Describing Gen/Kill Static Analysis Techniques with Kleene Algebra*

Therrezinha Fernandes and Jules Desharnais

Département d’informatique et de génie logiciel
Université Laval, Québec, QC, G1K 7P4 Canada
{Therrezinha.Fernandes, Jules.Desharnais}@ift.ulaval.ca

Abstract. Static program analysis consists of compile-time techniques for determining properties of programs without actually running them. Using Kleene algebra, we formalize four instances of a static data flow analysis technique known as gen/kill analysis. This formalization clearly reveals the dualities between the four instances; although these dualities are known, the standard formalization does not reveal them in such a clear and concise manner. We provide two equivalent sets of equations characterizing the four analyses for two representations of programs, one in which the statements label the nodes of a control flow graph and one in which the statements label the transitions.

1 Introduction

Static program analysis consists of compile-time techniques for determining properties of programs without actually running them. Information gathered by these techniques is traditionally used by compilers for optimizing the object code [1] and by CASE tools for software engineering and reengineering [2, 3]. Among the more recent applications is the detection of malicious code or code that might be maliciously exploited [4, 5]. Due to ongoing research in this area [5], the latter application is the main motivation for developing the algebraic approach to static analysis described in this paper (but we will not discuss applications to security here). Our goal is the development of an algebraic framework based on Kleene algebra (KA) [6–11], in which the relevant properties can be expressed in a compact and readable way.

In this paper, we examine four instances of a static data flow analysis technique known as gen/kill analysis [1, 12, 13]. The standard description of the four instances is given in Sect. 2. The necessary concepts of Kleene algebra are then presented in Sect. 3. The four gen/kill analyses are formalized with KA in Sect. 4. This formalization clearly reveals the dualities between the four kinds of analysis; although these dualities are known, the standard formalization does not reveal them in such a clear and concise manner. We provide two equivalent sets of equations characterizing the four analyses for two representations of programs, one in which the statements label the nodes of a control flow graph and one in

* This research is supported by NSERC (Natural Sciences and Engineering Research Council of Canada).
which the statements label the transitions. In the conclusion, we make additional comments on the approach and on directions for future research.

2 Four Different Gen/Kill Analyses

The programming language we will use is the standard while language, with atomic statements skip and \( x := E \) (assignment), and compound statements \( S_1; S_2 \) (sequence), if \( b \) then \( S_1 \) else \( S_2 \) (conditional) and while \( b \) do \( S \) (while loop). In data flow analysis, it is common to use an abstract graph representation of a program from which one can extract useful information. Traditionally [1, 12, 13], this representation is a control flow graph (CFG), which is a directed graph where each node corresponds to a statement and the edges describe how control might flow from one statement to another. Labeled Transition Systems (LTSs) can also be used. With LTSs, edges (arcs, arrows) are labeled by the statements of the program and nodes are points from which and toward which control leaves and returns. Figure 1 shows CFGs and LTSs for the compound statements, and the corresponding matrix representations; the CFG for an atomic statement consists of a single node while its LTS consists of two nodes linked by an arrow labeled with the statement. The numbers at the left of the nodes for the CFGs and inside the nodes for the LTSs are labels that also correspond to the lines/columns in the matrix representations. Note that the two arrows leaving node 1 in the LTSs of the conditional and while loop are both labelled \( b \), i.e., the cases where \( b \) holds and does not hold are not distinguished. This distinction will not be needed here (and it is not present in the CFGs either). For both representations, the nodes of the graphs will usually be called program points, or points for short.

![Fig. 1. CFGs and LTSs for compound statements](image-url)