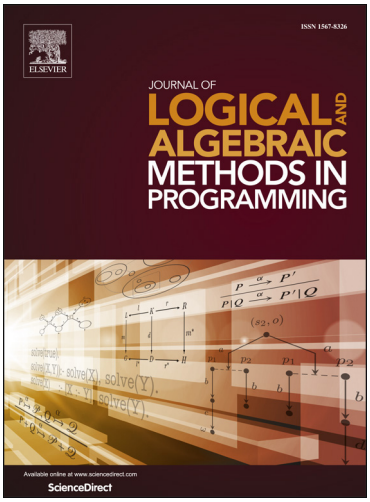
Article:
Plump, Detlef orcid.org/0000-0002-1148-822X (2017) From Imperative to Rule-based Graph Programs. Journal of Logical and Algebraic Methods in Programming. pp. 154-173. ISSN 2352-2216

https://doi.org/10.1016/j.jlamp.2016.12.001

Accepted Manuscript



From Imperative to Rule-based Graph Programs

Detlef Plump

PII: S2352-2208(16)30167-5
DOI: http://dx.doi.org/10.1016/j.jlamp.2016.12.001
Reference: JLAMP 157

To appear in: *Journal of Logical and Algebraic Methods in Programming*

Received date: 23 January 2015
Revised date: 10 May 2016
Accepted date: 5 December 2016

Please cite this article in press as: D. Plump, From Imperative to Rule-based Graph Programs, *J. Log. Algebraic Methods Program.* (2017), http://dx.doi.org/10.1016/j.jlamp.2016.12.001

Highlights

- Translation of high-level random access machines to rule-based graph programs.
- Proof that the translation is correct.
- Every computable graph function is directly computable by a GP 2 program.
- GP 2 without conditional rules and rule sets is computationally complete.

From Imperative to Rule-based Graph Programs

Detlef Plump

University of York, United Kingdom

Abstract

We discuss the translation of a simple imperative programming language, *high-level random access machines*, to the rule-based graph programming language GP 2. By proving the correctness of the translation and using GP 2 programs for encoding and decoding between arbitrary graphs and so-called register graphs, we show that GP 2 is computationally complete in a strong sense: every computable graph function can be directly computed with a GP 2 program which transforms input graphs into output graphs. Moreover, by carefully restricting the form of rules and control constructs in translated programs, we identify *simple* graph programs as a computationally complete sublanguage of GP 2. Simple programs use unconditional rules and abandon, besides other features, the non-deterministic choice of rules.

Keywords: Graph Programs, GP 2, Rule-based Programming, Computational Completeness, Random Access Machines, Graph Transformation

1. Introduction

The use of graphs to model dynamic structures is ubiquitous in computer science; prominent example areas include compiler construction, pointer programming, natural language processing, and model-driven software development. The behaviour of systems in such areas can be naturally captured by graph transformation rules specifying small state changes which are typically of constant size. Domain-specific languages based on graph transformation rules include AGG [21], GReAT [1], GROOVE [11], GrGen.Net [14] and PORGY [9]. This paper focusses on the graph programming language GP [18, 19] which aims to support formal reasoning on programs (see [20] for a Hoare-logic approach to verifying GP programs).

In this paper, we discuss the translation of a simple imperative programming language to GP 2. The motivation for this is threefold:

1. To prove that GP 2 is computationally complete, in the strong sense that graph functions are computable if and only if they can be directly computed with GP 2 programs which transform input graphs into output graphs.
2. To identify a computationally complete sublanguage of GP 2, by restricting the form of rules and control constructs in the target code.

3. To demonstrate in principle that imperative languages based on registers and assignments can be smoothly translated to a language based on graph transformation rules and pattern matching.

We use a prototypical imperative language called HIRAM, for *high-level random access machines*. The language differs from standard random access machines [2, 16] in that it provides while loops, if-then-else commands, and registers containing integer lists. HIRAM programs are translated into equivalent GP 2 code working on edge-less graphs with register-like nodes. In addition, target programs contain subprograms for encoding graphs as register graphs and decoding register graphs into normal graphs.

The rest of this paper is structured as follows. Section 2 briefly reviews the language GP 2. In Section 3, we introduce HIRAM as our prototypical imperative language. The translation of HIRAM to GP 2 is presented in Section 4, along with a correctness proof and the definition of simple graph programs. Section 5 gives GP 2 programs for encoding and decoding graphs, and states the main result of the paper, viz. that simple graph programs are computationally complete in a strong sense. Related work is discussed in Section 6. In Section 7, we consider future work and conclude. Appendix A defines GP 2 labels and rule conditions, Appendix B reviews the operational semantics of GP 2, and Appendix C shows the translation of HIRAM assignments with repeated addresses (which is omitted in Section 4 for readability reasons).

2. The Graph Programming Language GP 2

This section provides a brief introduction to GP 2, a domain-specific language for graphs. The syntax and semantics of GP 2 are defined in [19] (see also Appendix A and Appendix B). The language currently has two implementations, a compiler generating C code [4] and an interpreter for exploring the language's non-determinism [3].

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

Definition 1 (Graph)**.** Let $\mathcal{L}$ be a set of labels. A *graph* over $\mathcal{L}$ is a system $\langle V, E, s, t, l, m\rangle$, where V and E are finite sets of nodes (or vertices) and edges, $s\colon E \to V$ and $t\colon E \to V$ are source and target functions for edges, and $l\colon V \to \mathcal{L}$ and $m\colon E \to \mathcal{L}$ are labelling functions for nodes and edges.

The principal programming construct in GP 2 are conditional graph transformation rules labelled with expressions. For example, Figure 1 shows the declaration of the rule `bridge` which has six formal parameters of various types, 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.

The set of GP 2 labels is given by the syntactic category Label in the grammar of Figure A.12. Labels consist of an expression and an optional mark (explained below). Expressions are of type `int`, `char`, `string`, `atom` or `list`,

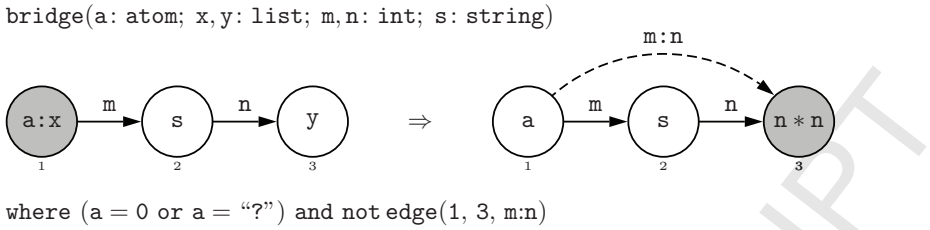


Figure 1: Declaration of a conditional rule

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 subtype hierarchy of GP 2 is shown in Figure 2 (both syntactically and semantically).

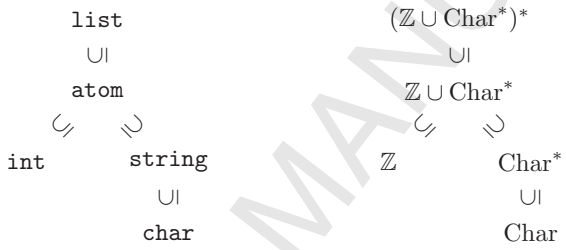


Figure 2: Subtype hierarchy

The concatenation of two lists x and y is written $x{:}y$[1], 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*. In Figure 1, the outermost nodes are marked by a grey shading and the dashed arrow between nodes 1 and 3 in the right-hand graph is a marked edge. Marks are convenient to highlight items in input or output graphs, and to record visited items during a graph traversal. For example, 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, see for example [7].

[1]Not to be confused with Haskell's ":" which adds an element to the beginning of a list.

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 `not edge(1, 3, m:n)` in the condition of Figure 1 forbids an edge in G from node $g(\texttt{1})$ to node $g(\texttt{3})$ with label $g(\texttt{m}){:}g(\texttt{n})$, where $g(\texttt{1})$ and $g(\texttt{3})$ are the nodes in S corresponding to `1` and `3`, and $g(\texttt{m})$ and $g(\texttt{n})$ are the labels in S corresponding to `m` and `n`.

Formally, GP 2 is based on a form of attributed graph transformation according to the so-called double-pushout approach [13, 8]. The grammar in Figure 3 gives the abstract syntax of GP 2 programs (see Appendix A for the syntax of graph labels). 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.

Prog	::=	Decl {Decl}
Decl	::=	RuleDecl | ProcDecl | MainDecl
ProcDecl	::=	ProcId '=' ['[' LocalDecl ']'] ComSeq
LocalDecl	::=	(RuleDecl | ProcDecl) {LocalDecl}
MainDecl	::=	`Main` '=' ComSeq
ComSeq	::=	Com {';' Com}
Com	::=	RuleSetCall | ProcCall
		| `if` ComSeq `then` ComSeq [`else` ComSeq]
		| `try` ComSeq [`then` ComSeq] [`else` ComSeq]
		| ComSeq '!' | ComSeq `or` ComSeq
		| '(' ComSeq ')' | `break` | `skip` | `fail`
RuleSetCall	::=	RuleId | '{' [RuleId {',' RuleId}] '}'
ProcCall	::=	ProcId

Figure 3: Abstract syntax of GP 2 programs

The call of a rule set $\{r_1, \ldots, r_n\}$ non-deterministically 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 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.

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 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 [19]. 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).

Example 1 (Recognising acyclic graphs). A graph is *acyclic* if it does not contain a directed cycle. The program in Figure 4 checks whether a graph G is acyclic and, if this is the case, executes program P on G; otherwise it executes program Q on G. The absence of cycles is checked by deleting as long as possible edges whose sources have no incoming edges, and testing whether any edges remain. This is correct since, by the condition of `delete`, a step $G \Rightarrow_{\texttt{delete}} H$ preserves both the absence of cycles and the presence of cycles. Moreover, a graph to which `delete` is not applicable is acyclic if and only if it is edge-less (every acyclic graph with edges must contain an edge to which `delete` is applicable). □

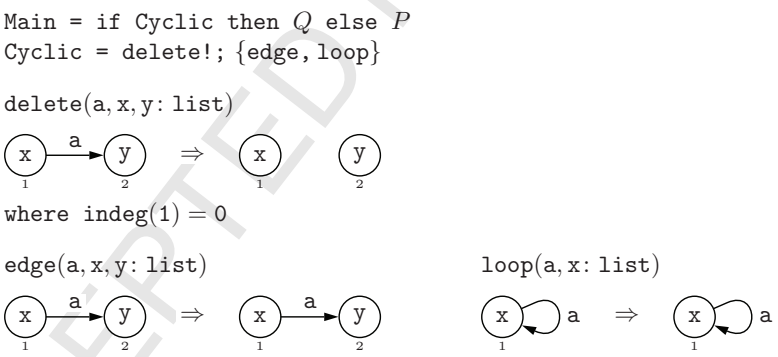


Figure 4: A program for recognising acyclic graphs

In general, the execution of a program on a host graph may result in different graphs, fail, or diverge. Let $\mathcal{G}$ be the set of all host graphs and $\mathcal{G}^{\oplus} = \mathcal{G} \cup \{\bot, \text{fail}\}$. The *semantics* of a program P is the function $[\![P]\!] \colon \mathcal{G} \to 2^{\mathcal{G}^{\oplus}}$, defined in Appendix B, which maps each host graph to the set of all possible outcomes.

3. HIRAM: High-level Random Access Machines

This section introduces a simple imperative programming language called HIRAM, for *high-level random access machines*. We define HIRAM by extending random access machines (RAMs), a computational model inspired by the von Neumann architecture of computers. RAMs represent real computers more accurately than Turing machines in that their operations resemble the instructions of real machines. In particular, all memory locations of a RAM can be accessed in constant time whereas the time needed to access a cell of a Turing machine grows with the distance of the cell from the tape head.

RAMs are used in algorithm analysis and complexity theory [2, 16], but there is no uniform model in the literature. The characteristic feature of a RAM is an infinite sequence of registers each of which can be randomly accessed. Typically, each register can hold an arbitrarily large integer.

In HIRAM, registers hold *lists* of integers and the language has operations for list manipulation. The conditional jump in conventional RAM models has been replaced with an if-then-else statement and a while loop. This does not affect the computational power since any program with jumps can be transformed into an equivalent jump-free program by using branching and loop statements [5]. Our enhancement of the RAM model with high-level constructs results in a simple programming language which is similar in flavour to the ALGOL 60 derivatives RAM-ALGOL [6] and Pidgin ALGOL [2].

The syntax of HIRAM is given in Figure 5, together with an explanation of the commands' effects on a state s (see also below). Integers are considered as lists of length one, and the colon operator is list concatenation. For example, `1:empty:-2` denotes the same list as `1:-2`. Registers are identified by non-negative integers called *addresses*. In Figure 5, the letters R, S and T stand for fixed addresses. Each register can hold an arbitrary list of integers.

HIRAM programs operate on an infinite memory of registers. Computations start with all registers empty (i.e., containing the empty list) with the possible exception of a finite number of input registers. Hence, at any stage of a computation, only finitely many registers will be non-empty.

Definition 2 (State)**.** A *state* is a mapping $s\colon \mathbb{N}_0 \to \mathbb{Z}^*$ with $s(n) = \lambda$ for all but finitely many addresses n.

Here λ is the empty list which is represented by the keyword `empty`. Given an address r, we say that $s(r)$ is the *contents* of register r. Let $\mathcal{S}$ be the set of all states. The result of executing a HIRAM program P is given by the semantic function $[\![P]\!]\colon \mathcal{S} \to \mathcal{S}^{\oplus}$ which maps an initial state either to a final state, to $\bot$ in case P diverges, or to the special element fail in case P fails.[2] We do not define $[\![P]\!]$ formally, the comments in Figure 5 and the following remarks should be sufficient.

[2]Given a class $\mathcal{A}$ of states or graphs, we write $\mathcal{A}^{\oplus}$ for $\mathcal{A} \cup \{\bot, \text{fail}\}$; functions $f\colon \mathcal{A} \to \mathcal{B}$ are extended to $f^{\oplus}\colon \mathcal{A}^{\oplus} \to \mathcal{B}^{\oplus}$ by $f^{\oplus}(\bot) = \bot$ and $f^{\oplus}(\text{fail}) = \text{fail}$.

Syntax		Comments
Int ::=	...	Integer numerals
R,S,T ::=	...	Non-negative integer numerals
List ::=	`empty`	Empty list
	| Int	Integers are lists of length 1
	| List ‘`:`’ List	Concatenation
B ::=	R ‘`=`’ S	True if $s(\mathrm{R}) = s(\mathrm{S})$; false otherwise
	| R ‘`>`’ S	True if $s(\mathrm{R}), s(\mathrm{S}) \in \mathbb{Z}$ and $s(\mathrm{R}) > s(\mathrm{S})$; false otherwise
Prog ::=	Prog ‘;’ Prog	Sequential composition
	| `if` B `then` Prog `else` Prog	Branching
	| `while` B `do` Prog	Loop
	| R ‘:=’ ‘$’ List	List $\mapsto$ R
	| R ‘:=’ S	$s(\mathrm{S}) \mapsto \mathrm{R}$
	| R ‘:=’ `head` S	$\mathrm{head}(s(\mathrm{S})) \mapsto \mathrm{R}$ (fails if $s(\mathrm{S}) = \lambda$)
	| R ‘:=’ `tail` S	$\mathrm{tail}(s(\mathrm{S})) \mapsto \mathrm{R}$ (fails if $s(\mathrm{S}) = \lambda$)
	| R ‘:=’ S ‘:’ T	$s(\mathrm{S}){:}s(\mathrm{T}) \mapsto \mathrm{R}$
	| R ‘:=’ ‘$*$’ S	$s(s(\mathrm{S})) \mapsto \mathrm{R}$ (fails if $s(\mathrm{S}) \notin \mathbb{N}_0$)
	| ‘$*$’ R ‘:=’ S	$s(\mathrm{S}) \mapsto s(\mathrm{R})$ (fails if $s(\mathrm{R}) \notin \mathbb{N}_0$)
	| R ‘:=’ `inc` S	$s(\mathrm{S}) + 1 \mapsto \mathrm{R}$ (fails if $s(\mathrm{S}) \notin \mathbb{Z}$)
	| R ‘:=’ `dec` S	$s(\mathrm{S}) - 1 \mapsto \mathrm{R}$ (fails if $s(\mathrm{S}) \notin \mathbb{Z}$)

Figure 5: HIRAM programs

An assignment R := S copies the list contained in register S to register R. In contrast, R := $List assigns the list following the dollar sign to R ($List indicates not to use List as an address). The assignments R := $*$S and $*$R := S use indirect addressing or *pointers*, that is, they interpret the contents of starred registers as addresses. In Figure 5, the notation $l \mapsto r$ means that list l is assigned to register r.

HIRAM programs can fail because of type errors. For example, the program `0 := inc 0` fails if $s(0)$ is not an integer. Similarly, `*0 := 1` fails if $s(0)$ is not an address. Failure of an assignment causes failure of the complete program so that, for instance, the program `1 := empty; while 0 = 0 do 0 := inc 1` fails on every state.

4. Translating HIRAM to GP 2

This section presents a translation of HIRAM to GP 2, where we are careful to generate code belonging to the sublanguage of *simple* GP 2 programs. In the next section, the translation will be used to show that every computable graph function can be computed by a simple program.

We translate HIRAM programs into GP 2 programs operating on edge-less graphs with register-like nodes. These nodes are marked grey to distinguish

them from ordinary nodes when arbitrary host graphs are encoded as register graphs (see next section).

Definition 3 (Register graph)**.** An edge-less graph in $\mathcal{G}$ is a *register graph* if nodes are marked grey and have labels of the form $r{:}l$, where r is an address and l a list, such that different nodes have different addresses.

Given a node v with label $r{:}l$ in a register graph, we refer to r as the address of v. The set of all register graphs is denoted by $\mathcal{G}_{\text{reg}}$ and $\varrho\colon \mathcal{G}_{\text{reg}} \to \mathcal{S}$ maps register graphs to corresponding HIRAM states. A register graph with labels $r_1{:}l_1, \ldots, r_n{:}l_n$ is mapped by ϱ to the state s defined by

$$s(r) = \begin{cases} l_i & \text{if } r = r_i \text{ for some } 1 \le i \le n, \\ \lambda & \text{otherwise.} \end{cases}$$

For example, Figure 6 shows a register graph and the corresponding state. It is easy to see that ϱ is surjective. Also, register graphs G and H with $\varrho(G) = \varrho(H)$ are isomorphic "up to nodes with empty registers". (That is, there is a label-preserving isomorphism $G^{\ominus} \to H^{\ominus}$ where $G^{\ominus}$ and $H^{\ominus}$ result from deleting all nodes with empty registers.)

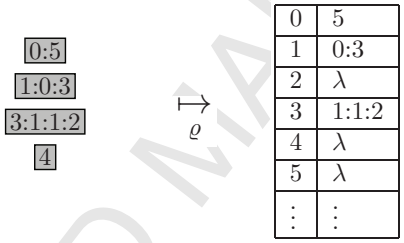


Figure 6: A register graph and the corresponding state

Since register graphs contain neither edges nor repeated node labels, they can be regarded as sets of lists. Consequently, the graph programs resulting from the following translation can be seen as rewrite programs for sets of lists.

The rest of this section presents the translation of HIRAM programs M to corresponding graph programs $\mathrm{P}_M = \texttt{Init}_M; \tau[\![M]\!]$, where $\texttt{Init}_M$ generates registers used in M that are not in the initial graph, and $\tau[\![M]\!]$ simulates the commands of M. The translation is based on the `try` command in GP 2 which, alternatively, could be replaced with the `if` command. The resulting code would look similar because the tests of `try` commands generated by the translation do not modify host graphs.

4.1. Register generation

The HIRAM model of computation assumes the existence of infinitely many registers, but a graph program needs to generate all used registers that are missing in the initial register graph. To avoid checking the existence of registers for every translated HIRAM command, the procedure $\texttt{Init}_M$ given below generates

nodes for all fixed addresses in M whose registers are not present in the input graph. This does not include addresses calculated in computations, which need special treatment.

Definition 4 (HIRAM addresses)**.** The *set of addresses* of a HIRAM program is inductively defined as follows:

- $\mathrm{Adr}(M;N) = \mathrm{Adr}(M) \cup \mathrm{Adr}(N)$
- $\mathrm{Adr}(\texttt{if}\ B\ \texttt{then}\ M\ \texttt{else}\ N) = \mathrm{Adr}(B) \cup \mathrm{Adr}(M) \cup \mathrm{Adr}(N)$
- $\mathrm{Adr}(\texttt{while}\ B\ \texttt{do}\ M) = \mathrm{Adr}(B) \cup \mathrm{Adr}(M)$
- $\mathrm{Adr}(\mathrm{R} = \mathrm{S}) = \mathrm{Adr}(\mathrm{R} > \mathrm{S}) = \{\mathrm{R}, \mathrm{S}\}$
- $\mathrm{Adr}(\mathrm{R} := \mathrm{S}) = \mathrm{Adr}(\mathrm{R} := *\mathrm{S}) = \mathrm{Adr}(*\mathrm{R} := \mathrm{S}) = \mathrm{Adr}(\mathrm{R} := \mathrm{op}\,\mathrm{S}) = \{\mathrm{R}, \mathrm{S}\}$ for $\mathrm{op} \in \{\texttt{head}, \texttt{tail}, \texttt{inc}, \texttt{dec}\}$
- $\mathrm{Adr}(\mathrm{R} := \mathrm{S{:}T}) = \{\mathrm{R}, \mathrm{S}, \mathrm{T}\}$
- $\mathrm{Adr}(\mathrm{R} := \$\mathrm{List}) = \{\mathrm{R}\}$

Given a HIRAM program M with $\mathrm{Adr}(M) = \{\mathrm{R}_1, \ldots, \mathrm{R}_n\}$, we define

$$\texttt{Init}_M = \texttt{try}\ \texttt{r}_1\ \texttt{else}\ \texttt{g}_1; \ldots; \texttt{try}\ \texttt{r}_n\ \texttt{else}\ \texttt{g}_n$$

where $\texttt{r}_1, \ldots, \texttt{r}_n$ and $\texttt{g}_1, \ldots, \texttt{g}_n$ are declared as follows:

$\texttt{r}_i(\texttt{x: list})$			$\texttt{g}_i$		
$\boxed{\mathrm{R}_i{:}\texttt{x}}$	$\Rightarrow$	$\boxed{\mathrm{R}_i{:}\texttt{x}}$	$\emptyset$	$\Rightarrow$	$\boxed{\mathrm{R}_i}$

For each address R_i in $\mathrm{Adr}(M)$, $\texttt{Init}_M$ checks whether there is a corresponding register node in the host graph; if this is not the case, rule $\texttt{g}_i$ generates that node (note that $\mathrm{R}_i = \mathrm{R}_i{:}\texttt{empty}$).

4.2. Translation of control constructs

In this subsection and the next, the commands of M are translated to corresponding GP 2 code. This subsection gives the translation of sequential composition, the if-then-else construct and the while loop; the next subsection treats assignment statements. The translation is presented as an operation τ which is inductively defined on the structure of HIRAM programs. We use R and S as placeholders for distinct addresses.

- $\tau[M;N] = \tau[M];\tau[N]$
- $\tau[\texttt{if}\ B\ \texttt{then}\ M\ \texttt{else}\ N] = \texttt{try}\ \tau[B]\ \texttt{then}\ \tau[M]\ \texttt{else}\ \tau[N]$

- $\tau[\mathrm{R} = \mathrm{S}] = \mathtt{eq}_{\mathrm{R,S}}$ with rule declaration

  $\mathtt{eq}_{\mathrm{R,S}}(\mathtt{x}, \mathtt{y}\colon \mathtt{list})$

  $\boxed{\mathrm{R}\!:\!\mathtt{x}}_{1}\ \boxed{\mathrm{S}\!:\!\mathtt{y}}_{2} \Rightarrow \boxed{\mathrm{R}\!:\!\mathtt{x}}_{1}\ \boxed{\mathrm{S}\!:\!\mathtt{y}}_{2}$

  $\mathtt{where}\ \mathtt{x} = \mathtt{y}$

- $\tau[\mathrm{R} = \mathrm{R}] = \mathtt{skip}$

- $\tau[\mathrm{R} > \mathrm{S}] = \mathtt{gt}_{\mathrm{R,S}}$ with rule declaration

  $\mathtt{gt}_{\mathrm{R,S}}(\mathtt{m}, \mathtt{n}\colon \mathtt{int})$

  $\boxed{\mathrm{R}\!:\!\mathtt{m}}_{1}\ \boxed{\mathrm{S}\!:\!\mathtt{n}}_{2} \Rightarrow \boxed{\mathrm{R}\!:\!\mathtt{m}}_{1}\ \boxed{\mathrm{S}\!:\!\mathtt{n}}_{2}$

  $\mathtt{where}\ \mathtt{m} > \mathtt{n}$

- $\tau[\mathrm{R} > \mathrm{R}] = \mathtt{fail}$

- $\tau[\mathtt{while}\ B\ \mathtt{do}\ M] =$
  $(\mathtt{try}\ \tau[B]\ \mathtt{then}\ \tau[M]\ \mathtt{else}\ \mathtt{fail})!;\ \mathtt{try}\ \tau[B]\ \mathtt{then}\ \mathtt{fail}\ \mathtt{else}\ \mathtt{skip}$

In the translation of the while loop, the second try command is needed in case $\tau[M]$ fails. In this case the !-loop terminates with the graph with which the loop's body was entered for the last time, hence we have to enforce failure.

4.3. Translation of assignments

This subsection shows the translation of assignment commands containing distinct addresses; the translation of assignments with repeated addresses is given in Appendix C. To demonstrate the treatment of indirect addressing, this subsection includes the slightly lengthy translation of $*\mathrm{R} := \mathrm{S}$. The assignment $\mathrm{R} := *\mathrm{S}$ has an analogous translation which is omitted.

- $\tau[\mathrm{R} := \$L] = \mathtt{as}_{\mathrm{R},\$L}$ with rule declaration

  $\mathtt{as}_{\mathrm{R},\$L}(\mathtt{x}\colon \mathtt{list})$

  $\boxed{\mathrm{R}\!:\!\mathtt{x}}_{1} \Rightarrow \boxed{\mathrm{R}\!:\!L}_{1}$

- $\tau[\mathrm{R} := \mathrm{S}] = \mathtt{as}_{\mathrm{R,S}}$ with rule declaration

  $\mathtt{as}_{\mathrm{R,S}}(\mathtt{x}, \mathtt{y}\colon \mathtt{list})$

  $\boxed{\mathrm{R}\!:\!\mathtt{x}}_{1}\ \boxed{\mathrm{S}\!:\!\mathtt{y}}_{2} \Rightarrow \boxed{\mathrm{R}\!:\!\mathtt{y}}_{1}\ \boxed{\mathrm{S}\!:\!\mathtt{y}}_{2}$

- $\tau[\mathrm{R} := \mathtt{head}\ \mathrm{S}] = \mathtt{hd}_{\mathrm{R,S}}$ with rule declaration

  $\mathtt{hd}_{\mathrm{R,S}}(\mathtt{x}, \mathtt{y}\colon \mathtt{list};\ \mathtt{n}\colon \mathtt{int})$

  $\boxed{\mathrm{R}\!:\!\mathtt{x}}_{1}\ \boxed{\mathrm{S}\!:\!\mathtt{n}\!:\!\mathtt{y}}_{2} \Rightarrow \boxed{\mathrm{R}\!:\!\mathtt{n}}_{1}\ \boxed{\mathrm{S}\!:\!\mathtt{n}\!:\!\mathtt{y}}_{2}$

- $\tau[\mathrm{R} := \texttt{tail}\ \mathrm{S}] = \texttt{tl}_{\mathrm{R,S}}$ with rule declaration

  $\texttt{tl}_{\mathrm{R,S}}(\texttt{x}, \texttt{y}\texttt{:}\ \texttt{list};\ \texttt{n}\texttt{:}\ \texttt{int})$

- $\tau[\mathrm{R} := \mathrm{S{:}T}] = \texttt{cnc}_{\mathrm{R,S,T}}$ with rule declaration

  $\texttt{cnc}_{\mathrm{R,S,T}}(\texttt{x}, \texttt{y}, \texttt{z}\texttt{:}\ \texttt{list})$

- $\tau[\mathrm{R} := \texttt{inc}\ \mathrm{S}] = \texttt{inc}_{\mathrm{R,S}}$ with rule declaration

  $\texttt{inc}_{\mathrm{R,S}}(\texttt{x}\texttt{:}\ \texttt{list};\ \texttt{n}\texttt{:}\ \texttt{int})$

- $\tau[\mathrm{R} := \texttt{dec}\ \mathrm{S}] = \texttt{dec}_{\mathrm{R,S}}$ with rule declaration

  $\texttt{dec}_{\mathrm{R,S}}(\texttt{x}\texttt{:}\ \texttt{list};\ \texttt{n}\texttt{:}\ \texttt{int})$

- $\tau[{*}\mathrm{R} := \mathrm{S}] = \texttt{try}\ \texttt{adr}_{\mathrm{R}}\ \texttt{then}\ \texttt{As}_{*\mathrm{R,S}}\ \texttt{else fail}$ with rule declaration

  $\texttt{adr}_{\mathrm{R}}(\texttt{n}\texttt{:}\ \texttt{int})$

  $\texttt{where}\ \texttt{n} > -1$

  and procedure declaration

  $$\texttt{As}_{*\mathrm{R,S}} = \texttt{try}\ \texttt{t1}_{*\mathrm{R,S}}\ \texttt{then}\ \texttt{as1}_{*\mathrm{R,S}}\ \texttt{else try}\ \texttt{t2}_{*\mathrm{R,S}}\ \texttt{then}\ \texttt{as2}_{*\mathrm{R,S}}\ \texttt{else try}\ \texttt{t3}_{*\mathrm{R,S}}\ \texttt{else}\ \texttt{gen}_{*\mathrm{R,S}}$$

  where the rules $\texttt{t}i_{*\mathrm{R,S}}$, $\texttt{as}i_{*\mathrm{R,S}}$ and $\texttt{gen}_{*\mathrm{R,S}}$ are declared as follows:

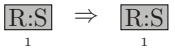
  $\texttt{t1}_{*\mathrm{R,S}}(\texttt{a}\texttt{:}\ \texttt{int};\ \texttt{x}, \texttt{y}\texttt{:}\ \texttt{list})$

  $\texttt{as1}_{*\mathrm{R,S}}(\texttt{a}\texttt{:}\ \texttt{int};\ \texttt{x}, \texttt{y}\texttt{:}\ \texttt{list})$

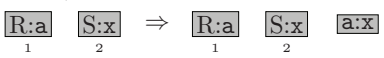
  $\texttt{t2}_{*\mathrm{R,S}}$

  $\texttt{as2}_{*\mathrm{R,S}}(\texttt{x}\texttt{:}\ \texttt{list})$

$\texttt{t3}_{*\mathrm{R,S}}$



$\texttt{gen}_{*\mathrm{R,S}}(\texttt{a: int; x: list})$



The code for $*\mathrm{R} := \mathrm{S}$ is lengthy because it does several checks on the contents of register R: whether it is an address, happens to be R or S, or is an address without corresponding register node. The latter is the case if neither $\texttt{t1}_{*\mathrm{R,S}}$ nor $\texttt{t2}_{*\mathrm{R,S}}$ nor $\texttt{t3}_{*\mathrm{R,S}}$ is applicable. We do not use the shorter definition $\texttt{As}_{*\mathrm{R,S}} = \texttt{try}\ \texttt{as1}_{*\mathrm{R,S}}\ \texttt{else}\ \texttt{try}\ \texttt{as2}_{*\mathrm{R,S}}\ \texttt{else}\ \texttt{gen}_{*\mathrm{R,S}}$ because the code violates the format of simple graph programs which will be introduced in Definition 5.

4.4. Correctness of the translation

Given a HIRAM program M, let P_M be the graph program $\texttt{Init}_M; \tau[M]$. We show that this translation is correct in that executing P_M on a register graph G and interpreting the result in $\mathcal{S}^{\oplus}$ has the same outcome as executing M on the state corresponding to G. We start by observing that P_M behaves deterministically on register graphs.

Lemma 1 (P_M is deterministic)**.** *For every HIRAM program M and register graph G, $[\![\mathrm{P}_M]\!]G$ is a singleton set.*[3]

Proof. It is easy to check that for each register graph G and each rule r in P_M, with the exception of the rules g_i in $\texttt{Init}_M$ and $\texttt{gen}_{*\mathrm{R,S}}$ and $\texttt{gen}_{*\mathrm{R,R}}$ in $\tau[\![M]\!]$, applying r to G either fails or produces a unique register graph. The rules g_i, $\texttt{gen}_{*\mathrm{R,S}}$ and $\texttt{gen}_{*\mathrm{R,R}}$ are included in `try` commands which ensure that the rules are applicable and produce unique register graphs. Moreover, P_M does not apply any rule sets with two or more rules. Since graph programs modify host graphs only by rule applications, it follows that executing P_M on a register graph either results in a unique register graph, fails or diverges. □

Since $[\![\mathrm{P}_M]\!]$ is single-valued on register graphs, we can consider it as a function $\mathcal{G}_{\mathrm{reg}} \to \mathcal{G}^{\oplus}_{\mathrm{reg}}$.

[3] The semantic function $[\![\mathrm{P}_M]\!] \colon \mathcal{G} \to 2^{\mathcal{G}^{\oplus}}$ is defined in Appendix B.

Theorem 1 (Correctness of P_M)**.** *For every HIRAM program M and register graph G, $\varrho^{\oplus}([\![\mathrm{P}_M]\!]G) = [\![M]\!]\varrho(G)$.*

$$\begin{array}{ccc} \mathcal{G}_{\mathrm{reg}} & \xrightarrow{[\![\mathrm{P}_M]\!]} & \mathcal{G}^{\oplus}_{\mathrm{reg}} \\ \varrho \downarrow & = & \downarrow \varrho^{\oplus} \\ \mathcal{S} & \xrightarrow[[\![M]\!]]{} & \mathcal{S}^{\oplus} \end{array}$$

Theorem 1 will be proved by induction on the structure of program M, where the induction base is given by assignment commands. The following lemma establishes the correctness in this case.

Lemma 2 (Correctness of translated assignments)**.** *Let c be a HIRAM assignment command and G a register graph such that for each r in* $\mathrm{Adr}(c)$*, G contains a node with address r. Then $\varrho^{\oplus}([\![\tau[c]]\!]G) = [\![c]\!]\varrho(G)$.*

Proof. We assume that c does not contain repeated addresses. The proof for assignments with repeated addresses is omitted because it is similar.

If c has the form R := \$$L$, R := S or R := S:T, then the rule $\tau[c]$ is applicable to G and, by inspection, assigns the correct value to register R.

Next, suppose that c has the form R := `head` S or R := `tail` S. If the node in G with address S holds a non-empty list, then rule $\tau[c]$ is applicable and has the desired effect. If the node holds the empty list, then the left-hand graph of $\tau[c]$ cannot be matched in G because variable `n` is of type `int`. Hence $\tau[c]$ fails on G, as required by the definition of `head` and `tail`.

Let now c have the form R := `inc` S or R := `dec` S. If G's node with address S holds a single integer, then $\tau[c]$ is applicable and assigns the incremented resp. decremented value to register R. If the node with address S holds the empty list or a list of length greater than one, then the rule cannot be matched because variable `n` is of type `int`. Hence $\tau[c]$ fails on G, as required by the definition of `inc` and `dec`.

Finally, let c be of the form $*$R := S. Then $\tau[c]$ is the program

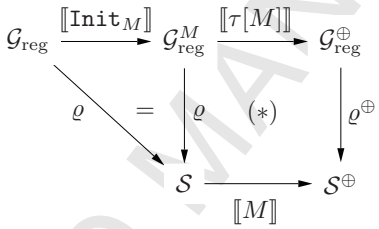
$$\texttt{try}\ \mathtt{adr}_{\mathrm{R}}\ \texttt{then}\ \mathtt{As}_{*\mathrm{R,S}}\ \texttt{else}\ \texttt{fail}$$

where rule $\mathtt{adr}_{\mathrm{R}}$ has no effect on G if the node with address R holds a non-negative integer, and fails otherwise. In the latter case, the else-branch of the try command will cause $\tau[c]$ to fail (in accordance with the definition of $*$R := S). The procedure $\mathtt{As}_{*\mathrm{R,S}}$ is defined by

$$\begin{aligned} \mathtt{As}_{*\mathrm{R,S}} = \texttt{try}\ \mathtt{t1}_{*\mathrm{R,S}}\ \texttt{then}\ \mathtt{as1}_{*\mathrm{R,S}} \\ \texttt{else try}\ \mathtt{t2}_{*\mathrm{R,S}}\ \texttt{then}\ \mathtt{as2}_{*\mathrm{R,S}} \\ \texttt{else try}\ \mathtt{t3}_{*\mathrm{R,S}}\ \texttt{else}\ \mathtt{gen}_{*\mathrm{R,S}}. \end{aligned}$$

Rule $\mathtt{t1}_{*\mathrm{R,S}}$ checks whether G contains a node with the address $\mathtt{a}$ stored in register R, such that $\mathtt{a}$ is different from R and S (the latter is ensured by injective rule matching). If the rule succeeds, rule $\mathtt{as1}_{*\mathrm{R,S}}$ copies the contents of register S to register $\mathtt{a}$. If $\mathtt{t1}_{*\mathrm{R,S}}$ fails, then rule $\mathtt{t2}_{*\mathrm{R,S}}$ checks whether $\mathtt{a} = \mathrm{R}$ (register R may contain its own address). If this is the case, then rule $\mathtt{as2}_{*\mathrm{R,S}}$ copies the contents of register S to register R. If $\mathtt{t2}_{*\mathrm{R,S}}$ fails, then rule $\mathtt{t3}_{*\mathrm{R,S}}$ checks whether $\mathtt{a} = \mathrm{S}$. If the rule succeeds, then register R contains address S so that G need not be modified. If $\mathtt{t2}_{*\mathrm{R,S}}$ fails, then by the previous checks it is clear that G does not contain a node with address $\mathtt{a}$. Hence rule $\mathtt{gen}_{*\mathrm{R,S}}$ generates a new node with address $\mathtt{a}$ and copies the contents of register S to register $\mathtt{a}$. Thus, overall the program $\tau[{*\mathrm{R}} := \mathrm{S}]$ correctly implements the assignment ${*\mathrm{R}} := \mathrm{S}$. □

Proof of Theorem 1. We show that the graph program $\mathrm{P}_M = \mathtt{Init}_M; \tau[M]$ simulates the HIRAM program M, which amounts to prove that $\varrho^{\oplus}([\![\mathrm{P}_M]\!]G) = [\![M]\!]\varrho(G)$ for every register graph G. Let $\mathcal{G}^M_{\mathrm{reg}}$ be the set of register graphs G such that for each r in $\mathrm{Adr}(M)$, G contains a node with address r. The proof boils down to showing the commutativity of square $(*)$ below:



This is because the left triangle is commutative: $\mathtt{Init}_M$ merely adds nodes with empty registers to a graph, which does not affect the corresponding state in $\mathcal{S}$.

We proceed by induction on the structure of M. The induction base is given by the HIRAM assignment commands, for which square $(*)$ is commutative by Lemma 2. Let now M be a composite command.

Case 1: M has the form $P; Q$ for some HIRAM programs P and Q. Then $\tau[M] = \tau[P; Q] = \tau[P]; \tau[Q]$. Consider any graph G in $\mathcal{G}^M_{\mathrm{reg}}$. By induction hypothesis,

$$\varrho^{\oplus}([\![\tau[P]]\!]G) = [\![P]\!]\varrho(G). \tag{1}$$

Case 1.1: $[\![\tau[P]]\!]G$ is some graph H. Then $H \in \mathcal{G}^M_{\mathrm{reg}}$ because no rule in M deletes any node or changes the address of any node. Hence, by induction hypothesis,

$$\varrho^{\oplus}([\![\tau[Q]]\!]H) = [\![Q]\!]\varrho(H). \tag{2}$$

Using (2) and (1), we obtain

$$\begin{array}{rcl} \varrho^{\oplus}([\![\tau[M]]\!]G) & = & \varrho^{\oplus}([\![\tau[P];\tau[Q]]\!]G) \\ & = & \varrho^{\oplus}([\![\tau[Q]]\!][\![\tau[P]]\!]G) \\ & = & \varrho^{\oplus}([\![\tau[Q]]\!]H) \\ & = & [\![Q]\!]\varrho(H) \\ & = & [\![Q]\!]\varrho^{\oplus}(H) \\ & = & [\![Q]\!]\varrho^{\oplus}([\![\tau[P]]\!]G) \\ & = & [\![Q]\!][\![P]\!]\varrho(G) \\ & = & [\![P;Q]\!]\varrho(G) \\ & = & [\![M]\!]\varrho(G) \end{array}$$

where the second and the last but one equality hold by the semantics of GP 2 and HIRAM, respectively.

Case 1.2: $[\![\tau[P]]\!]G \in \{\text{fail}, \bot\}$. The semantics of GP 2 and HIRAM propagate failure and divergence: if $X;Y$ is a graph program such that $\text{fail} \in [\![X]\!]N$ for some graph N, then $\text{fail} \in [\![X;Y]\!]N$; similarly, if $X;Y$ is a HIRAM program such that $[\![X]\!]N = \text{fail}$, then $[\![X;Y]\!]N = \text{fail}$. The same holds for divergence (replacing fail with $\bot$).

Hence, using (1), we get

$$\begin{array}{rcl} \varrho^{\oplus}([\![\tau[M]]\!]G) & = & \varrho^{\oplus}([\![\tau[P];\tau[Q]]\!]G) \\ & = & \varrho^{\oplus}([\![\tau[P]]\!]G) \\ & = & [\![P]\!]\varrho(G) \\ & = & [\![P;Q]\!]\varrho(G) \\ & = & [\![M]\!]\varrho(G) \end{array}$$

where $[\![P]\!]\varrho(G) = \varrho^{\oplus}([\![\tau[P]]\!]G) \in \{\text{fail}, \bot\}$ justifies the last but one equality.

Case 2: M has the form `if` B `then` P `else` Q for some HIRAM condition B and programs P and Q. Then $\tau[M] =$ `try` $\tau[B]$ `then` $\tau[P]$ `else` $\tau[Q]$. Consider any graph G in $\mathcal{G}^M_{\text{reg}}$. By induction hypothesis, we have

$$\varrho^{\oplus}([\![\tau[X]]\!]G) = [\![X]\!]\varrho(G) \text{ for } X = P, Q. \tag{3}$$

With the definition of τ it is easy to check that

$$[\![B]\!]\varrho(G) = \text{true if and only if } [\![\tau[B]]\!]G = G \tag{4}$$

and

$$[\![B]\!]\varrho(G) = \text{false if and only if } [\![\tau[B]]\!]G = \text{fail}. \tag{5}$$

Case 2.1: $[\![B]\!]\varrho(G) = \text{true}$. Then, by (4), $[\![\tau[B]]\!]G = G$ and hence

$$\begin{array}{rcl} \varrho^{\oplus}([\![\tau[M]]\!]G) & = & \varrho^{\oplus}([\![\texttt{try } \tau[B] \texttt{ then } \tau[P] \texttt{ else } \tau[Q]]\!]G) \\ & = & \varrho^{\oplus}([\![\tau[P]]\!]G) \\ & = & [\![P]\!]\varrho(G) \\ & = & [\![\texttt{if } B \texttt{ then } P \texttt{ else } Q]\!]\varrho(G) \\ & = & [\![M]\!]\varrho(G) \end{array}$$

where the third equality holds by (3).

Case 2.2: $[\![B]\!]\varrho(G) = \text{false}$. Then, by (5), $[\![\tau[B]]\!]G = \text{fail}$ and hence

$$\begin{array}{rcl} \varrho^{\oplus}([\![\tau[M]]\!]G) & = & \varrho^{\oplus}([\![\texttt{try}\ \tau[B]\ \texttt{then}\ \tau[P]\ \texttt{else}\ \tau[Q]]\!]G) \\ & = & \varrho^{\oplus}([\![\tau[Q]]\!]G) \\ & = & [\![Q]\!]\varrho(G) \\ & = & [\![\texttt{if}\ B\ \texttt{then}\ P\ \texttt{else}\ Q]\!]\varrho(G) \\ & = & [\![M]\!]\varrho(G) \end{array}$$

where the third equality holds by (3).

Case 3: M has the form `while` B `do` P for some HIRAM condition B and program P. Then

$$\tau[M] = (\texttt{try}\ \tau[B]\ \texttt{then}\ \tau[P]\ \texttt{else fail})!;\ \texttt{try}\ \tau[B]\ \texttt{then fail else skip}.$$

By induction hypothesis, we have

$$\varrho^{\oplus}([\![\tau[P]]\!]G) = [\![P]\!]\varrho(G) \tag{6}$$

for every graph G in $\mathcal{G}^M_{\text{reg}}$. Also, (4) and (5) are valid for each such graph.

We first show that for every $n \geq 0$ such that $[\![P]\!]^n\varrho(G)$ is well-defined (meaning that $[\![P]\!]^i\varrho(G) \in \mathcal{S}$ for $i = 0, \ldots, n-1$), $[\![\tau[P]]\!]^nG$ is also well-defined and

$$[\![P]\!]^n\varrho(G) = \varrho^{\oplus}([\![\tau[P]]\!]^nG). \tag{7}$$

We proceed by induction on n. If $n = 0$ then $[\![P]\!]^n\varrho(G) = \varrho(G) = \varrho^{\oplus}(G) = \varrho^{\oplus}([\![\tau[P]]\!]^nG)$. The case $n = 1$ is given by (6). If $n > 1$ then, by induction hypothesis, $[\![\tau[P]]\!]^{n-1}G$ is well-defined and $[\![P]\!]^{n-1}\varrho(G) = \varrho^{\oplus}([\![\tau[P]]\!]^{n-1}G)$. Moreover, since $[\![P]\!]^{n-1}\varrho(G)$ is a state, $\varrho^{\oplus}([\![\tau[P]]\!]^{n-1}G)$ is a graph in $\mathcal{G}^M_{\text{reg}}$. Thus

$$\begin{array}{rcl} [\![P]\!]^n\varrho(G) & = & [\![P]\!][\![P]\!]^{n-1}\varrho(G) \\ & = & [\![P]\!]\varrho^{\oplus}([\![\tau[P]]\!]^{n-1}G) \\ & = & [\![P]\!]\varrho([\![\tau[P]]\!]^{n-1}G) \\ & = & \varrho^{\oplus}([\![\tau[P]]\!][\![\tau[P]]\!]^{n-1}G) \\ & = & \varrho^{\oplus}([\![\tau[P]]\!]^nG) \end{array}$$

where the last but one equality holds by (6).

Next, we consider the possible outcomes of executing M on $\varrho(G)$.

Case 3.1: $[\![M]\!]\varrho(G) \in \mathcal{S}$. Then, by the semantics of HIRAM, there is some $n \geq 0$ such that

$$[\![M]\!]\varrho(G) = [\![\texttt{while}\ B\ \texttt{do}\ P]\!]\varrho(G) = [\![P]\!]^n\varrho(G) \tag{8}$$

and

$$[\![B]\!][\![P]\!]^i\varrho(G) = \text{true for } i = 0, \ldots, n-1 \text{ and } [\![B]\!][\![P]\!]^n\varrho(G) = \text{false}. \tag{9}$$

Applying (7) to (8) gives

$$[\![M]\!]\varrho(G) = \varrho([\![\tau[P]]\!]^nG). \tag{10}$$

Using (4) and (5), (9) implies

$$[\![\tau[B]]\!][\![\tau[P]]\!]^i G = G \text{ for } i = 0, \ldots, n-1 \text{ and } [\![\tau[B]]\!][\![\tau[P]]\!]^n G = \text{fail}. \tag{11}$$

We show that

$$[\![\tau[P]]\!]^n G = [\![\tau[M]]\!] G. \tag{12}$$

Let $\tau[M] = \texttt{Loop};\texttt{Test}$ with

$$\texttt{Loop} = (\texttt{try } \tau[B] \texttt{ then } \tau[P] \texttt{ else fail})!$$

and

$$\texttt{Test} = \texttt{try } \tau[B] \texttt{ then fail else skip}.$$

By the semantics of GP 2 and (11),

$$[\![\texttt{Loop}]\!] G = [\![\tau[P]]\!]^n G \text{ and } [\![\texttt{Test}]\!][\![\tau[P]]\!]^n G = [\![\tau[P]]\!]^n G$$

and hence $[\![\tau[M]]\!] G = [\![\texttt{Loop};\texttt{Test}]\!] G = [\![\tau[P]]\!]^n G$, proving (12). Combining (10) and (12), we obtain

$$[\![M]\!]\varrho(G) = \varrho([\![\tau[P]]\!]^n G) = \varrho([\![\tau[M]]\!] G).$$

Case 3.2: $[\![M]\!]\varrho(G) = \text{fail}$. Then, by the semantics of HIRAM, there is some $n \geq 1$ such that

$$[\![P]\!]^i \varrho(G) \in \mathcal{S} \text{ for } i = 0, \ldots, n-1 \text{ and } [\![P]\!]^n \varrho(G) = \text{fail} \tag{13}$$

and

$$[\![B]\!][\![P]\!]^i \varrho(G) = \text{true for } i = 0, \ldots, n-1. \tag{14}$$

Applying (7) to (13) gives $\varrho^{\oplus}([\![\tau[P]]\!]^i G) \in \mathcal{S}$ for $i = 0, \ldots, n-1$ and $\varrho^{\oplus}([\![\tau[P]]\!]^n G) = \text{fail}$, and hence

$$[\![\tau[P]]\!]^i G \in \mathcal{G}_{\text{reg}} \text{ for } i = 0, \ldots, n-1 \text{ and } [\![\tau[P]]\!]^n G = \text{fail}$$

and applying (4) to (14) gives

$$[\![\tau[B]]\!][\![\tau[P]]\!]^i G = G \text{ for } i = 0, \ldots, n-1.$$

Thus, by the semantics of GP 2,

$$[\![\texttt{Loop}]\!] G = [\![\tau[P]]\!]^{n-1} G \text{ and } [\![\texttt{Test}]\!][\![\tau[P]]\!]^{n-1} G = \text{fail}$$

and hence $[\![\texttt{Loop};\texttt{Test}]\!] G = \text{fail}$. It follows

$$[\![M]\!]\varrho(G) = \text{fail} = \varrho^{\oplus}(\text{fail}) = \varrho^{\oplus}([\![\texttt{Loop};\texttt{Test}]\!] G) = \varrho^{\oplus}([\![\tau[M]]\!] G).$$

Case 3.3: $[\![M]\!]\varrho(G) = \bot$. By the semantics of HIRAM, M diverges either because the loop body P is executed infinitely often or because at some point, P diverges.

Case 3.3.1: For every $i \geq 0$, $[\![B]\!][\![P]\!]^i \varrho(G) = \text{true}$ and $[\![P]\!]^i \varrho(G) \in \mathcal{S}$. Then (4) and (7) imply that

$$\text{for every } i \geq 0,\ [\![\tau[B]]\!][\![\tau[P]]\!]^i G = G \text{ and } [\![\tau[P]]\!]^i G \in \mathcal{G}_{\text{reg}}.$$

With the semantics of GP 2 follows $[\![\texttt{Loop}]\!]G = \bot = [\![\texttt{Loop}; \texttt{Test}]\!]G$. Thus

$$[\![M]\!]\varrho(G) = \bot = \varrho^{\oplus}(\bot) = \varrho^{\oplus}([\![\texttt{Loop}; \texttt{Test}]\!]G) = \varrho^{\oplus}([\![\tau[M]]\!]G).$$

Case 3.3.2: There is some $n \geq 1$ such that $[\![P]\!]^n \varrho(G) = \bot$ and for $i = 0, \ldots, n-1$, $[\![P]\!]^i \varrho(G) \in \mathcal{S}$ and $[\![B]\!][\![P]\!]^i \varrho(G) = \text{true}$. Using (7) and (4), and the fact that $\varrho^{\oplus}$ maps $\bot$ to $\bot$ and graphs to states, we obtain

$$[\![\tau[P]]\!]^n G = \bot$$

and

$$\text{for } i = 0, \ldots, n-1,\ [\![\tau[P]]\!]^i G \in \mathcal{G}_{\text{reg}} \text{ and } [\![\tau[B]]\!][\![\tau[P]]\!]^i G = G.$$

Then, by the semantics of GP 2,

$$[\![\texttt{Loop}]\!]G = \bot$$

because Loop "gets stuck" from $[\![\tau[P]]\!]^{n-1} G$ (see the definition of the semantic function in Appendix B). Thus

$$[\![M]\!]\varrho(G) = \bot = \varrho^{\oplus}(\bot) = \varrho^{\oplus}([\![\texttt{Loop}]\!]G) = \varrho^{\oplus}([\![\texttt{Loop}; \texttt{Test}]\!]G) = \varrho^{\oplus}([\![\tau[M]]\!]G).$$

This concludes the proof of Theorem 1. □

4.5. Restricted graph programs

In this subsection, we define the GP 2 sublanguage of simple programs and the even more restricted class of basic programs:

$$\text{Basic programs} \subset \text{Simple programs} \subset \text{GP 2}$$

Program P_M as defined in Subsections 4.1 to 4.3 is up to syntactic sugar a basic program. Hence, by Theorem 1, basic graph programs are computationally complete in the sense that they can simulate HIRAM programs. However, the rules of basic programs neither contain edges nor delete nodes, and thus there are graph functions $f\colon \mathcal{G} \to \mathcal{G}$ that are computable in an intuitive sense but cannot be directly computed by basic programs. For example, the function mapping every graph to the empty graph is of this kind: there is no basic program that transforms non-empty input graphs into the empty graph.

Therefore we consider a stronger notion of computational completeness in the next section, namely that every computable graph function can be directly computed by some graph program. We will show that simple GP 2 programs enjoy this property. The proof uses subprograms for encoding and decoding graphs which require some non-basic rules.

Definition 5 (Simple graph program)**.** *Simple* GP 2 programs adhere to the following grammar:

Prog	::=	`main` '=' ComSeq {RuleDecl}
ComSeq	::=	Com {';' Com}
Com	::=	UncondRuleId | ComSeq '!' | `fail`
		| `try` TestRuleId `then` ComSeq `else` ComSeq

Here UncondRuleId is the call of an unconditional rule and TestRuleId is the call of a rule $L \Rightarrow L$ that possibly has a condition of the form `x` = `y` or `m` > `n`, where L is an edge-less graph of at most three nodes which are preserved (numbered).

Hence, simple programs abandon rule sets and if-then-else statements, and modify graphs by unconditional rules. The test T of any command `try` T `then` P `else` Q merely checks the occurrence of at most three nodes, without changing the host graph.

Definition 6 (Basic graph program)**.** A simple GP 2 program is *basic* if all rules consist of edge-less graphs of at most three nodes and do not delete nodes.

The translation of HIRAM programs defined in Subsections 4.1 to 4.3 produces basic graph programs, possibly with some syntactic sugar added. The resulting programs can easily be desugared so that they conform to Definition 6: a phrase `try` T `else` P is equivalent to `try` T `then erule else` P, where `erule` is declared as $\emptyset \Rightarrow \emptyset$; the `skip` command is equivalent to a call of `erule`; and the procedures in the code for assignments with starred registers can be dropped if procedure calls are replaced with the command sequences they stand for.

In view of Theorem 1, it is clear that basic graph programs inherit some undecidability results from random access machines.

Corollary (Undecidable properties of basic programs)**.** *The following problems are undecidable in general:*

Divergence:	*Given as input a basic graph program P and an edge-less host graph G, can P diverge from G?*
Reachability:	*Given as input a basic graph program P and edge-less host graphs G and H, is H contained in $[\![P]\!]G$?*

Proof. If either of these properties were decidable, Theorem 1 would allow to decide the corresponding property for random access machines (with graphs replaced by states). But both properties are known to be undecidable for RAMs. □

5. Computational Completeness of Simple Graph Programs

To define computable graph functions, we represent host graphs as HIRAM states. A function f on graphs will be considered as computable if there exists

a HIRAM program that transforms each state representing a graph G into a state representing $f(G)$.

In representing graphs as states, we face the problem of what to do with character strings in labels (as HIRAM is based on integer lists). One could either encode strings as integer lists or, more in line with our overall approach, replace HIRAM lists with GP 2 lists and add some string operations. Both options would result in extra technicalities though, without providing new insights. Hence, for simplicity, we will restrict ourselves to functions on graphs labelled with integer lists.

We represent graphs as sketched in Figure 7. Register 0 holds the size of the graph and is followed by node registers and edge registers. The entries of node registers start with 0 while the entries of edge registers start with 1 (to distinguish nodes and edges). Edge registers contain, besides the label, the addresses of the edge's source and target.

0	$m+n$	graph size
1	$0 : \langle\text{label}\rangle$	node 1
2	$0 : \langle\text{label}\rangle$	node 2
$\vdots$	$\vdots$	$\vdots$
m	$0 : \langle\text{label}\rangle$	node m
$m+1$	$1 : \langle\text{source}\rangle : \langle\text{target}\rangle : \langle\text{label}\rangle$	edge 1
$m+2$	$1 : \langle\text{source}\rangle : \langle\text{target}\rangle : \langle\text{label}\rangle$	edge 2
$\vdots$	$\vdots$	$\vdots$
$m+n$	$1 : \langle\text{source}\rangle : \langle\text{target}\rangle : \langle\text{label}\rangle$	edge n
$m+n+1$	λ	
$\vdots$	$\vdots$	

Figure 7: HIRAM representation of a graph with m nodes and n edges

Definition 7 (Graph state)**.** A state s is a *graph state* if there are $m, n \geq 0$ (the numbers of nodes and edges) such that (1) $s(0) = m + n$, (2) for $i = 1, \ldots, m$, $s(i) = 0{:}l$ with $l \in \mathbb{Z}^*$, (3) for $i = m+1, \ldots, m+n$, $s(i) = 1{:}a{:}b{:}l$ with $a, b \in \{1, \ldots, m\}$ and $l \in \mathbb{Z}^*$, and (4) $s(i) = \lambda$ for all $i > m + n$.

We write $\mathcal{S}_{\text{gra}}$ for the set of all graph states and $\mathcal{G}_{\mathbb{Z}}$ for the set of all host graphs labelled with integer lists. The function $\gamma \colon \mathcal{S}_{\text{gra}} \to \mathcal{G}_{\mathbb{Z}}$ maps graph states to corresponding graphs. For example, Figure 8 shows a graph state in the middle and the corresponding graph on the right. Note that γ is surjective if we consider graphs up to isomorphism. More precisely, for every graph G in $\mathcal{G}_{\mathbb{Z}}$ there is a graph state s such that $\gamma(s)$ is isomorphic to G.

Definition 8 (Computable graph function)**.** A function $f \colon \mathcal{G}_{\mathbb{Z}} \to \mathcal{G}_{\mathbb{Z}}^{\oplus}$ is *computable* if there exists a HIRAM program M such that for all $s \in \mathcal{S}_{\text{gra}}$, $[\![M]\!]s \in \mathcal{S}_{\text{gra}}^{\oplus}$ and $\gamma^{\oplus}([\![M]\!]s) = f(\gamma(s))$. See Figure 9.

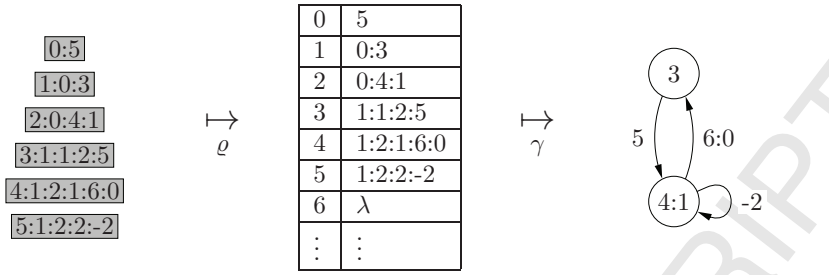


Figure 8: A graph state (in the middle) and the corresponding graph (on the right)

In particular, this requires M to fail on all graphs G with $f(G) = \text{fail}$, and to diverge on all G with $f(G) = \bot$.

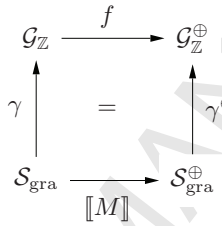


Figure 9: Computable graph function

The main result of this paper will show that f can be computed by a graph program `Enc`; P_M; `Dec`, where `Enc` encodes graphs as register graphs, P_M simulates M, and `Dec` decodes register graphs back to graphs. First, we present the programs `Enc` and `Dec`.

Graph encoding. The procedure `Enc` in Figure 10 nondeterministically picks nodes and edges to convert them into register nodes.

Let $\mathfrak{G} = \varrho^{-1}(\mathcal{S}_{\text{gra}})$ be the set of register graphs representing graph states. It is easy to see that `Enc` transforms graphs in $\mathcal{G}_{\mathbb{Z}}$ into graphs in $\mathfrak{G}$, hence $[\![\texttt{Enc}]\!]$ can be considered as a relation between $\mathcal{G}_{\mathbb{Z}}$ and $\mathfrak{G}$. It is not a function because the assignment of nodes and edges to registers is nondeterministic. However, graph states representing the same graph are identified by γ.

Lemma 3. *For every graph G in $\mathcal{G}_{\mathbb{Z}}$, $\gamma(\varrho([\![\texttt{Enc}]\!]G)) = G$.*

Graph decoding. The procedure `Dec` in Figure 11 decodes register graphs back into unique graphs. Note that after termination of the loop `regtoedge!`, all registers representing non-loop edges have been converted to edges and hence `regtoloop!` only converts registers representing loops.

Enc = reg0; nodetoreg!; edgetoreg!; looptoreg! with rule declarations

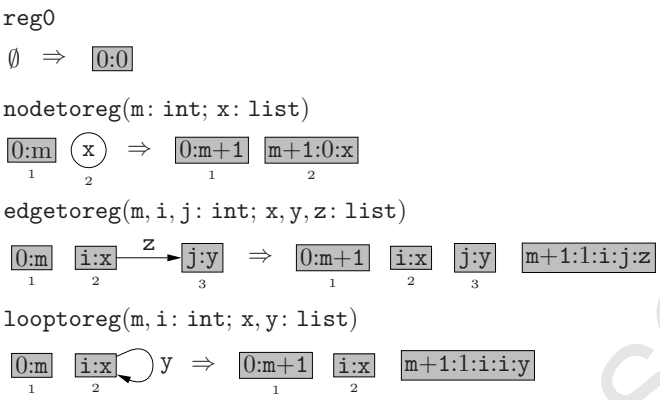


Figure 10: Procedure Enc for graph encoding

Since Dec transforms register graphs into unique graphs in $\mathcal{G}_{\mathbb{Z}}$, $[\![\texttt{Dec}]\!]$ can be considered as a function $\mathfrak{G} \to \mathcal{G}_{\mathbb{Z}}$. We write $[\![\texttt{Dec}]\!]^{\oplus}$ for its extension to $\mathfrak{G}^{\oplus}$ and $\mathcal{G}_{\mathbb{Z}}^{\oplus}$.

Lemma 4. *For every register graph G in $\mathfrak{G}$, $[\![\texttt{Dec}]\!]G = \gamma(\varrho(G))$.*

Dec = 0reg; regtoedge!; regtoloop!; regtonode! with rule declarations

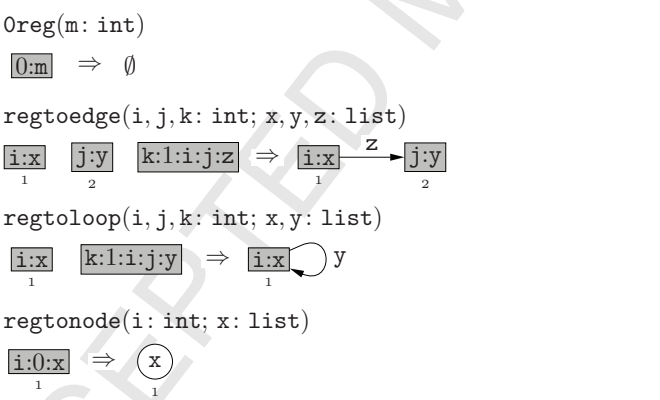


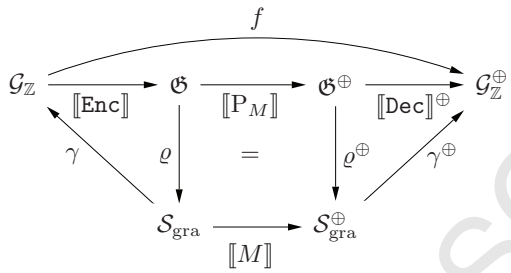
Figure 11: Procedure Dec for graph decoding

For readability reasons, we have introduced Enc and Dec as procedures. But their command sequences adhere to the restrictions of simple programs and hence simple programs with these procedure calls can easily be transformed to obey Definition 5. We now state our main result.

Theorem 2. *For every computable function $f\colon \mathcal{G}_{\mathbb{Z}} \to \mathcal{G}_{\mathbb{Z}}^{\oplus}$ there exists a simple graph program P such that for all G in $\mathcal{G}_{\mathbb{Z}}$, $[\![P]\!]G = f(G)$.*

Note that this is a strong form of completeness: program P computes f directly, not just a corresponding function on register graphs. This is an instance of Weihrauch's abstract concept of *strong relative computability* [24].

Proof of Theorem 2. Consider the following diagram:



By Definition 8 there exists a HIRAM program M whose semantics can be restricted to $[\![M]\!]\colon \mathcal{S}_{\mathrm{gra}} \to \mathcal{S}_{\mathrm{gra}}^{\oplus}$, satisfying $\gamma^{\oplus}([\![M]\!]s) = f(\gamma(s))$ for each $s \in \mathcal{S}_{\mathrm{gra}}$. Given any register graph G in $\mathfrak{G}$, Theorem 1 gives $\varrho^{\oplus}([\![\mathrm{P}_M]\!]G) = [\![M]\!]\varrho(G)$ and hence $[\![\mathrm{P}_M]\!]$ can be considered as a function from $\mathfrak{G}$ to $\mathfrak{G}^{\oplus}$.

The plan is to show that the program $\texttt{Enc}; \mathrm{P}_M; \texttt{Dec}$ computes f, that is, that for every graph G in $\mathcal{G}_{\mathbb{Z}}$, $[\![\texttt{Dec}]\!]^{\oplus}[\![\mathrm{P}_M]\!][\![\texttt{Enc}]\!]G = f(G)$. Consider any G' in $\mathfrak{G}$ such that $[\![\texttt{Enc}]\!]G = G'$. Then, by Lemma 4, Theorem 1 and Definition 8, $[\![\texttt{Dec}]\!]^{\oplus}[\![\mathrm{P}_M]\!]G' = \gamma^{\oplus}(\varrho^{\oplus}([\![\mathrm{P}_M]\!]G')) = \gamma^{\oplus}([\![M]\!]\varrho(G') = f(\gamma(\varrho(G'))))$. Thus, by Lemma 3, $[\![\texttt{Dec}]\!]^{\oplus}[\![\mathrm{P}_M]\!][\![\texttt{Enc}]\!]G = f(\gamma(\varrho([\![\texttt{Enc}]\!]G))) = f(G)$. □

6. Related Work

Computability-theoretic aspects of unrestricted graph transformation approaches have received little attention in the literature. A fundamental result due to Uesu is that in the double-pushout approach, every recursively enumerable set of graphs can be generated by some graph grammar [23]. Here a set of graphs is said to be recursively enumerable if there exists a Turing machine that enumerates all graphs in the set, using some fixed encoding of graphs as strings.

As to the computability of functions or relations on graphs, it is folklore in the field of graph transformation that every Turing machine can be simulated by a set of double-pushout graph transformation rules. Such a simulation is presented, for example, in [15]. However, the ability to simulate a Turing machine on graphs representing machine configurations does not imply that every computable graph function can be directly computed by a set of graph transformation rules. We refer to the discussion in Subsection 4.5.

To the author's best knowledge, [12] has been the only paper so far proving the computational completeness of a graph transformation language in the strong sense of the present paper. In that paper, a rudimentary language based on graph transformation rules is used to encode input graphs as string-like

graphs, simulate a Turing machine on the latter, and decode the resulting graphs. The language's control constructs are application of a set of rules, sequential composition, and as-long-as-possible iteration. It is also shown that without sequential composition, the language is not complete in the strong sense.

The main differences to the present work are as follows. Our completeness results (Theorem 1 & 2) hold for programs with rules of very restricted shape and size. In contrast, almost all of the rules in [12] contain both nodes and edges, many of them delete nodes, and some of the rules contain comparatively large graphs (up to 15 items). Also, the present programs for encoding and decoding graphs are much simpler than those in [12], in terms of the number and size of rules, and number of control constructs. To be fair, however, the present paper exploits the greater expressiveness of attributed rules compared with the standard graph transformation rules of [12]. For example, rules such as $\texttt{as}_{\mathrm{R,S}}$ or $\texttt{cnc}_{\mathrm{R,S,T}}$ (Subsection 4.3) copy lists of unbounded length in a single step, and the rules `regtoedge` and `regtoloop` (Section 5) use repeated integer variables in their left-hand sides to check the equality of addresses.

The most significant difference between the present work and [12] is the focus on translating an imperative programming language. HIRAM can be seen as a simple C-like language when register addresses are considered as variables. It has realistic data types (integers, lists and pointers) and high-level control constructs (if-then-else branching and while-loop). Viewed from this angle, our results prove the correctness of a *source-to-source* compiler which bridges the gap between an imperative language based on assignments and a rule-based graph programming language.

7. Conclusion and Future Work

Simple GP 2 programs can directly compute all computable graph functions, by transforming input graphs into output graphs. Simple programs contain only unconditional rules, abandon non-deterministic rule selection, and use branching commands which only test for the occurrence of graph patterns. Basic GP 2 programs are even more restricted in that rules consist of edge-less graphs and do not delete nodes. These rules resemble rewrite rules operating on sets of lists. Basic programs are sufficient to simulate arbitrary HIRAM programs but cannot directly compute all computable graph functions.

An aspect of the translation of HIRAM to GP 2 not discussed yet is that the number of rule applications by program P_M is linear in the number of operations performed by the source program M. Future work should focus on this linear relationship. It appears that the simple graphs occurring in rules can be matched in linear time within host graphs, and that any application conditions (such as $\texttt{m} = \texttt{n}$ in the rule $\texttt{eq}_{\mathrm{R,S}}$) are not more expensive to check than corresponding HIRAM operations. Hence P_M appears to simulate M with a quadratic time overhead at most.

Given that GP 2 is a non-deterministic language, it makes sense to consider the translation of a non-deterministic version of HIRAM to GP 2. There exist a few non-deterministic RAM models in the literature which could be adapted

to HIRAM. For example, one could introduce a `guess` command which assumes that register 0 holds a non-negative integer n and replaces n with some integer x such that $0 \leq x \leq n$ [10]. Alternatively, and more suitable for a translation to GP 2, one could add a command `choice`$(c_1, \ldots, c_n)$ which non-deterministically selects one of the commands $c_1, \ldots, c_n$ and executes it [2]. It is straightforward to translate such a statement by using GP 2's `or` command.

Finally, an ambitious project would be to extend HIRAM to a full-blown programming language, say a large subset of C, and extend the translation and its correctness proof accordingly. This would allow an automatic and correct translation of conventional graph algorithms to GP 2. For example, suppose that the C programs in [22] are covered by an extended HIRAM language. Then the translation would provide a GP 2 library of graph algorithms which could be used in future program developments.

Appendix A. Labels and Conditions in GP 2

Figure A.12 and Figure A.13 give grammars in Extended Backus-Naur Form defining the abstract syntax of labels and conditions of GP 2 rules. These grammars are ambiguous; in examples we use parentheses to disambiguate expressions if necessary.

Label	::=	List [Mark]
List	::=	`empty` | Atom | List ':' List | Var
Atom	::=	Integer | String | Var
Integer	::=	['–'] Digit {Digit} | '('Integer')'
		| Integer ('+' | '-' | '∗' | '/') Integer
		| (`indeg` | `outdeg`) '('NodeId')'
		| `length` '('Var')' | Var
String	::=	'"'{Character}'"' | String '.' String | Var
Mark	::=	`red` | `green` | `blue` | `grey` | `dashed` | `any`

Figure A.12: Abstract syntax of labels

The binary arithmetic operators '`+`', '`-`', '∗' and '`/`' expect integer expressions as arguments while '`.`' is string concatenation. Variables in category Var are typed by their declarations and can be used as expressions of supertypes. For example, integer variables can be used as lists of length one. The operators `indeg` and `outdeg` denote the number of ingoing respectively outgoing edges of a node, their arguments must be node identifiers of the left graph of the rule declaration in which the operators are used. The `length` operator returns the length of a list or string represented by a variable.

Marks are represented graphically in rule declarations, where `grey` is reserved for nodes and `dashed` is reserved for edges. Mark `any` can only be used in rule schemata and matches arbitrary marks in host graphs.

Conditions are Boolean combinations of subtype assertions, applications of the `edge` predicate to left-hand node identifiers, or relational comparisons of expressions (where `=` and `!=` can be used for arbitrary expressions).

```
Condition  ::=  (int | char | string | atom) '('Var')'
             | List ('=' | '!=') List
             | Integer ('>' | '>=' | '<' | '<=') Integer
             | edge '(' NodeId ',' NodeId [',' Label] ')'
             | not Condition
             | Condition (and | or) Condition
             | '(' Condition ')'
```

Figure A.13: Abstract syntax of conditions

Appendix B. Operational Semantics of GP 2

This appendix reviews the semantics of GP 2 (except for the definition of rule applications) in the style of structural operational semantics [17]. In this approach, inference rules inductively define a small-step transition relation $\rightarrow$ on *configurations*. In the setting of GP 2, a configuration is either a command sequence together with a host graph, just a host graph or the special element fail:

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

Configurations in $\mathrm{ComSeq} \times \mathcal{G}$, given by a rest program and a host graph, represent states of unfinished computations while graphs in $\mathcal{G}$ are final states or *results* of computations. The element fail represents a failure state. A configuration γ is said to be *terminal* if there is no configuration δ such that $\gamma \rightarrow \delta$.

Figure B.14 shows the inference rules for the core commands of GP 2. The rules contain meta-variables for command sequences and graphs, where R stands for a call in category RuleSetCall (as defined by the grammar in Figure 3), C, P, P', Q stand for command sequences in category ComSeq, and G, H stand for graphs in $\mathcal{G}$. The transitive and reflexive-transitive closures of $\rightarrow$ are written $\rightarrow^+$ and $\rightarrow^*$, respectively. We write $G \Rightarrow_R H$ if H results from host graph G by applying the rule set R, while $G \not\Rightarrow_R$ means that there is no graph H such that $G \Rightarrow_R H$ (application of R fails).

The inference rules for the remaining GP 2 commands are given in Figure B.15. These commands are referred to as *derived* commands because they can be defined by the core commands.

The meaning of GP 2 programs is summarised by the semantic function $[\![\text{-}]\!]$ which assigns to each command sequence P the function $[\![P]\!]$ mapping an input graph G to the set $[\![P]\!]G$ of all possible results of executing P on G. The value fail indicates a failed program run while $\perp$ indicates a run that does not terminate or gets stuck. Program P *can diverge from* G if there is an infinite

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

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

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

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

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

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

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

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

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

Figure B.14: Inference rules for core commands

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

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

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

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

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

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

Figure B.15: Inference rules for derived commands

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

The *semantic function* $[\![\_]\!]\colon \mathrm{ComSeq} \to (\mathcal{G} \to 2^{\mathcal{G}^{\oplus}})$ is defined by

$$[\![P]\!]G = \{X \in (\mathcal{G} \cup \{\mathrm{fail}\}) \mid \langle P, G\rangle \xrightarrow{+} X\} \cup \{\bot \mid P \text{ can diverge or get stuck from } G\}.$$

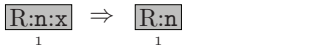
Getting stuck indicates a form of divergence that can happen with a command `if` C `then` P `else` Q or `try` C `then` P `else` Q in case C can diverge from a graph G and neither produce a graph nor fail from G, or with a loop $B!$ whose body B possesses the said property.

Appendix C. Translation of Assignments with Repeated Addresses

This appendix shows the translation of assignments containing repeated addresses.

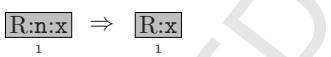
- $\tau[\mathrm{R} := \mathrm{R}] = \texttt{skip}$
- $\tau[\mathrm{R} := \texttt{head}\,\mathrm{R}] = \texttt{hd}_{\mathrm{R,R}}$ with rule declaration

  $\texttt{hd}_{\mathrm{R,R}}(\texttt{x: list; n: int})$

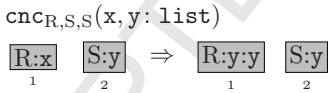
- $\tau[\mathrm{R} := \texttt{tail}\,\mathrm{R}] = \texttt{tl}_{\mathrm{R,R}}$ with rule declaration

  $\texttt{tl}_{\mathrm{R,R}}(\texttt{x: list; n: int})$

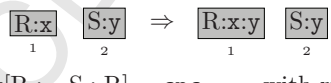
- $\tau[\mathrm{R} := \mathrm{S} : \mathrm{S}] = \texttt{cnc}_{\mathrm{R,S,S}}$ with rule declaration

  $\texttt{cnc}_{\mathrm{R,S,S}}(\texttt{x, y: list})$

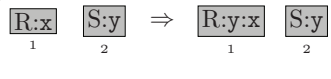
- $\tau[\mathrm{R} := \mathrm{R} : \mathrm{S}] = \texttt{cnc}_{\mathrm{R,R,S}}$ with rule declaration

  $\texttt{cnc}_{\mathrm{R,R,S}}(\texttt{x, y: list})$

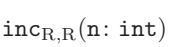
- $\tau[\mathrm{R} := \mathrm{S} : \mathrm{R}] = \texttt{cnc}_{\mathrm{R,S,R}}$ with rule declaration

  $\texttt{cnc}_{\mathrm{R,S,R}}(\texttt{x, y: list})$

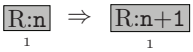
- $\tau[\mathrm{R} := \mathrm{R} : \mathrm{R}] = \mathtt{cnc}_{\mathrm{R,R,R}}$ with rule declaration

  $\mathtt{cnc}_{\mathrm{R,R,R}}(\mathtt{x}\colon \mathtt{list})$

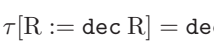
- $\tau[\mathrm{R} := \mathtt{inc}\ \mathrm{R}] = \mathtt{inc}_{\mathrm{R,R}}$ with rule declaration

  $\mathtt{inc}_{\mathrm{R,R}}(\mathtt{n}\colon \mathtt{int})$

- $\tau[\mathrm{R} := \mathtt{dec}\ \mathrm{R}] = \mathtt{dec}_{\mathrm{R,R}}$ with rule declaration

  $\mathtt{dec}_{\mathrm{R,R}}(\mathtt{n}\colon \mathtt{int})$

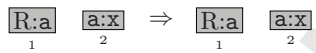
- $\tau[*\mathrm{R} := \mathrm{R}] = \mathtt{try}\ \mathtt{adr}_{\mathrm{R}}\ \mathtt{then}\ \mathtt{As}_{*\mathrm{R,R}}\ \mathtt{else}\ \mathtt{fail}$[4]

  with procedure declaration

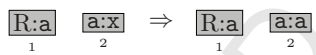
  $\mathtt{As}_{*\mathrm{R,R}} = \mathtt{try}\ \mathtt{t1}_{*\mathrm{R,R}}\ \mathtt{then}\ \mathtt{as1}_{*\mathrm{R,R}}\ \mathtt{else}\ \mathtt{try}\ \mathtt{t2}_{*\mathrm{R,R}}\ \mathtt{else}\ \mathtt{gen}_{*\mathrm{R,R}}$

  where the rules $\mathtt{t1}_{*\mathrm{R,R}}$, $\mathtt{as1}_{*\mathrm{R,R}}$, $\mathtt{t2}_{*\mathrm{R,R}}$ and $\mathtt{gen}_{*\mathrm{R,R}}$ are declared as follows:

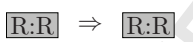
  $\mathtt{t1}_{*\mathrm{R,R}}(\mathtt{a}\colon \mathtt{int};\ \mathtt{x}\colon \mathtt{list})$

  $\mathtt{as1}_{*\mathrm{R,R}}(\mathtt{a}\colon \mathtt{int};\ \mathtt{x}\colon \mathtt{list})$

  $\mathtt{t2}_{*\mathrm{R,R}}$

  $\mathtt{gen}_{*\mathrm{R,R}}(\mathtt{a}\colon \mathtt{int})$

Acknowledgements. I am grateful to the anonymous referees of both the 26th Nordic Workshop on Programming Theory and this JLAMP issue. Their comments helped to improve previous versions of this paper.

[4] See Subsection 4.3 for the declaration of rule $\mathtt{adr}_{\mathrm{R}}$. The code for $\mathtt{As}_{*\mathrm{R,R}}$ is equivalent to $\mathtt{try}\ \mathtt{as1}_{*\mathrm{R,R}}\ \mathtt{else}\ \mathtt{try}\ \mathtt{t2}_{*\mathrm{R,R}}\ \mathtt{else}\ \mathtt{gen}_{*\mathrm{R,R}}$, which however is not simple GP 2 code.

References

[1] Aditya Agrawal, Gabor Karsai, Sandeep Neema, Feng Shi, and Attila Vizhanyo. The design of a language for model transformations. *Software and System Modeling*, 5(3):261–288, 2006.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[3] Christopher Bak, Glyn Faulkner, Detlef Plump, and Colin Runciman. A reference interpreter for the graph programming language GP 2. In *Proc. Graphs as Models (GaM 2015)*, volume 181 of *Electronic Proceedings in Theoretical Computer Science*, pages 48–64, 2015.

[4] Christopher Bak and Detlef Plump. Compiling graph programs to C. In *Proc. International Conference on Graph Transformation (ICGT 2016)*, volume 9761 of *Lecture Notes in Computer Science*, pages 102–117. Springer, 2016.

[5] Corado Böhm and Giuseppe Jacopini. Flow diagrams, Turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, 1966.

[6] Stephen A. Cook and Robert A. Reckhow. Time bounded random access machines. *Journal of Computer and System Sciences*, 7(4):354–375, 1973.

[7] Thomas H. Cormen, Charles E. Leiserson, Robert L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[8] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. Springer, 2006.

[9] Maribel Fernández, Hélène Kirchner, Ian Mackie, and Bruno Pinaud. Visual modelling of complex systems: Towards an abstract machine for PORGY. In *Proc. Computability in Europe (CiE 2014)*, volume 8493 of *Lecture Notes in Computer Science*, pages 183–193. Springer, 2014.

[10] Jörg Flum and Martin Grohe. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. Springer, 2006.

[11] Amir Hossein Ghamarian, Maarten de Mol, Arend Rensink, Eduardo Zambon, and Maria Zimakova. Modelling and analysis using GROOVE. *International Journal on Software Tools for Technology Transfer*, 14(1):15–40, 2012.

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

[13] Ivaylo Hristakiev and Detlef Plump. Attributed graph transformation via rule schemata: Church-Rosser theorem. In *Software Technologies: Applications and Foundations – STAF 2016 Collocated Workshops, Revised Selected Papers*, volume 9946 of *Lecture Notes in Computer Science*, pages 145–160. Springer, 2016.

[14] Edgar Jakumeit, Sebastian Buchwald, and Moritz Kroll. GrGen.NET - the expressive, convenient and fast graph rewrite system. *International Journal on Software Tools for Technology Transfer*, 12(3–4):263–271, 2010.

[15] Hans-Jörg Kreowski. Translations into the graph grammar machine. In Ronan Sleep, Rinus Plasmeijer, and Marko van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, chapter 13, pages 171–183. John Wiley, 1993.

[16] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

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

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

[19] Detlef Plump. The design of GP 2. In *Proc. Workshop on Reduction Strategies in Rewriting and Programming (WRS 2011)*, volume 82 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–16, 2012.

[20] Christopher M. Poskitt and Detlef Plump. Hoare-style verification of graph programs. *Fundamenta Informaticae*, 118(1-2):135–175, 2012.

[21] Olga Runge, Claudia Ermel, and Gabriele Taentzer. AGG 2.0 — new features for specifying and analyzing algebraic graph transformations. In *Proc. Applications of Graph Transformations with Industrial Relevance (AGTIVE 2011)*, volume 7233 of *Lecture Notes in Computer Science*, pages 81–88. Springer, 2012.

[22] Robert Sedgewick. *Algorithms in C. Part 5: Graph Algorithms*. Addison-Wesley, third edition, 2002.

[23] Tadahiro Uesu. A system of graph grammars which generates all recursively enumerable sets of labelled graphs. *Tsukuba Journal of Mathematics*, 2:11–26, 1978.

[24] Klaus Weihrauch. *Computability*. Springer, 1987.