edge, loop}

delete(a,x,y:list) edge(a,x,y:list) loop(a,x:list)
```

where indeg(1)=0

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 xy, 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 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].
Figure 2: GP 2 Program Syntax

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 \( \text{indeg}(1) = 0 \) in the condition of \( \text{delete} \) in Figure 1 forbids the node \( g(1) \) to have incoming edges, where \( g(1) \) is the node in \( S \) corresponding to 1.

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
\[ \begin{align*}
[\text{call}1] & \quad G \Rightarrow RH \\
& \quad (R, G) \leadsto H \\
[\text{seq}1] & \quad \langle P, G \rangle \leadsto \langle P', H \rangle \\
& \quad \langle P', Q, G \rangle \leadsto \langle P', Q, H \rangle \\
[\text{seq}2] & \quad \langle P, G \rangle \leadsto H \\
& \quad \langle P, Q, G \rangle \leadsto \langle Q, H \rangle \\
\end{align*} \]

\[ \begin{align*}
[\text{call}2] & \quad G \not\Rightarrow RH \\
& \quad (R, G) \leadsto \text{fail} \\
[\text{seq}3] & \quad \langle P, G \rangle \leadsto \text{fail} \\
& \quad \langle P, Q, G \rangle \leadsto \text{fail} \\
\end{align*} \]

\[ \begin{align*}
[\text{if}1] & \quad \langle C, G \rangle \leadsto^+ H \\
& \quad \langle C \text{ if } P \text{ else } Q, G \rangle \leadsto \langle P, G \rangle \\
[\text{if}2] & \quad \langle C, G \rangle \leadsto^+ \text{ fail} \\
& \quad \langle C \text{ if } P \text{ else } Q, G \rangle \leadsto \langle Q, G \rangle \\
\end{align*} \]

\[ \begin{align*}
[\text{try}1] & \quad \langle C, G \rangle \leadsto^+ H \\
& \quad \langle \text{try } C \text{ then } P \text{ else } Q, G \rangle \leadsto \langle P, H \rangle \\
[\text{try}2] & \quad \langle C, G \rangle \leadsto^+ \text{ fail} \\
& \quad \langle \text{try } C \text{ then } P \text{ else } Q, G \rangle \leadsto \langle Q, G \rangle \\
\end{align*} \]

\[ \begin{align*}
[\text{alap}1] & \quad \langle P, G \rangle \leadsto^+ H \\
& \quad \langle P!, G \rangle \leadsto \langle P!, H \rangle \\
[\text{alap}2] & \quad \langle P, G \rangle \leadsto^+ \text{ fail} \\
& \quad \langle P!, G \rangle \leadsto \langle P!, G \rangle \\
\end{align*} \]

\[ \begin{align*}
[\text{alap}3] & \quad \langle P, G \rangle \leadsto^* \langle \text{break}, H \rangle \\
& \quad \langle P!, G \rangle \leadsto \langle H \rangle \\
\end{align*} \]

\[ \begin{align*}
[\text{break}] & \quad \langle \text{break}; P, G \rangle \leadsto \langle \text{break}, G \rangle \\
\end{align*} \]

(a) Inference rules for core commands

\[ \begin{align*}
[\text{or}1] & \quad \langle P \text{ or } Q, G \rangle \leadsto \langle P, G \rangle \\
[\text{or}2] & \quad \langle P \text{ or } Q, G \rangle \leadsto \langle Q, G \rangle \\
[\text{skip}] & \quad \langle \text{skip}, G \rangle \leadsto G \\
[\text{fail}] & \quad \langle \text{fail}, G \rangle \leadsto \text{fail} \\
[\text{if}3] & \quad \langle \text{if } C \text{ then } P, G \rangle \leadsto \langle \text{if } C \text{ then } P \text{ else } \text{skip}, G \rangle \\
[\text{try}3] & \quad \langle \text{try } C \text{ then } P, G \rangle \leadsto \langle \text{try } C \text{ then } P \text{ else } \text{skip}, G \rangle \\
[\text{try}4] & \quad \langle \text{try } C \text{ else } P, G \rangle \leadsto \langle \text{try } C \text{ then } \text{skip} \text{ else } P, G \rangle \\
[\text{try}5] & \quad \langle \text{try } C, G \rangle \leadsto \langle \text{try } C \text{ then } \text{skip} \text{ else } \text{skip}, G \rangle \\
\end{align*} \]

(b) Inference rules for derived commands

Figure 3: Previous GP 2 Semantics
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 a copy of \( G \). 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 \( / 0 \) (where \( / 0 \) 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 \( \Rightarrow \) over the following set:

\[
(\text{ComSeq} \times \mathcal{G}) \times ((\text{ComSeq} \times \mathcal{G}) \cup \mathcal{G} \cup \{\text{fail}\}).
\]

Here \( \mathcal{G} \) is the set of all GP 2 host graphs and ComSeq is the set of command sequences as defined in the syntax (Figure 2), and fail is an element representing 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 \( \Rightarrow \) is denoted by \( \Rightarrow^+ \), and the reflexive transitive closure by \( \Rightarrow^* \).

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 \( \mathcal{G} \) be the set of all host graphs and \( \mathcal{G}^\oplus = \mathcal{G} \cup \{\bot, \text{fail}\} \). These outcomes are described by the semantic function \([\cdot] : \text{ComSeq} \rightarrow (\mathcal{G} \rightarrow 2^{\mathcal{G}^\oplus}) \) which, for a command sequence \( P \) and a host graph \( G \), is defined as

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

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 can only be called inside a loop. If a break is in the condition of a branching statement, 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.

The rules in Figure 4a inductively define a transition relation \( \rightarrow \) over the following set:

\[
(\text{ExtComSeq} \times \mathcal{S}) \times ((\text{ExtComSeq} \times \mathcal{S}) \cup \mathcal{S} \cup \{\text{fail}\}),
\]

where \( \mathcal{S} \) is the set of all stacks of GP 2 host graphs (explained below). We call an element of the set \((\text{ExtComSeq} \times \mathcal{S}) \cup \mathcal{S} \cup \{\text{fail}\}\) an extended configuration, whereas \((\text{CommandSeq} \times \mathcal{S}) \cup \mathcal{S} \cup \{\text{fail}\}\) 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 \( \mathcal{S} \) 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 \( \text{pop}^2(S) = [G_1, G_3, \ldots, G_n] \), 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 \( \text{ExtComSeq} \). \( S \) stands for a graph stack in \( \mathcal{S} \), \( G \) represents a host graph, and \( R \) represents a rule set. We denote the transitive closure of \( \rightarrow \) by \( \rightarrow^+ \), and the reflexive transitive closure by \( \rightarrow^* \).

The inference rules inductively define the transition relation \( \rightarrow \). 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 \( \text{top}(S) \Rightarrow_R G \) or \( \text{top}(S) \not\Rightarrow_R G \), 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 \([\text{if}1]\) and \([\text{try}1]\) 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 \([\text{break}]\), 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 \( \text{fail} \), 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 [16].

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
(a) Inference rules for core commands

[call] \( \text{top}(S) \Rightarrow_R G \quad \text{top}(S) \not\Rightarrow_R \)

\[ \langle R, S \rangle \rightarrow \text{push}(G, \text{pop}(S)) \quad \langle R, S \rangle \rightarrow \text{fail} \]

[seq] \( \langle P, S \rangle \rightarrow \langle P', S' \rangle \quad \langle P, Q, S \rangle \rightarrow \langle P', Q, S' \rangle \)

[seq2] \( \langle P, S \rangle \rightarrow S' \quad \langle P, Q, S \rangle \rightarrow \langle Q, S' \rangle \)

[seq3] \( \langle P, S \rangle \rightarrow \text{fail} \quad \langle P, Q, S \rangle \rightarrow \text{fail} \)

[alap] \( \langle P!, S \rangle \rightarrow \langle \text{try } P \text{ then } P! \text{ else } \text{skip}, S \rangle \quad \langle \text{TRY(break}, P!, \text{skip}), S \rangle \rightarrow \text{pop2}(S) \)

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

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

[if2] \( \langle C, S \rangle \rightarrow \langle C', S' \rangle \quad \langle C, S \rangle \rightarrow \langle C', S' \rangle \)

\[ \langle \text{ITE}(C, P, Q), S \rangle \rightarrow \langle \text{ITE}(C', P, Q), S' \rangle \quad \langle \text{TRY}(C, P, Q), S \rangle \rightarrow \langle \text{TRY}(C', P, Q), S' \rangle \]

[if3] \( \langle C, S \rangle \rightarrow S' \quad \langle C, S \rangle \rightarrow S' \)

\[ \langle \text{ITE}(C, P, Q), S \rangle \rightarrow \langle P, \text{pop}(S') \rangle \quad \langle \text{TRY}(C, P, Q), S \rangle \rightarrow \langle P, \text{pop2}(S') \rangle \]

[if4] \( \langle C, S \rangle \rightarrow \text{fail} \quad \langle C, S \rangle \rightarrow \text{fail} \)

\[ \langle \text{ITE}(C, P, Q), S \rangle \rightarrow \langle Q, \text{pop}(S) \rangle \quad \langle \text{TRY}(C, P, Q), S \rangle \rightarrow \langle Q, \text{pop}(S) \rangle \]

(b) Inference rules for derived commands

[or] \( \langle P \text{ or } Q, S \rangle \rightarrow \langle P, S \rangle \quad \langle P \text{ or } Q, S \rangle \rightarrow \langle Q, S \rangle \)

[s] \( \langle \text{skip}, S \rangle \rightarrow S \quad \langle \text{fail}, S \rangle \rightarrow \text{fail} \)

[if] \( \langle \text{if } C \text{ then } P, S \rangle \rightarrow \langle \text{if } C \text{ then } P \text{ else } \text{skip}, S \rangle \)

[try] \( \langle \text{try } C \text{ then } P, S \rangle \rightarrow \langle \text{try } C \text{ then } P \text{ else } \text{skip}, S \rangle \)

[try] \( \langle \text{try } C \text{ else } P, S \rangle \rightarrow \langle \text{try } C \text{ then } \text{skip else } P, S \rangle \)

[try] \( \langle \text{try } C, S \rangle \rightarrow \langle \text{try } C \text{ then } \text{skip else } \text{skip}, S \rangle \)

Figure 4: Improved GP 2 Semantics
rule as a subscript. If the conclusion of $[\text{rule}_1]$ is used as a premise for $[\text{rule}_2]$, we denote it by $[\text{rule}_2]_{[\text{rule}_1]}$.

**Example 3.1.** Consider the program $P=r!$ and the rule $r: 1 \xrightarrow{} O \Rightarrow 1$. Let us examine a transition sequence of $P$ applied to the graph $O \xrightarrow{} O \Rightarrow O$, 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.

Next consider program $P'=\text{try}(\text{if } (r_1;r_1) \text{ then } (r_1;r_1))$ and the rule $r_1: 1 \xrightarrow{} O \Rightarrow O \Rightarrow 1$. A transition sequence of $P'$ applied to host graph $O \xrightarrow{} O \Rightarrow O$ 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 $r_1$ 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 $r_1$ 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 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 configuration 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.
\begin{align*}
\langle r!, [O \rightarrow O \rightarrow O] \rangle \\
\rightarrow [\text{alap}_1] \langle \text{try } r \text{ then } r! \text{ else skip}, [O \rightarrow O \rightarrow O] \rangle \\
\rightarrow [\text{try}_1] \langle \text{TRY}(r,r!,\text{skip}), [O \rightarrow O \rightarrow O, O \rightarrow O \rightarrow O] \rangle \\
\rightarrow [\text{call}_1]_{\text{try}_3} \langle r!, [O \rightarrow O] \rangle \\
\rightarrow [\text{alap}_1] \langle \text{try } r \text{ then } r! \text{ else skip}, [O \rightarrow O] \rangle \\
\rightarrow [\text{try}_1] \langle \text{TRY}(r,r!,\text{skip}), [O \rightarrow O, O \rightarrow O] \rangle \\
\rightarrow [\text{call}_1]_{\text{try}_3} \langle r!, [O] \rangle \\
\rightarrow [\text{alap}_1] \langle \text{try } r \text{ then } r! \text{ else skip}, [O] \rangle \\
\rightarrow [\text{try}_1] \langle \text{TRY}(r,r!,\text{skip}), [O, O] \rangle \\
\rightarrow [\text{call}_2]_{\text{try}_4} \langle \text{skip, [O]} \rangle \\
\rightarrow [\text{skip}] \langle [O] \rangle
\end{align*}

(a) Transition sequence of program P applied to graph $O \rightarrow O \rightarrow O$.

\begin{align*}
\langle \text{try}(\text{if}(r_1;r_1) \text{ then}(r_1;r_1)), [O \rightarrow O \rightarrow O] \rangle \\
\rightarrow [\text{try}_7] \langle \text{try } (\text{if } (r_1;r_1) \text{ then } (r_1;r_1)) \text{ then skip else skip}, [O \rightarrow O \rightarrow O] \rangle \\
\rightarrow [\text{try}_3] \langle \text{TRY}(\text{if } (r_1;r_1) \text{ then } (r_1;r_1), \text{skip, skip}), [O \rightarrow O \rightarrow O, O \rightarrow O \rightarrow O] \rangle \\
\rightarrow [\text{if}_6]_{\text{try}_2} \langle \text{TRY}(\text{if } (r_1;r_1) \text{ then } (r_1;r_1) \text{ else skip, skip, skip}), [O \rightarrow O \rightarrow O, O \rightarrow O \rightarrow O] \rangle \\
\rightarrow [\text{call}_1]_{\text{seq}_5} \langle \text{TRY}(\text{ITE}(r_1;r_1,r_1;r_1,\text{skip},\text{skip}),\text{skip, skip}), [O \rightarrow O, O \rightarrow O \rightarrow O, O \rightarrow O \rightarrow O] \rangle \\
\rightarrow [\text{call}_1]_{\text{try}_5} \langle \text{TRY}(\text{ITE}(r_1;r_1,r_1;\text{skip}),\text{skip, skip}), [O \rightarrow O, O \rightarrow O \rightarrow O, O \rightarrow O \rightarrow O] \rangle \\
\rightarrow [\text{call}_1]_{\text{try}_5} \langle \text{TRY}(r_1;\text{skip, skip}), [O \rightarrow O, O \rightarrow O \rightarrow O] \rangle \\
\rightarrow [\text{call}_1]_{\text{try}_5} \langle \text{TRY}(r_1;\text{skip, skip}), [O \rightarrow O, O \rightarrow O \rightarrow O] \rangle \\
\rightarrow [\text{call}_1]_{\text{try}_5} \langle \text{TRY}(\text{skip, skip}), [O \rightarrow O, O \rightarrow O \rightarrow O] \rangle \\
\rightarrow [\text{call}_1]_{\text{try}_5} \langle \text{skip, [O]} \rangle \\
\rightarrow [\text{skip}] \langle [O] \rangle
\end{align*}

(b) Transition sequence of program $P'$ applied to graph $O \rightarrow O \rightarrow O$.

Figure 5: Examples of transition sequences
Lemma 4.1 (Progress from Extended Configurations). Let \( \langle P, S \rangle \) be an extended configuration in ExtComSeq \( \times \mathcal{S} \). Then one of the following applies:

- \( \langle P, S \rangle \rightarrow \langle P', S' \rangle \) for some extended configuration \( \langle P', S' \rangle \).
- \( \langle P, S \rangle \rightarrow S' \) for some graph stack \( S' \in \mathcal{S} \).
- \( \langle P, S \rangle \rightarrow \text{fail} \).
- \( P \) is not a command sequence and contains a \text{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 \( \text{top}(S) \Rightarrow_p G \) or \( \text{top}(S) \not\Rightarrow_p \). So either \([\text{call}_1]\) or \([\text{call}_2]\) can be applied.

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

Case 3: \( P \) is fail, \text{skip} or an \text{or} statement. Then \([\text{fail}], [\text{skip}], \text{or} [\text{or}_1]\) can be applied respectively.

Case 4: \( P \) is of the form \text{if} \( P_1 \) \( \text{then} \) \( P_2 \) \( \text{else} \) \( P_3 \) \( \text{or} \) \( \text{try} \) \( P_1 \) \( \text{then} \) \( P_2 \) \( \text{else} \) \( P_3 \). Then \([\text{if}_1], [\text{try}_1]\) can be applied. If any then-clause or else-clause is omitted as specified by the syntax, \([\text{if}_5], [\text{try}_3], [\text{try}_6], \) or \([\text{try}_7]\) 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 \text{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 \text{break}. If \( P_1 \) is a sequential composition, let \( P_1 = P'_1; P''_1 \) where \( P'_1 \) is atomic. 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 atomic command of the condition.

- For the base case, assume \( P'_1 \) is not an ITE or TRY statement. Then \( P'_1 \) is atomic and either covered by cases 1 to 4 (\( P'_1 \) cannot be \text{break} since \( P \) contains no \text{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 \([\text{if}_2], [\text{if}_3], [\text{if}_4], [\text{try}_2], [\text{try}_3], \) or \([\text{try}_4]\) 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 atomic, i.e. not a sequential composition. We can apply \([\text{seq}_1], [\text{seq}_2], \) or \([\text{seq}_3]\) since \( \langle P_1, S \rangle \rightarrow \langle P'_1, S' \rangle \), \( \langle P_1, S \rangle \rightarrow S' \), or \( \langle P_1, S \rangle \rightarrow \text{fail} \) respectively by cases 1 to 5. \( \Box \)

Lemma 4.1 has a case where the extended command sequence contains a \text{break}. This is because for a transition sequence not to get stuck on a \text{break}, we need to start with a command sequence where the \text{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 \text{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 \( \langle P, [G] \rangle \rightarrow^* C \). 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 \text{break} Statement). Let \( \langle P, S \rangle \) be an extended configuration in ExtComSeq \( \times \mathcal{S} \) that is reachable and non-terminal. Suppose that \( P \) contains \text{break}. Then one of the following applies.

- There is an extended configuration \( \langle P', S' \rangle \) containing no \text{break} statement such that \( \langle P, S \rangle \rightarrow^* \langle P', S' \rangle \).
• There is a graph stack $S'$ such that $\langle P, S \rangle \rightarrow^+ S'$.

• $\langle P, S \rangle \rightarrow^+ \text{fail}$

**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 followed by $\text{try}$ to get $\text{TRY}(Q_1, Q_1!, \text{skip})$. We can then use Lemma 4.1 repeatedly as a premise for $\text{try}$ until we get $\text{TRY}(Q_3; Q_4; Q_1!, \text{skip})$, where $Q_3$ contains break and is atomic (i.e. not a sequential composition). 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 $\text{[or]}$ to either remove the break or lead to $\text{TRY}(\text{break}; Q_5, Q_1!, \text{skip})$. 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 $\text{TRY}(\text{break}; Q_5, Q_1!, \text{skip})$. We can now apply $\text{try}$ under the premise of $\text{break}$ to get $\text{TRY}(\text{break}, Q_1!, \text{skip})$. To this, we can apply $\text{[alap]}$, 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 $Q_3$ contain a single break. We can use the same arguments as in the base case in addition to $\text{seq}$ 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 $\text{seq}$ to remove the break.

Finally, assume that $\langle P, S \rangle$ does not satisfy context conditions, i.e. break statements must appear within a loop, and if one appears in the condition of a branching statement, the enclosing loop must be within the same condition. Since $\langle P, S \rangle$ is reachable, the latter condition is verified since the transition relation preserves it (loops can only be removed by inference rules, they cannot be “moved”). Suppose there is a break without an enclosing loop. This must be because $\text{alap}$ 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 $\text{alap}$ or $\text{try}$ 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 $\langle P, S \rangle \rightarrow C$ 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 $\times \mathcal{S}$. 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 $\times \mathcal{S}$.

1. If $\langle P, [G] \rangle \rightarrow^* \langle P', S \rangle$, where $\langle P', S \rangle$ is an extended configuration, then $|S| \geq 1$.
2. If $\langle P, [G] \rangle \rightarrow^* S$, 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 `[if1]` 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 `[alap]_2`, `[if3]`, `[if4]`, `[try]_3`, and `[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 $\langle P, [G] \rangle \rightarrow^* \langle P', S' \rangle \rightarrow S$. 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]`, `[call]_1`, or `[alap]_2`. To apply `[skip]$, $P'$ must be `skip` and $\#([skip]) = 0$, so $|S| = |S'| = 1$. To apply `[call]_1$, $P'$ must be rule set call, and hence cannot contain ITE or TRY, so $|S| = |S'| = 1$. To apply `[alap]_2$, $P'$ must be of the form TRYS (`break`, $P''$+, `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 `[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 `[alap]_2$, we have $S = pop2(S')$, 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 $\langle P, [G] \rangle \rightarrow^* (P', pop2(S))$ (an extended configuration) or $\langle P, [G] \rangle \rightarrow^* pop2(S)$ (a graph stack), then $|S| \geq 2$. This follows directly from Lemma 4.4 since $|pop2(S)| = |S| - 1$.

Let us now use Lemmata 4.3, 4.2, and 4.4 to describe what the possible outcomes of a transition sequence starting in a valid GP 2 program are.

**Proposition 4.5 (Outcomes of Transition Sequences).** Let $\langle P, [G] \rangle$ be a configuration in CommandSeq $\times \mathcal{S}$. Then one of the following applies:
• There is an infinite transition sequence \( \langle P_i[G] \rangle \rightarrow \langle P_1, S_1 \rangle \rightarrow \langle P_2, S_2 \rangle \rightarrow \ldots \) where \( \langle P_i, S_i \rangle \) is an extended configuration for all \( i \geq 1 \).
• \( \langle P_i[G] \rangle \rightarrow^+ [G'] \) for some host graph \( G' \).
• \( \langle P_i[G] \rangle \rightarrow^+ \text{fail} \).

Proof. Lemma 4.4 guarantees that if a transition sequence starts in \( \langle P_i[G] \rangle \) 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 \( \langle P_i[G] \rangle \). If we get a graph stack or fail, we fulfil the second or third case of this proposition. Otherwise we get an extended configuration \( \langle P'_i, S_i \rangle \) that contains no break.

Since there is now no break in either \( \langle P_i[G] \rangle \) or \( \langle P'_i, S_i \rangle \), 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. \( \square \)

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

\[
[J]G = \{X \in \mathcal{S} \cup \{\text{fail}\} | \langle P, G \rangle \rightarrow^+ X\} \cup \{\bot | P \text{ can diverge from } G\}.
\]

This function differs from the one presented in [17] and Section 2 since \( \bot \) is only used when \( P \) diverges, not when \( P \) gets stuck, i.e. there is a non-terminal configuration.

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 depends on the program only (and not on the size of the input). GP 2 has finite nondeterminism, but not bounded nondeterminism, which the following example illustrates. It also shows that bounded nondeterminism is a stronger property than finite nondeterminism.

Example 4.6. Consider the rule \( \sigma : 1 \circ \rightarrow \circ \) and the comb graph \( G_4 \) as shown in Figure 6.

![Figure 6: The comb graph \( G_4 \)](image)

are four possible matches for the left-hand side of rule \( \sigma \) in graph \( G_4 \), so applying the rule can result in four different non-isomorphic graphs, which is a finite amount. When applying \( \sigma \) 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. \( \square \)

Proposition 4.7 (Finite Nondeterminism). Let \( \gamma \in \text{ExtComSeq} \times \mathcal{G} \) be an extended configuration, and \( T_\gamma = \{\gamma' \mid \gamma \rightarrow \gamma' \in \text{ExtCommSeq} \times \mathcal{G} \} \). 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 $\gamma$ then there are exactly two configurations reachable from $\gamma$. 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 $\gamma$ 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 fulfills that definition as well.

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

Proof. Let $\gamma_0 = (P, [G])$. Then $T_{\gamma_0}^* = \{ \gamma \mid \gamma_0 \rightarrow^* \gamma \in \text{ExtCommSeq} \times \mathcal{G} \}$ 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 $T_{\gamma}$ is finite for all configurations $\gamma$ 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 $\gamma_0$, there is an infinite path starting from $\gamma_0$. By definition of the tree, this means there’s an infinite transition sequence starting with $\gamma_0$. By definition of the semantic function, we can conclude that $\bot \in [P]G$. 

Many references [8, 10, 18, 19] equate the concepts of finite and bounded nondeterminism and call it “bounded nondeterminism”. This is likely because imperative programming languages, unlike GP 2, usually satisfy both properties, so there is no need to distinguish between them.

**5 Conclusion**

We have introduced a new small-step 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 diverging computations. 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.

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].

**References**


