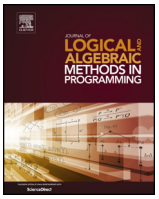
Journal of Logical and Algebraic Methods in Programming 88 (2017) 154–173

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp



From imperative to rule-based graph programs

Detlef Plump

University of York, United Kingdom

ARTICLE INFO

Article history:
Received 23 January 2015
Received in revised form 10 May 2016
Accepted 5 December 2016
Available online 13 January 2017

Keywords:
Graph programs
GP 2
Rule-based programming
Computational completeness
Random access machines
Graph transformation

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.

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 focuses 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.

E-mail address: detlef.plump@york.ac.uk.

http://dx.doi.org/10.1016/j.jlamp.2016.12.001
2352-2208/

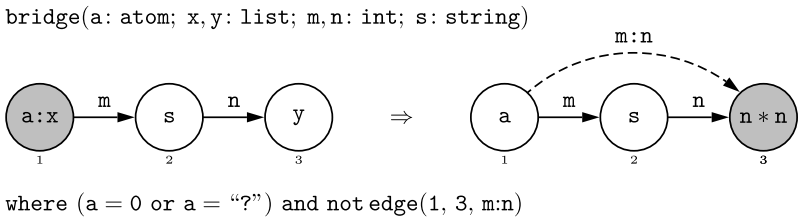


Fig. 1. Declaration of a conditional rule.

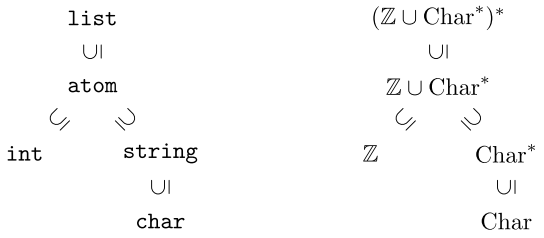


Fig. 2. Subtype hierarchy.

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, Fig. 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 Fig. A.12. Labels consist of an expression and an optional mark (explained below). Expressions are of type `int`, `char`, `string`, `atom` or `list`, where `atom` is the union of `int` and `string`, and `list` is the type of a (possibly empty) list of atoms. Lists of length one are equated with their entries and hence every expression can be considered as a list. The subtype hierarchy of GP 2 is shown in Fig. 2 (both syntactically and semantically).

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 Fig. 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

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