From Imperative to Rule-based Graph Programs (Extended Abstract)

Detlef Plump

The University of York, UK

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. Domain-specific languages based on graph transformation rules include GROOVE [4], GRGEN.NET [5] and PORGY [3]. This paper focusses on the graph programming language GP [6, 7] which aims to support formal reasoning on programs (see [8] for a Hoare-logic approach to verifying GP programs).

We discuss the translation of (the core of) a simple imperative programming language to GP. The motivation for this is threefold:

1. To prove that GP is computationally complete, in the strong sense that graph functions are computable if and only if they can be directly computed with GP programs.
2. To identify a complete sublanguage of GP, by restricting as much as possible the form of rules and control constructs in the target language.
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 *high-level random access machines* (HRAMs) as a prototypical imperative language. They differ from standard RAMs in that they operate on registers holding lists of integers and use while loops and if-then-else commands. HRAM commands are translated into equivalent GP commands working on graphs that consist of register-like nodes. The number of rule applications of a translated program is linear in the number of operations executed by the source HRAM.

2 Graph Programs

We discuss a simple GP example program, see [7] for a language definition (the abstract syntax is given in the Appendix). The principal programming construct in GP are conditional graph transformation rules labelled with expressions. The program in Figure 1 checks whether a graph G is cyclic and, depending on the success or failure of the macro `cyclic`, executes either program P or program Q. The test is executed on a copy of G whereas P or Q is executed on the input G. Cycles are detected by deleting as long as possible edges whose sources have no incoming edges, and testing whether any edges remain. This is correct by the condition of `delete`: node 1 must have no incoming edges and hence a step $G \Rightarrow_{\texttt{delete}} H$ preserves both cycles and the absence of cycles. Moreover, graphs irreducible by `delete` are cyclic if and only if they contain edges.

In general, programs are executed on graphs labelled with lists whose entries are integers or character strings. Let $\mathcal{G}$ be the set of all host graphs and $\mathcal{G}^{\oplus} = \mathcal{G} \cup \{\bot, \text{fail}\}$. The semantics of a graph program P is a function $[\![P]\!] \colon \mathcal{G} \to 2^{\mathcal{G}^{\oplus}}$ which maps an input graph G to the set $[\![P]\!]G$ of all possible results (see [7] for a formal operational semantics).

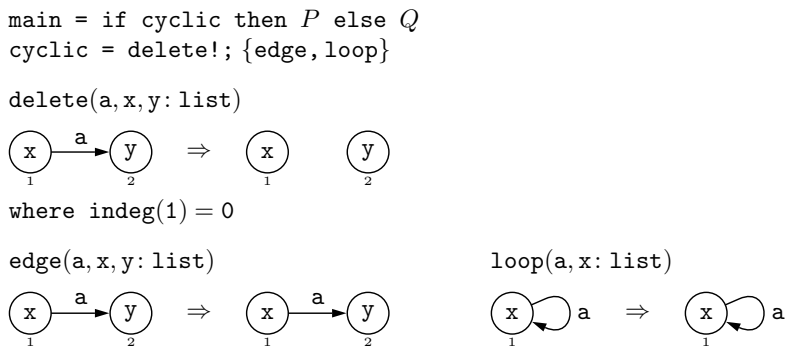

Figure 1: GP program for recognising cyclic graphs

3 From HRAMs to GP

Random access machines (RAMs) are a computational model characterised by an infinite sequence of registers which can be randomly accessed [1]. Typically, each register holds an integer. In contrast, the HRAMs of this paper consist of registers containing lists of integers.

Similar to the RAMs of [2], HRAMs can copy the contents of any register to any other register without a detour via an accumulator. The conditional jump in RAMs has been replaced by an if-then-else statement and a while loop. Other RAM models enhanced with high-level programming constructs include RAM-ALGOL [2] and Pidgin ALGOL [1].

Appendix B lists the HRAM commands and their meaning. Formally, the effect of a HRAM M is described by a function $[\![M]\!]\colon \mathcal{S} \to \mathcal{S}^{\oplus}$ which maps an initial state either to a final state, to $\bot$ in case of divergence, or to the special element fail in case of failure.

We translate HRAMs into GP programs operating on graphs consisting of register-like nodes. A *register graph* consists of nodes labelled $a{:}l$, where a is an address (a non-negative integer) and l an integer list, such that all addresses are distinct. 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 HRAM states. For example, Figure 2 shows a register graph on the left and the corresponding HRAM state in the middle (where registers not shown contain the empty list λ).

Arbitrary HRAMs M are translated into corresponding graph programs P_M. Lack of space prevents to present the translation; we just demonstrate it on one command. A loop `while` B `do` M is translated into

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

where $\tau[\![B]\!]$ is a rule checking the condition B and $\tau[\![M]\!]$ is the translation of the while loop's body. The second try command is needed in case $\tau[\![M]\!]$ fails, for then the !-loop terminates with the graph on which the loop's body was entered for the last time.

Rule applications in P_M are deterministic in that they either produce a unique graph or fail. It follows that the semantic function $[\![\mathrm{P}_M]\!]$ can be considered as a function $\mathcal{G}_{\text{reg}} \to \mathcal{G}_{\text{reg}}^{\oplus}$.[1]

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

The target program P_M belongs to the subclass of *simple* graph programs which are defined by the following restrictions: (1) graphs are only changed by unconditional rules; (2) non-deterministic rule selection (rule sets) and if-then-else statements are abandoned; (3) branching

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

commands have the form `try` r `then` P `else` Q, where r is the call of a rule with identical left and right graphs and possibly a condition `x` = `y` or `m` > `n`.

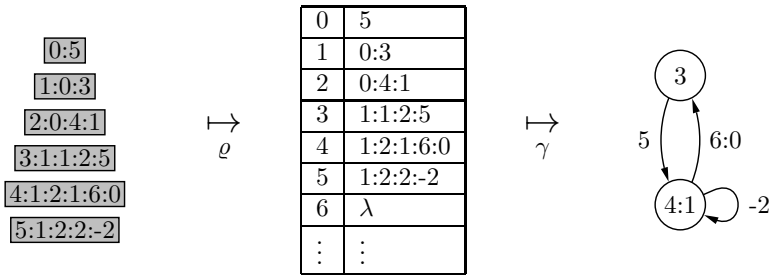

Figure 2: A register graph, the corresponding state, and the corresponding graph

The following corollary of Theorem 1 holds by the restricted form of the rules in P_M and the fact that termination and equivalence are undecidable for RAMs.

Corollary. *Termination and equivalence are undecidable for the class of simple graph programs with rules that are non-deleting and contain edge-less graphs of at most 3 nodes.*

To define computable graph functions, we need to represent graphs as HRAM states. We use register 0 to store graph size, followed by registers for nodes and edges (distinguished by a 0/1 flag). Edge registers contain, besides the label, the addresses of the edge's source and target. The set of all graph states is denoted by $\mathcal{S}_{\mathrm{gra}}$ and $\gamma\colon \mathcal{S}_{\mathrm{gra}} \to \mathcal{G}$ maps graph states to corresponding graphs. Figure 2 shows a graph state in the middle and the corresponding graph on the right.

A function $f\colon \mathcal{G} \to \mathcal{G}^{\oplus}$ is *computable* if there exists a HRAM M such that for all $s \in \mathcal{S}_{\mathrm{gra}}$, $[\![M]\!]s \in \mathcal{S}^{\oplus}_{\mathrm{gra}}$ and $\gamma^{\oplus}([\![M]\!]s) = f(\gamma(s))$.

Theorem 2. *For every computable function $f\colon \mathcal{G} \to \mathcal{G}^{\oplus}$ there exists a simple graph program P such that $[\![P]\!] = f$.*

Note that this is a strong form of completeness: program P computes f directly, not just a corresponding function on register graphs; P has the form `enc`; P_M; `dec`, where `enc` encodes graphs as register graphs representing graph states and `dec` decodes register graphs.

4 Conclusion

Simple GP programs can compute all computable graph functions. They modify graphs only by unconditional rules, abandon non-deterministic rule selection, and use a branching command which checks the occurrence of a graph pattern.

An aspect of the HRAM translation not discussed in the main text is that the number of rule applications of program P_M is linear in the number of operations performed by M. Also, the number of rule applications for graph encoding and decoding equals graph size.

A practical application of the completeness result would be the automatic translation of conventional graph algorithms to GP. For example, suppose that the C programs in [9] are rewritten as HRAM programs. Then the translation constructed in the proof of Theorem 2 would provide a library of GP graph algorithms which can be used as macros.

References

[1] A. V. Aho, J. E. Hopcroft, J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] S. A. Cook and R. A. Reckhow. Time bounded random access machines. *Journal of Computer and System Sciences*, 7(4):354-375, 1973.

[3] M. Fernández, H. Kirchner, I. Mackie, B. Pinaud. Visual modelling of complex systems: Towards an abstract machine for PORGY. In *Proc. CiE 2014*, LNCS 8493, pp 183-193, 2014.

[4] A.H. Ghamarian, M. de Mol, A. Rensink, E. Zambon, M. Zimakova. Modelling and analysis using GROOVE. *Int. J. on Softw. Tools for Techn. Transfer*, 14(1):15-40, 2012.

[5] E. Jakumeit, S. Buchwald, M. Kroll. GrGen.NET - the expressive, convenient and fast graph rewrite system. *Int. J. on Softw. Tools for Techn. Transfer*, 12(3-4):263-271, 2010.

[6] D. Plump. The graph programming language GP. In *Proc. CAI 2009*, LNCS 5725, pp 99-122, 2009.

[7] D. Plump. The design of GP 2. In *Proc. WRS 2011*, EPTCS 82, pp 1-16, 2012.

[8] C. M. Poskitt and D. Plump. Hoare-style verification of graph programs. *Fundamenta Informaticae*, 118(1-2):135-175, 2012.

[9] R. Sedgewick. *Algorithms in C. Part 5: Graph Algorithms*. Addison-Wesley, 2002.

Appendix

A GP Syntax

The grammar in Figure 3 gives the abstract syntax of GP 2 programs. A program consists of declarations of conditional rules and macros, and exactly one main command sequence. The category RuleId refers to declarations of conditional rules in RuleDecl whose syntax is omitted.

Prog	::=	Decl {Decl}
Decl	::=	RuleDecl | MacroDecl | MainDecl
MacroDecl	::=	MacroId '=' ComSeq
MainDecl	::=	`main` '=' ComSeq
ComSeq	::=	Com {';' Com}
Com	::=	RuleSetCall | MacroCall
		| `if` ComSeq `then` ComSeq [`else` ComSeq]
		| `try` ComSeq [`then` ComSeq] [`else` ComSeq]
		| ComSeq '!' | `skip` | `fail`
RuleSetCall	::=	RuleId | '{' [RuleId {',' RuleId}] '}'
MacroCall	::=	MacroId

Figure 3: Abstract syntax of GP programs

The call of a rule set $\{r_1, \ldots, r_n\}$ nondeterministically applies any of the rules to the current graph. The call fails if none of the left-hand graphs of the rules matches a subgraph. (Graph matching is injective and fails if applying the rule would create dangling edges.) The command `if` C `then` P `else` Q is executed on a 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; 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 $P!$ executes the body P repeatedly until P fails. When this is the case, $P!$ terminates with the graph with which the body was entered for the last time. The commands `skip` and `fail` have the obvious meaning. They are equivalent to the rule $\emptyset \Rightarrow \emptyset$ and the rule set $\{\}$, respectively.

Labels have type `int`, `string`, `atom` or `list`. Type `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 label is considered as a list. Given lists `x` and `y`, `x:y` denotes the concatenation of `x` and `y`.

B High-level RAMs

The syntax of HRAMs is given in Figure 4, together with an explanation of the commands' effect on a state s. Most operations should be self-explanatory. Note that integers are not distinguished from lists of length one and that the colon operator is list concatenation. When writing concrete lists in examples, the colon is also used to separate the entries of a list.

Commands			Comments
Int	::=	...	Integer numerals
R, S, T	::=	...	Nonnegative integer numerals (addresses)
List	::=	`empty`	Empty list
		| Int	Integers are lists of length 1
		| List : List	Concatenation
B	::=	R = S	True if $s(\mathrm{R}) = s(\mathrm{S})$
		| R > S	True if $s(\mathrm{R}), s(\mathrm{S}) \in \mathbb{Z}$ and $s(\mathrm{R}) > s(\mathrm{S})$
M	::=	M; M	Sequential composition
		| `if` B `then` M `else` M	Branching
		| `while` B `do` M	Loop
		| R := 'List'	Assign constant list to R
		| R := S	Copy $s(\mathrm{S})$ to R
		| R := `head` S	Copy head of $s(\mathrm{S})$ to R (fails if $s(\mathrm{S})$ is empty)
		| R := `tail` S	Copy tail of $s(\mathrm{S})$ to R (fails if $s(\mathrm{S})$ is empty)
		| R := S : T	Assign $s(\mathrm{S}) : s(\mathrm{T})$ to R
		| R := ∗S	Copy $s(s(\mathrm{S}))$ to R (fails if $s(\mathrm{S}) \notin \mathbb{N}_0$)
		| ∗R := S	Copy $s(\mathrm{S})$ to $s(\mathrm{R})$ (fails if $s(\mathrm{R}) \notin \mathbb{N}_0$)
		| R := S + 1	Assign $s(\mathrm{S}) + 1$ to R (fails if $s(\mathrm{S}) \notin \mathbb{Z}$)
		| R := S − 1	Assign $s(\mathrm{S}) - 1$ to R (fails if $s(\mathrm{S}) \notin \mathbb{Z}$)

Figure 4: High-level random access machines (HRAMs)

Registers are represented by nonnegative integers called addresses. In Figure 4, the letters R, S and T stand for fixed addresses. The contents of a register r is a list of integers. Formally, a state is a mapping $s\colon \mathbb{N}_0 \to \mathbb{Z}^*$ with $s(n) = \lambda$ almost everywhere. Here λ is the empty list which in programs is represented by the keyword `empty`.

An assignment R := S copies the list contained in register S to register R. In contrast, R := 'L' assigns the constant list L to R (the list is written in quotes to avoid confusion with addresses). The assignments R := ∗S and ∗R := S use indirect addressing (pointers), that is, they interpret the contents of starred registers as addresses.

A HRAM computation starts from a state s such that $s(n) = \lambda$ for all $n > s(0)$, where registers $1, 2, \ldots, s(0)$ contain the input. When the program terminates in a state s' (rather than fails or diverges), registers $1, 2, \ldots, s'(0)$ contain the output.

HRAM failure is caused by type errors. For example, R := S + 1 fails if $s(\mathrm{S})$ is not an integer. Similarly, ∗R := S fails if $s(\mathrm{R})$ is not an address. A computation also fails if it finishes in a state s such that $s(0)$ is not an address.