Available online at www.sciencedirect.com
ScienceDirect
Science of Computer Programming 65 (2007) 173–194
Science of Computer Programming
www.elsevier.com/locate/scico

Describing data flow analysis techniques with Kleene algebra☆

Therrezinha Fernandes, Jules Desharnais*

Département d'informatique et de génie logiciel, Université Laval, Québec, QC, G1K 7P4, Canada

Received 31 January 2005; received in revised form 15 January 2006; accepted 30 January 2006
Available online 28 November 2006

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 general class of static intraprocedural data flow analyses known as 'gen/kill' analyses. This formalization exhibits the dualities between the four analyses in a clear and concise manner. We provide two equivalent sets of equations characterizing the four analyses for two different 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. We formally describe how the data flow equations for the two representations are related. We also prove the soundness of the KA based approach with respect to the standard approach.

Keywords: Static intraprocedural data flow analysis; 'Gen/kill' analyses; Kleene algebra with tests; Matrices over a Kleene algebra; Data flow graphs; Labelled transition systems

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 [1–4] and by CASE tools for software engineering and reengineering [5,6]. Among the more recent applications is the detection of malicious code or code that might be maliciously exploited [7,8]. Due to recent research in this area [8], 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) [9–14], in which the relevant properties can be expressed in a compact and readable way.

In this paper, we present an approach based on KA for the static *data flow* analysis of programs. With this approach, it is possible to compute the precise 'meet-over-all-paths' (MOP) [15,16] solutions to a general class of intraprocedural data flow analysis problems. This class consists of all problems which are instances of the bit vector framework [4]. All the classical and well-known intraprocedural analyses are in this class: *reaching definitions* analysis, *live variables* analysis, *very busy expressions* analysis, *available expressions* analysis and *copy propagation* analysis [1–4]. This

☆ This research is supported by NSERC (Natural Sciences and Engineering Research Council of Canada).
* Corresponding author.
E-mail addresses: Therrezinha.Fernandes@ift.ulaval.ca (T. Fernandes), Jules.Desharnais@ift.ulaval.ca (J. Desharnais).

0167-6423/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2006.01.009

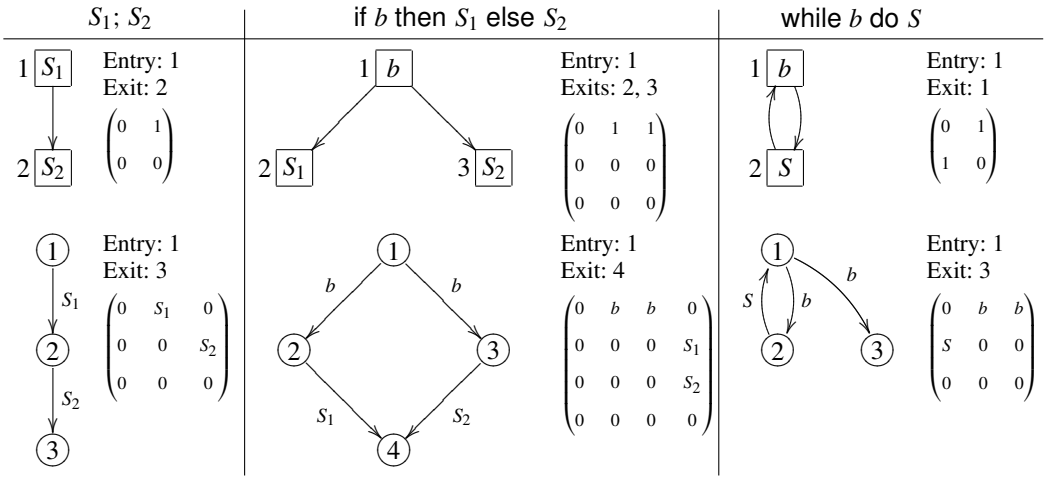


Fig. 1. CFGs and LTSs for compound statements.

class has a broad scope of application in program code optimization like, for instance, code motion and partial dead-code elimination.

We focus on four instances of the class of data flow analyses just described. The standard definition of the four instances is given in Section 2. The necessary concepts of KA are then presented in Section 3. The four gen/kill analyses are formalized with KA in Section 4. This formalization exhibits the dualities between the four analyses in 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. In Section 5, we formally describe how the data flow equations for the two representations are related. In Section 6, we prove the soundness of the KA based approach with respect to the standard approach. Then there is a short section on related work and, in the conclusion, we make additional comments on the approach and on directions for future research.

2. Four gen/kill analyses

The programming language we will use is the standard while language, with atomic statements[1] 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 static program analysis, and in particular in data flow analysis, it is common to use an abstract graph representation of a program from which one can extract useful information. Traditionally [1–4], the representation used by optimizing compilers and data flow analyzers 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), which are (more) typically used for model-checking [17], 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. Fig. 1 shows CFGs and LTSs for the compound statements of the while language, and the corresponding matrix representations; the CFG for an atomic statement consists of a single node corresponding to the statement while its LTS consists of two nodes linked by an arrow labelled 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 is not needed here (and 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.

[1] We also use the term *instruction* as a synonym for *statement*. In the sequel, the term *atomic statement* will include both atomic statements as defined in this paragraph and tests.

Table 1
Classification of the four analyses

	Forward	Backward
May	RD	LV
Must	AE	VBE

The four instances of the class of gen/kill analysis problems that we will consider are *Reaching Definitions Analysis* (RD), *Live Variables Analysis* (LV), *Available Expressions Analysis* (AE) and *Very Busy Expressions Analysis* (VBE). An informal description, extracted from [4], follows.

RD Reaching definitions analysis determines, for each program point, which assignments *may* have been made and not overwritten when program execution reaches this point along *some* path.

A main application of RD is in the construction of direct links between statements that produce values and statements that use them.

LV A variable is *live* at the exit from a program point if *there exists a path*, from that point to a use of the variable, that does not redefine the variable. Live variables analysis determines, for each program point, which variables are live at the exit from this point.

This analysis might be used as the basis for *dead code elimination*. If a variable is not live at the exit from a statement, then, if the statement is an assignment to the variable, the statement can be eliminated.

AE Available expressions analysis determines, for each program point, which expressions have already been computed, and not later modified, on *all paths* to the program point.

This information can be used to avoid the recomputation of an expression.

VBE An expression is *very busy* at the exit from a program point if, *no matter which path* is taken from that point, the expression is always evaluated before any of the variables occurring in it are redefined. Very busy expressions analysis determines, for each program point, which expressions are very busy at the exit from the point.

A possible optimization based on this information is to evaluate the expression and store its value for later use.

Each of the four analyses uses a universal dataset D whose type of elements depends on the analysis. This set D contains information about the program under consideration. Statements *generate* and *kill* elements from D. Statements are viewed either as producing a subset $\mathsf{out} \subseteq D$ from a subset $\mathsf{in} \subseteq D$, or as producing $\mathsf{in} \subseteq D$ from $\mathsf{out} \subseteq D$, depending on the direction of the analysis. Calculating in from out (or the converse) is the main goal of the analyses. Each analysis is either *forward* or *backward*, and is said to be either a *may* or *must* analysis. This is detailed in the following description and summarized in Table 1.

RD The set D is a set of *definitions*. A definition is a pair (x, l), where l is the label of an assignment $x := E$. From the above definition of RD, it can be seen that, for each program point, the analysis looks at paths between the entry point of the program and that program point; thus, the analysis is a forward one. Also, it is a may analysis, since the existence of a path with the desired property suffices.

LV The set D is a set of variables. The analysis looks for the existence of a path with a specific property between program points and the exit of the program. It is thus a backward may analysis.

AE The set D is a set of expressions or subexpressions. The paths considered are those between the entry point of the program and the program points (forward analysis). Since all paths to a program point must have the desired property, it is a must analysis.

VBE The set D is a set of expressions or subexpressions. The paths considered are those between the program points and the exit point of the program (backward analysis). Since all paths from a program point must have the desired property, it is a must analysis.

Table 2 gives the definitions of gen and kill for the atomic statements and tests for the four analyses. Note that for each statement S, $\mathsf{gen}(S) \subseteq \overline{\mathsf{kill}(S)}$ (the complement of $\mathsf{kill}(S)$) for the four analyses. This is a natural property, meaning that if something is generated, then it is not killed. In this table, $\mathsf{Var}(E)$ denotes the set of variables of expression E and $\mathsf{Exp}(E)$ denotes the set of its subexpressions. These definitions can be extended recursively to the case of compound statements (Table 3). The forward/backward duality is apparent when comparing the values of

Table 2
Gen/kill values for atomic statements and tests

	$l : x := E$		skip		b	
	gen	kill	gen	kill	gen	kill
RD	$\{(x, l)\}$	$\{(x, l') \in D \mid l' \neq l\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
LV	$\mathsf{Var}(E)$	$\{x\} - \mathsf{Var}(E)$	$\emptyset$	$\emptyset$	$\mathsf{Var}(b)$	$\emptyset$
AE	$\{E' \in \mathsf{Exp}(E) \mid x \notin \mathsf{Var}(E')\}$	$\{E' \in D \mid x \in \mathsf{Var}(E')\}$	$\emptyset$	$\emptyset$	$\mathsf{Exp}(b)$	$\emptyset$
VBE	$\mathsf{Exp}(E)$	$\{E' \in D \mid x \in \mathsf{Var}(E')\}$ $- \mathsf{Exp}(E)$	$\emptyset$	$\emptyset$	$\mathsf{Exp}(b)$	$\emptyset$

The symbol l denotes a label and the symbol b a test.

Table 3
Gen/kill expressions for compound statements

	$S_1; S_2$	
	gen	kill
RD, AE	$\mathsf{gen}(S_2) \cup (\mathsf{gen}(S_1) - \mathsf{kill}(S_2))$	$\mathsf{kill}(S_2) \cup (\mathsf{kill}(S_1) - \mathsf{gen}(S_2))$
LV, VBE	$\mathsf{gen}(S_1) \cup (\mathsf{gen}(S_2) - \mathsf{kill}(S_1))$	$\mathsf{kill}(S_1) \cup (\mathsf{kill}(S_2) - \mathsf{gen}(S_1))$

	if b then S_1 else S_2	
	gen	kill
RD	$\mathsf{gen}(S_1) \cup \mathsf{gen}(S_2)$	$\mathsf{kill}(S_1) \cap \mathsf{kill}(S_2)$
LV	$\mathsf{gen}(b) \cup \mathsf{gen}(S_1) \cup \mathsf{gen}(S_2)$	$(\mathsf{kill}(S_1) \cap \mathsf{kill}(S_2)) - \mathsf{gen}(b)$
AE	$(\mathsf{gen}(S_1) \cap \mathsf{gen}(S_2))$ $\cup (\mathsf{gen}(b) - (\mathsf{kill}(S_1) \cup \mathsf{kill}(S_2)))$	$\mathsf{kill}(S_1) \cup \mathsf{kill}(S_2)$
VBE	$\mathsf{gen}(b) \cup (\mathsf{gen}(S_1) \cap \mathsf{gen}(S_2))$	$(\mathsf{kill}(S_1) \cup \mathsf{kill}(S_2)) - \mathsf{gen}(b)$

	while b do S_1	
	gen	kill
RD, LV	$\mathsf{gen}(b) \cup \mathsf{gen}(S_1)$	$\emptyset$
AE, VBE	$\mathsf{gen}(b)$	$\mathsf{kill}(S_1) - \mathsf{gen}(b)$

Table 4
Linking in and out

	$\mathsf{in}(S)$	$\mathsf{out}(S)$
RD	$\bigcup(S' \mid S' \in \mathsf{pred}(S) : \mathsf{out}(S'))$	$\mathsf{gen}(S) \cup (\mathsf{in}(S) - \mathsf{kill}(S))$
LV	$\mathsf{gen}(S) \cup (\mathsf{out}(S) - \mathsf{kill}(S))$	$\bigcup(S' \mid S' \in \mathsf{succ}(S) : \mathsf{in}(S'))$
AE	$\bigcap(S' \mid S' \in \mathsf{pred}(S) : \mathsf{out}(S'))$	$\mathsf{gen}(S) \cup (\mathsf{in}(S) - \mathsf{kill}(S))$
VBE	$\mathsf{gen}(S) \cup (\mathsf{out}(S) - \mathsf{kill}(S))$	$\bigcap(S' \mid S' \in \mathsf{succ}(S) : \mathsf{in}(S'))$

$\mathsf{gen}(S_1; S_2)$ and $\mathsf{kill}(S_1; S_2)$ for RD and AE with those for LV and VBE. The may/must duality between RD, LV and AE, VBE is most visible for the conditional (uses of $\cup$ vs $\cap$ in the expressions for kill).

Finally, Table 4 shows how $\mathsf{out}(S)$ and $\mathsf{in}(S)$ can be recursively calculated. The expressions $\mathsf{pred}(S)$ and $\mathsf{succ}(S)$ denote the sets of immediate predecessors and immediate successors of node S. Here too, the dualities forward/backward and may/must are easily seen.

We now illustrate RD analysis with the program given in Fig. 2. We will use this program all through the paper. The numbers at the left of the program are labels. These labels are the same in the given CFG representation.

The set of definitions that appear in the program is $\{(x, 1), (x, 3), (y, 4)\}$. Using Table 2 for RD, we get the following gen/kill values for the atomic statements, where S_l denotes the atomic statement at label l:

$$\begin{aligned} &\mathsf{gen}(S_1) = \{(x, 1)\}, && \mathsf{kill}(S_1) = \{(x, 3)\}, \\ &\mathsf{gen}(S_2) = \emptyset, && \mathsf{kill}(S_2) = \emptyset, \\ &\mathsf{gen}(S_3) = \{(x, 3)\}, && \mathsf{kill}(S_3) = \{(x, 1)\}, \\ &\mathsf{gen}(S_4) = \{(y, 4)\}, && \mathsf{kill}(S_4) = \emptyset. \end{aligned} \tag{1}$$

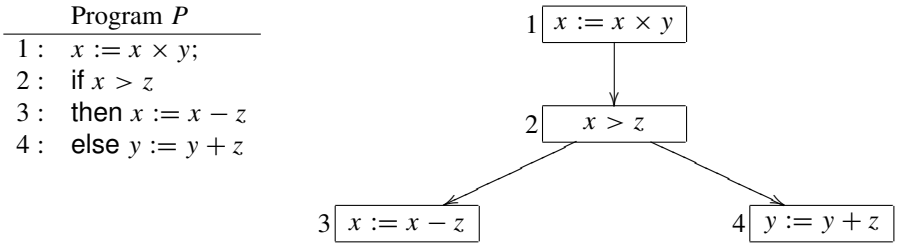


Fig. 2. CFG for the running example.

Using Table 3 for RD, we get

$$\begin{array}{lcl} \mathsf{gen}(S_1; \text{if } S_2 \text{ then } S_3 \text{ else } S_4) & = & \{(x,3),(y,4)\} \cup (\{(x,1)\} - \emptyset) \\ & = & \{(x,1),(x,3),(y,4)\}, \\ \mathsf{kill}(S_1; \text{if } S_2 \text{ then } S_3 \text{ else } S_4) & = & \emptyset \cup (\{(x,3)\} - \{(x,3),(y,4)\}) \quad = \quad \emptyset. \end{array}$$

Finally, Table 4 for RD yields

$$\begin{array}{ll} \mathsf{in}(S_1) = \emptyset, & \mathsf{out}(S_1) = \{(x,1)\} \cup (\emptyset - \{(x,3)\}) = \{(x,1)\} \\ \mathsf{in}(S_2) = \{(x,1)\}, & \mathsf{out}(S_2) = \emptyset \cup (\{(x,1)\} - \emptyset) = \{(x,1)\} \\ \mathsf{in}(S_3) = \{(x,1)\}, & \mathsf{out}(S_3) = \{(x,3)\} \cup (\{(x,1)\} - \{(x,1)\}) = \{(x,3)\} \\ \mathsf{in}(S_4) = \{(x,1)\}, & \mathsf{out}(S_4) = \{(y,4)\} \cup (\{(x,1)\} - \emptyset) = \{(x,1),(y,4)\} \end{array}$$

and

$$\begin{array}{lcl} \mathsf{in}(S_1; \text{if } S_2 \text{ then } S_3 \text{ else } S_4) & = & \emptyset, \\ \mathsf{out}(S_1; \text{if } S_2 \text{ then } S_3 \text{ else } S_4) & = & \{(x,1),(x,3),(y,4)\} \cup (\emptyset - \emptyset) \\ & = & \{(x,1),(x,3),(y,4)\}. \end{array}$$

3. Kleene algebra with tests

In this section, we first introduce Kleene algebra [9,12] and a specialization of it, namely Kleene algebra with tests [13]. Then, we recall the notion of matrices over a Kleene algebra and discuss how we will use them for our application.

Definition 1. A *Kleene algebra* (KA) [12] is a structure $\mathcal{K} = (K, +, \cdot, ^*, 0, 1)$ such that $(K, +, 0)$ is a commutative monoid, $(K, \cdot, 1)$ is a monoid, and the following laws hold:

$$\begin{array}{ll} a + a = a, & a \cdot (b + c) = a \cdot b + a \cdot c, \\ a \cdot 0 = 0 \cdot a = 0, & (a + b) \cdot c = a \cdot c + b \cdot c, \\ 1 + a \cdot a^* = a^*, & b + a \cdot c \leq c \;\Rightarrow\; a^* \cdot b \leq c, \\ 1 + a^* \cdot a = a^*, & b + c \cdot a \leq c \;\Rightarrow\; b \cdot a^* \leq c, \end{array}$$

where $\leq$ is the partial order induced by $+$, that is,

$$a \leq b \;\Leftrightarrow\; a + b = b.$$

A *Kleene algebra with tests* [13] is a two-sorted algebra $(K, T, +, \cdot, ^*, 0, 1, \neg)$ such that $(K, +, \cdot, ^*, 0, 1)$ is a Kleene algebra and $(T, +, \cdot, \neg, 0, 1)$ is a Boolean algebra, where $T \subseteq K$ and $\neg$ is a unary operator defined only on T.

Operator precedence, from lowest to highest, is $+$, $\cdot$, $(^*, \neg)$.

It is immediate from the definition that $t \leq 1$ for any test $t \in T$. The meet of two tests $t, u \in T$ is their product $t \cdot u$. Every KA can be made into a KA with tests, by taking $\{0, 1\}$ as the set of tests.

The following laws can be derived from the axioms and will be used in the sequel:

$$\begin{array}{l}\text{(a) } a\cdot a^* \le a^*, \quad a^*\cdot a \le a^*, \quad a^*\cdot a^* = a^*,\\ \text{(b) } (a+b)^* = a^*\cdot(b\cdot a^*)^* = (a^*\cdot b)^*\cdot a^*,\\ \text{(c) } a^* = 1 + a\cdot a^* = 1 + a^*\cdot a,\\ \text{(d) } (a\cdot b)^*\cdot a = a\cdot(b\cdot a)^*.\end{array} \tag{2}$$

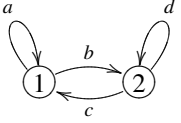
Models of KA with tests include algebras of languages over an alphabet, algebras of path sets in a directed graph [18], algebras of relations over a set and abstract relation algebras with transitive closure [19,20].

A very simple model of KA with tests is obtained by taking K and T to both be the powerset of some set D, and defining, for every $a, b \subseteq D$,

$$0 \stackrel{\text{def}}{=} \emptyset, \qquad 1 \stackrel{\text{def}}{=} D, \qquad a^* \stackrel{\text{def}}{=} D, \qquad \neg a \stackrel{\text{def}}{=} \overline{a}, \qquad a + b \stackrel{\text{def}}{=} a \cup b, \qquad a\cdot b \stackrel{\text{def}}{=} a \cap b. \tag{3}$$

The set of matrices of size $n \times n$ over a KA with tests can itself be turned into a KA with tests by defining the following operations. The notation $A[i, j]$ refers to the entry in row i and column j of matrix A.

- 0: matrix whose entries are all 0, i.e., $0[i, j] = 0$. The symbol 0 is thus overloaded and may denote an element of the base KA as well as matrices of different sizes.
- 1: identity matrix (square), i.e., $1[i, j] = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j. \end{cases}$
  Like 0, the symbol 1 is overloaded.
- $(A + B)[i, j] \stackrel{\text{def}}{=} A[i, j] + B[i, j]$.
- $(A\cdot B)[i, j] \stackrel{\text{def}}{=} \sum(k \mid : A[i, k]\cdot B[k, j])$.
- The Kleene star of a square matrix is defined recursively [12]. If $A = (a)$, for some $a \in K$, then $A^* \stackrel{\text{def}}{=} (a^*)$. If

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \qquad \Big(\text{with graphic representation } \raisebox{-0.5em}{  } \Big),$$

  for some $a, b, c, d \in K$, then

$$A^* \stackrel{\text{def}}{=} \begin{pmatrix} f^* & f^*\cdot b\cdot d^* \\ d^*\cdot c\cdot f^* & d^* + d^*\cdot c\cdot f^*\cdot b\cdot d^* \end{pmatrix}, \tag{4}$$

  where $f = a + b\cdot d^*\cdot c$; the automaton corresponding to A helps understand that f^* corresponds to paths from state 1 to state 1. If A is a larger matrix, it is decomposed as a 2×2 matrix of submatrices: $A = \begin{pmatrix} B & C \\ D & E \end{pmatrix}$, where B and E are square and nonempty. Then A^* is calculated recursively using (4). For our simple application below, $A^* = \sum(n \mid n \ge 0 : A^n)$, where $A^0 = 1$ and $A^{n+1} = A\cdot A^n$.

We take as tests all square matrices below 1. Hence, a square matrix T is a test if it is a diagonal matrix whose diagonal contains tests. For instance, if t_1, t_2 and t_3 are tests,

$$\begin{pmatrix} t_1 & 0 & 0 \\ 0 & t_2 & 0 \\ 0 & 0 & t_3 \end{pmatrix} \text{ is a test and } \neg\begin{pmatrix} t_1 & 0 & 0 \\ 0 & t_2 & 0 \\ 0 & 0 & t_3 \end{pmatrix} = \begin{pmatrix} \neg t_1 & 0 & 0 \\ 0 & \neg t_2 & 0 \\ 0 & 0 & \neg t_3 \end{pmatrix}.$$

In Section 4, we will only use matrices whose entries are all tests. Such matrices are relations [21]; indeed, a top relation can be defined as the matrix filled with 1. However, this is not completely convenient for our purpose. Rather, we will use a matrix S to represent the structure of programs and consider only matrices below the reflexive transitive closure S^* of S. Complementation can then be defined as complementation relative to S^*,

$$(\overline{A})[i, j] \stackrel{\text{def}}{=} \neg(A[i, j])\cdot S^*[i, j],$$

and the meet of two matrices by

$$(A \sqcap B)[i, j] \stackrel{\text{def}}{=} A[i, j]\cdot B[i, j].$$

It is easily checked that applying all the above operations to matrices below S^* results in a matrix below S^*. This means that the KA we will use is Boolean, with a top element $\top \stackrel{\text{def}}{=} S^*$ satisfying $1 \leq \top$ (reflexivity) and $\top \cdot \top \leq \top$ (transitivity).

By setting up an appropriate type discipline, one can define *heterogeneous Kleene algebras* as is done for heterogeneous relation algebras [22–24]. One can get a heterogeneous KA by considering matrices with different sizes over a KA; matrices can be joined or composed only if they satisfy the usual size constraints.

Finally, we note that for convenience, a matrix with a single entry will often be identified with that entry. That is, we take $(\, a \,) = a$.

4. Gen/kill analysis with Kleene algebra

In order to explain the KA-based approach, we present in Section 4.1 the equations describing RD analysis and apply them to the same example as in Section 2. The equations for the other analyses are presented in Section 4.2.

4.1. RD analysis

We first explain how the programs to analyze are modelled, using the example program from Fig. 2. Then we show how to carry out RD analysis, first by using a CFG-related matrix representation and then an LTS-related matrix representation.

Recall that the set of definitions for the whole program is

$$D \stackrel{\text{def}}{=} \{(x, 1), (x, 3), (y, 4)\}. \tag{5}$$

We consider the powerset of D as a Kleene algebra, as explained in Section 3 (see (3)).

The input to the analysis consists of three matrices S, g and k representing respectively the CFG structure of the program, what is generated and what is killed at each label. For our example, these matrices are as follows, with g and k being abstractly defined in terms of parameters $g_j \subseteq D$ and $k_j \subseteq D$, for $j = 1, \ldots, 4$, whose value is given after.

$$S = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \qquad g = \begin{pmatrix} g_1 & 0 & 0 & 0 \\ 0 & g_2 & 0 & 0 \\ 0 & 0 & g_3 & 0 \\ 0 & 0 & 0 & g_4 \end{pmatrix} \qquad k = \begin{pmatrix} k_1 & 0 & 0 & 0 \\ 0 & k_2 & 0 & 0 \\ 0 & 0 & k_3 & 0 \\ 0 & 0 & 0 & k_4 \end{pmatrix}.$$

Recall that $1 = D$ (see (3)). Using the values already found in (1), we get the following as concrete instantiations for the example program (with g_j for $\mathsf{gen}(S_j)$ and k_j for $\mathsf{kill}(S_j)$):

$$\begin{array}{llll} g_1 \stackrel{\text{def}}{=} \{(x, 1)\}, & g_2 \stackrel{\text{def}}{=} \emptyset, & g_3 \stackrel{\text{def}}{=} \{(x, 3)\}, & g_4 \stackrel{\text{def}}{=} \{(y, 4)\}, \\ k_1 \stackrel{\text{def}}{=} \{(x, 3)\}, & k_2 \stackrel{\text{def}}{=} \emptyset, & k_3 \stackrel{\text{def}}{=} \{(x, 1)\}, & k_4 \stackrel{\text{def}}{=} \emptyset. \end{array} \tag{6}$$

Note that g and k are tests. Table 2 imposes the condition $\mathsf{gen}(S) \subseteq \overline{\mathsf{kill}(S)}$ for any atomic statement S; this translates to $g \leq \neg k$ for the matrices given above.

Our ultimate goal is to provide expressions for calculating matrices O and I, where $O[i, j]$ (resp. $I[i, j]$) is the set of data flow information outputted from (resp. inputted at) node j by the sequences of instructions occurring on the paths between i and j. The matrices S^*, G and $\overline{K}$ are used to define O and I (see Section 4.2). We have $S^*[i, j] = 1$ if and only if there exists a path (of length 0 or more) between nodes i and j, while $G[i, j]$ (resp. $\overline{K}[i, j]$) is the set of data flow information generated (resp. not killed) by the sequences of instructions occurring on paths between i and j. Note that a path between i and j may include i or j more than once.

Table 5 contains the equations that describe how to carry out RD analysis. This table has a simple and natural reading:

(1) G: To generate something, move on a path (S^*), generate that something (g), then continue moving on a path while not killing what was generated ($(S \cdot \neg k)^*$).
(2) $\overline{K}$: To not kill something, do not kill it on the first step and move along the program while not killing it, or generate it.

Table 5
RD existential ('may') parameters for CFGs

	RD
G	$S^* \cdot g \cdot (S \cdot \neg k)^*$
$\overline{K}$	$\neg k \cdot (S \cdot \neg k)^* + G$
O	G
I	$O \cdot S$

Complementation is relative to S^*.

(3) O: To output something, generate it.
(4) I: To input something to a node, output it to at least one of its predecessors ($O \cdot S$).

Given that I and O are the most interesting variables, one might express O directly in terms of S, g and k. The variables G and K could then be dropped from Table 5. However, as will be apparent in the sequel, G and K are still useful, in particular for comparing the approaches.

We use the table to calculate G and $\overline{K}$ for RD. It is a simple task to verify the following result (one has to use $g \leq \neg k$, i.e., $g_i \leq \neg k_i$, for $i = 1, \ldots, 4$). We give the result for the abstract matrices, because it is more instructive.

$$G = \begin{pmatrix} g_1 & g_2 + g_1 \cdot \neg k_2 & g_3 + g_2 \cdot \neg k_3 + g_1 \cdot \neg k_2 \cdot \neg k_3 & g_4 + g_2 \cdot \neg k_4 + g_1 \cdot \neg k_2 \cdot \neg k_4 \\ 0 & g_2 & g_3 + g_2 \cdot \neg k_3 & g_4 + g_2 \cdot \neg k_4 \\ 0 & 0 & g_3 & 0 \\ 0 & 0 & 0 & g_4 \end{pmatrix}$$

$$\overline{K} = \begin{pmatrix} \neg k_1 & g_2 + \neg k_1 \cdot \neg k_2 & g_3 + g_2 \cdot \neg k_3 + \neg k_1 \cdot \neg k_2 \cdot \neg k_3 & g_4 + g_2 \cdot \neg k_4 + \neg k_1 \cdot \neg k_2 \cdot \neg k_4 \\ 0 & \neg k_2 & g_3 + \neg k_2 \cdot \neg k_3 & g_4 + \neg k_2 \cdot \neg k_4 \\ 0 & 0 & \neg k_3 & 0 \\ 0 & 0 & 0 & \neg k_4 \end{pmatrix}.$$

Consider the entry $G[1, 3]$, for instance. This entry shows that what is generated when executing all paths from label 1 to label 3 – here, since there is a single such path, this means sequentially executing statements at labels 1, 2 and 3 – is what is generated at label 3, plus what is generated at label 2 and not killed at label 3, plus what is generated at label 1 and not killed at labels 2 and 3. Similarly, $\overline{K}[1, 2]$ shows that what is not killed on the path from label 1 to label 2 is either what is generated at label 2 or what is not killed at label 1 and not killed at label 2.

One can get K from the above matrix $\overline{K}$ by complementing, but complementation must be done relatively to S^*; the reason is that anything that gets killed is killed along a program path, and unconstrained complementation incorrectly introduces nonzero values outside program paths. The result is

$$K = \begin{pmatrix} k_1 & k_2 + k_1 \cdot \neg g_2 & k_3 + k_2 \cdot \neg g_3 + k_1 \cdot \neg g_2 \cdot \neg g_3 & k_4 + k_2 \cdot \neg g_4 + k_1 \cdot \neg g_2 \cdot \neg g_4 \\ 0 & k_2 & k_3 + k_2 \cdot \neg g_3 & k_4 + k_2 \cdot \neg g_4 \\ 0 & 0 & k_3 & 0 \\ 0 & 0 & 0 & k_4 \end{pmatrix}$$

and this is easily read and understood by looking at the CFG. Finally, $O = G$ and $I = O \cdot S$, i.e.,

$$I = \begin{pmatrix} 0 & g_1 & g_2 + g_1 \cdot \neg k_2 & g_2 + g_1 \cdot \neg k_2 \\ 0 & 0 & g_2 & g_2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

To compare these results with those of the classical approach presented in Section 2, it suffices to show the information produced along paths starting from the entry point. This can always be done by means of a row vector s selecting the entry point. For our program, given that the entry node corresponds to the first line of the matrices,

$$s \stackrel{\text{def}}{=} (1 \quad 0 \quad 0 \quad 0),$$

so that, for instance,

$$s \cdot I \;=\; (\,0 \quad g_1 \quad g_2 + g_1 \cdot \neg k_2 \quad g_2 + g_1 \cdot \neg k_2\,)\,.$$

Using (6), the concrete values are the following.

$$\begin{array}{l} s \cdot O = s \cdot G = (\,\{(x,1)\} \quad \{(x,1)\} \quad \{(x,3)\} \quad \{(x,1),(y,4)\}\,)\,, \\ s \cdot \overline{K} = (\,\{(x,1),(y,4)\} \quad \{(x,1),(y,4)\} \quad \{(x,3),(y,4)\} \quad \{(x,1),(y,4)\}\,)\,, \\ s \cdot K = (\,\{(x,3)\} \quad \{(x,3)\} \quad \{(x,1)\} \quad \{(x,3)\}\,)\,, \\ s \cdot I = (\,\emptyset \quad \{(x,1)\} \quad \{(x,1)\} \quad \{(x,1)\}\,)\,. \end{array}$$

Each $(s \cdot I)[i]$ and $(s \cdot O)[i]$ is the set of definitions reaching the entry and exit, respectively, of the atomic instruction at node i. These values agree with those given in Section 2.

Now, the functions gen, kill, in and out in Tables 3 and 4 take as parameters statements or programs, not program points. For instance, for the program P of the running example, $\mathsf{gen}(P)$ is what is generated by sequences of instructions occurring along paths between the entry point and the exit points of P. To derive this result from the above matrices, we use a a column vector t for selecting the exit points. For the program P,

$$t \stackrel{\text{def}}{=} \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}, \tag{7}$$

so that $s \cdot G \cdot t = G[1,3] + G[1,4] = \{(x,1),(x,3),(y,4)\}$ is indeed equal to $\mathsf{gen}(P)$. Similarly, $s \cdot \overline{K} \cdot t = \overline{K}[1,3] + \overline{K}[1,4] = \{(x,1),(x,3),(y,4)\} = \neg\mathsf{kill}(P)$ and $s \cdot O \cdot t = s \cdot G \cdot t = \mathsf{out}(P)$.

The story is different for I and K.

- In the case of I, the entry $I[i,j]$ is the set of definitions that get in node j due to paths starting at node i. Thus, $s \cdot I \cdot t = \{(x,1)\}$ is what gets in the exit nodes, while $\mathsf{in}(P)$ is what gets in the entry node. To find what gets in the entry node, it suffices to calculate $s \cdot I \cdot s'$, where s' is a column vector selecting the entry node (it is the transpose of s). For program P, this yields $s \cdot I \cdot s' = \emptyset = \mathsf{in}(P)$.
- In the case of K, $s \cdot K \cdot t = K[1,3] + K[1,4] = \{(x,1),(x,3)\}$, whereas $\mathsf{kill}(P) = \emptyset$. The reason for this behaviour is that for RD, *not killing*, like *generating*, is existential (additive), in the sense that results from converging paths are joined ('may' analysis). In the case of *killing*, these results should be intersected. But the effect of $s \cdot K \cdot t$ is to join all entries of the form $K[\text{entry point}, \text{exit point}]$ ($K[1,3] + K[1,4]$ for the example). Note that if the program has only one entry and one exit point, one may use either $s \cdot K \cdot t$ or $\neg(s \cdot \overline{K} \cdot t)$; the equivalence follows from the fact that s is then a total function, while t is injective and surjective. This distinction between existential and nonexistential parameters explains the caption of Table 5.

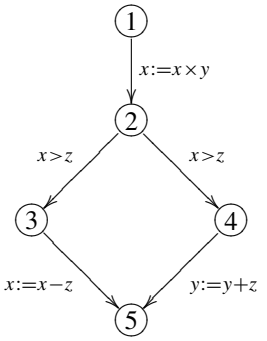
One may question why we set $O = G$ in Table 5, while Tables 3 and 4 distinguish out and gen. To explain the difference, consider the program while b do $(x := 1; y := 1)$ and suppose that the labels of $x := 1$ and $y := 1$ are 2 and 3, respectively. The classical analysis yields $\mathsf{gen}(2 : x := 1) = \{(x,2)\}$ and $\mathsf{out}(2 : x := 1) = \{(x,2),(y,3)\}$, while Table 5 gives $G[2,2] = O[2,2] = \mathsf{out}(2 : x := 1)$. Looking at the expression for G in Table 5, i.e., $S^* \cdot g \cdot (S \cdot \neg k)^*$, reveals why: this expression sums the effect of all paths starting from node 2 and returning to node 2. So, to calculate $\mathsf{gen}(P)$ with the matrix approach, it suffices to use the matrices for P, without the context of an enclosing program.

We now turn to the LTS representation of programs, which is often more natural. For instance, it is used for the representation of automata, or for giving relational descriptions of programs [24]. Our example program and its LTS graph are given in Fig. 3. As mentioned in Section 2, we do not distinguish between the two possible run-time values of the test $x > z$, since this has no impact on any of the four analyses.

The matrices S, g and k again respectively represent the structure of the program, what is generated and what is killed by atomic statements. Note that g and k are not tests as for the CFG representation. Rather, entries g_i and k_i label arrows and in a way can be viewed as an abstraction of the effect of the corresponding statements. The concrete instantiations (5) and (6) still apply.

Table 6, which contains the formulae that describe how to carry out RD analysis with the LTS representation of a program, can be used in the same manner as Table 5 to derive G, $\overline{K}$, O and I for the running example. It can be read

1 : $x := x \times y$;
2 : if $x > z$
3 : then $x := x - z$
4 : else $y := y + z$
5 :



$$S = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$g = \begin{pmatrix} 0 & g_1 & 0 & 0 & 0 \\ 0 & 0 & g_2 & g_2 & 0 \\ 0 & 0 & 0 & 0 & g_3 \\ 0 & 0 & 0 & 0 & g_4 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$k = \begin{pmatrix} 0 & k_1 & 0 & 0 & 0 \\ 0 & 0 & k_2 & k_2 & 0 \\ 0 & 0 & 0 & 0 & k_3 \\ 0 & 0 & 0 & 0 & k_4 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Fig. 3. LTS for the running example.

Table 6
RD existential ('may') parameters for LTSs

	RD
G	$S^* \cdot g \cdot (\overline{k} \sqcap S)^*$
$\overline{K}$	$(\overline{k} \sqcap S)^* + G$
O	G
I	O

Complementation is relative to S^*.

intuitively pretty much in the same manner as Table 5. It is obvious that $I = O$ is needed, provided that $I_L[m, n]$ is correctly interpreted as the data produced by paths from m to n that are fed to node n; but this is just the same as the data produced by paths from m to n that get out node n, since no transformation (instruction) occurs at node n. The calculation of G gives

$$G = \begin{pmatrix} 0 & g_1 & g_2 + g_1 \cdot \neg k_2 & g_2 + g_1 \cdot \neg k_2 & g_3 + g_4 + (g_2 + g_1 \cdot \neg k_2) \cdot (\neg k_3 + \neg k_4) \\ 0 & 0 & g_2 & g_2 & g_3 + g_4 + g_2 \cdot (\neg k_3 + \neg k_4) \\ 0 & 0 & 0 & 0 & g_3 \\ 0 & 0 & 0 & 0 & g_4 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

Using

$$s \stackrel{\text{def}}{=} (1 \quad 0 \quad 0 \quad 0 \quad 0),$$

and the concrete values given in (6), we find

$$s \cdot G = (\emptyset \quad \{(x, 1)\} \quad \{(x, 1)\} \quad \{(x, 1)\} \quad \{(x, 1), (x, 3), (y, 4)\}).$$

4.2. Gen/kill analysis for the four analyses

In this section, we present the equations for all four analyses. We begin with the CFG representation (Table 7). The following remarks are in order.

Table 7
Existential ('may') parameters for CFGs

	RD	LV	AE	VBE
G	$S^* \cdot g \cdot (S \cdot \neg k)^*$	$(\neg k \cdot S)^* \cdot g \cdot S^*$		
$\overline{G}$			$\neg g \cdot (S \cdot \neg g)^* + K$	$(\neg g \cdot S)^* \cdot \neg g + K$
K			$S^* \cdot k \cdot (S \cdot \neg g)^*$	$(\neg g \cdot S)^* \cdot k \cdot S^*$
$\overline{K}$	$\neg k \cdot (S \cdot \neg k)^* + G$	$(\neg k \cdot S)^* \cdot \neg k + G$		
O	G	$S \cdot I$		
I	$O \cdot S$	G		
$\overline{O}$			$\overline{G}$	$S \cdot \overline{I} + 1$
$\overline{I}$			$\overline{O} \cdot S + 1$	$\overline{G}$

Complementation is relative to S^*.

(1) Reading the expressions is mostly done as for RD. For LV and VBE, it is better to read the expressions backward, because they are backward analyses. The reading is then the same as for RD.
(2) The forward/backward and may/must dualities are very apparent.
   (a) The forward/backward correspondences RD $\leftrightarrow$ LV and AE $\leftrightarrow$ VBE are obtained by reading the expressions in the reverse direction and by switching in and out: $O \leftrightarrow I$. If we were in the framework of relation algebra instead of that of Kleene algebra, we could also use the relational converse operator $^\smile$; then, for instance for LV, $G = (\neg k \cdot S)^* \cdot g \cdot S^* = (S^{\smile *} \cdot g^\smile \cdot (S^\smile \cdot \neg k^\smile)^*)^\smile$. The same can be done for $\overline{K}$, I and O. Thus, to make an LV analysis, one can reverse the program, do the calculations of an RD analysis, reverse the result and switch I, O (of course, g and k are those for LV, not for RD). The same can be said about AE and VBE.
   (b) The may/must duality between RD and AE is first revealed by the fact that G, $\overline{K}$ and O are existential for RD, whereas $\overline{G}$, K and $\overline{O}$ are existential for AE (similar comment for LV and VBE). But the correspondence RD $\leftrightarrow$ AE and LV $\leftrightarrow$ VBE is deeper since gen and kill too are switched: $g \leftrightarrow k$, $G \leftrightarrow K$. The 1 in the AE expression for $\overline{I}$ is due to the fact that no expression is made available to a node on all paths starting from itself, since the empty path is also a path; by complementing, one has that all the expressions are not available at the entry of the node. The same comment is appropriate for the 1 in the VBE expression for $\overline{O}$.

   These dualities mean that only one kind of analysis is really necessary, since the other three can be obtained by simple substitutions (switching I and O) and simple additional operations (converse and complementation).
(3) All nonempty entries in Table 7 correspond to existential cases; collecting data with entry and exit vectors as we have done in Section 4.1 should be done with the parameters as given in Table 7 and not on their negation (unless there is only one entry and one exit point).
(4) Table 7 can be applied to programs with goto statements to fixed labels without any additional machinery.

The equations for LTSs are given in Table 8. Similar comments can be made about this table as for Table 7. Since $O = I$, 'in' and 'out' could be replaced by the concept 'at' describing the information at a node. For an LTS, this would be most appropriate, since no transformation occurs at a node.

4.3. Computational complexity

Suppose that the number of atomic statements is n, so that matrices for CFGs have n rows and n columns. The naive algorithm to compute the multiplication of two $n \times n$ matrices is $O(n^3)$. The question then naturally arises about the practicality of the KA approach with matrices, since multiplications and Kleene stars have to be computed. The classical approach, which is $O(n \cdot d)$, where d is the depth of the semi-lattice L of information (here, $L = \mathcal{P}(D)$), seems far more attractive. However, the matrix S is sparse (at most two nonzero entries on each line) and the matrices g and k are tests and thus even more sparse. They can thus be implemented by space efficient data structures such as linked lists. Also, it is not necessary to compute all entries of O and I, since only the row corresponding to the entry node of the program is needed. Using these techniques, the computation with matrices can be as efficient as the classical one. Thus, the software could present a nice algebraic interface, but be efficient nevertheless.

Table 8
Existential ('may') parameters for LTSs

	RD	LV	AE	VBE
G	$S^* \cdot g \cdot (\overline{k} \sqcap S)^*$	$(\overline{k} \sqcap S)^* \cdot g \cdot S^*$		
$\overline{G}$			$(\overline{g} \sqcap S)^* + K$	$(\overline{g} \sqcap S)^* + K$
K			$S^* \cdot k \cdot (\overline{g} \sqcap S)^*$	$(\overline{g} \sqcap S)^* \cdot k \cdot S^*$
$\overline{K}$	$(\overline{k} \sqcap S)^* + G$	$(\overline{k} \sqcap S)^* + G$		
O	G	I		
I	O	G		
$\overline{O}$			$\overline{G}$	$\overline{I}$
$\overline{I}$			$\overline{O}$	$\overline{G}$

Complementation is relative to S^*.

5. Linking the expressions for CFGs and LTSs

The formulae in Tables 7 and 8 are obviously related, but it is interesting to see how the connection can be described formally and this is what we now do.

The basic idea is best explained in terms of graphs. To go from a CFG to an LTS, it suffices to add a new node and to add arrows from the exit nodes of the CFG to the new node – which becomes the new and only exit node – and then to 'push' the information associated to nodes of the CFG to the appropriate arrows of the LTS.

Let us see how this is done with matrices. For these explanations, we append a subscript $_{\mathrm{L}}$ to the matrices related to an LTS. The following matrices S and S_{L} represent the structure of the CFG of Fig. 2 and that of the LTS of Fig. 3.

$$S = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \qquad S_{\mathrm{L}} = \left(\begin{array}{cccc|c} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ \hline 0 & 0 & 0 & 0 & 0 \end{array}\right) = \begin{pmatrix} S & t \\ 0 & 0 \end{pmatrix}. \tag{8}$$

The matrix S_{L} is structured as a matrix of four submatrices, one of which is the CFG matrix S and another is the column vector t that was used in (7) to select the exit nodes of the CFG. The rôle of this vector in S_{L} is to add links from the exit nodes of the CFG to the new node corresponding to column 5.

Now consider the CFG matrix g. It can be converted to the LTS matrix g, here called g_{L}, as follows.

$$g_{\mathrm{L}} = \begin{pmatrix} g \cdot S & g \cdot t \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} g & x \\ 0 & y \end{pmatrix} \cdot \begin{pmatrix} S & t \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} g & x \\ 0 & y \end{pmatrix} \cdot S_{\mathrm{L}}.$$

The value of the submatrices x and y does not matter, since these disappear in the result of the composition. One can use the concrete values given in Section 4.1 and check that indeed in that case $g_{\mathrm{L}} = \begin{pmatrix} g \cdot S & g \cdot t \\ 0 & 0 \end{pmatrix}$. This also holds generally. Indeed, suppose that $g[m, m] = g_m$. This says that the instruction in node m of the CFG generates g_m. Suppose that $S[m, n] = 1$, i.e., that n is a successor of m in the CFG. Then, g_m should be on the arrow between m and n in the LTS. This is produced by $g \cdot S$. Note that if m has many successors, g_m is put on the many corresponding arrows; this is what happens at CFG nodes containing program test instructions. Now suppose that m is an exit node of the CFG. Then, g_m should be moved on the arrow between m and the exit node of the LTS; this effect is obtained by $g \cdot t$. The matrix $\begin{pmatrix} g & x \\ 0 & y \end{pmatrix}$ is an embedding of g in a larger structure with an additional node. Composition with S_{L} 'pushes' the information provided by g on the appropriate arrows of g_{L}. The same comments can be made for k.

5.1. Relationship between CFG and LTS expressions for RD analysis

We will show that the RD CFG expressions given in Table 7 can be transformed into the corresponding RD LTS expressions given in Table 8. To distinguish expressions related to CFGs and LTSs, we add a subscript $_{\mathrm{L}}$ to variables in the latter expressions.

The first step is to lay out the hypotheses we will use in the proof. In these hypotheses, the function $f(a) \stackrel{\text{def}}{=} \begin{pmatrix} a & t \\ 0 & 0 \end{pmatrix}$ is used as well as variable t whose rôle is to link the exit nodes of the CFG to the exit node of the LTS (as in the matrix example above). The hypotheses follow.

$$\begin{array}{lll} S_{\mathrm{L}} = f(S) & g_{\mathrm{L}} = f(g)\cdot f(S) & k_{\mathrm{L}} = f(k)\cdot f(S) \\ O_{\mathrm{L}} = I_{\mathrm{L}} = G_{\mathrm{L}} & G_{\mathrm{L}} = \begin{pmatrix} G\cdot S & G\cdot t \\ 0 & 0 \end{pmatrix} & \overline{K_{\mathrm{L}}} = 1 + \begin{pmatrix} \overline{K}\cdot S & \overline{K}\cdot t \\ 0 & 0 \end{pmatrix}. \end{array} \tag{9}$$

The intuitive justification for the equations concerning S_{L}, g_{L} and k_{L} was given above in terms of concrete matrices. The equations $O_{\mathrm{L}} = I_{\mathrm{L}} = G_{\mathrm{L}}$ are forced by Table 8. We proceed with the justification of the equations for G_{L} and $\overline{K_{\mathrm{L}}}$; this is done in terms of the concrete matrices G and K. The justification is necessarily informal and reflects the intended semantics of the various parameters.

G_{L}: The case of G is similar to that of g, even though G is not a test. Suppose that $G[m, n] = p$, for some p. This says that p is generated along paths from node m to node n (more precisely, each $x \in p$ is generated along some path). Note that this includes the effect of the instruction at node n in the CFG; but since this instruction is pushed on the arrows going to the successors of n and to the exit node of the LTS if n is an exit node of the CFG, we must set $G_{\mathrm{L}}[m, j] = p$, for each successor j of n and for j the exit node if n is an exit node of G. This is the effect of $G\cdot S$ and $G\cdot t$ in the expression for G_{L} in (9).

$\overline{K_{\mathrm{L}}}$: The matrix $\overline{K}$ could be treated exactly like G. The interpretation of $\overline{K_{\mathrm{L}}}$ would then be that its cells contain data not killed over positive-length paths. But it is natural to consider that nothing is killed at an LTS node (since it does not contain an instruction). This explains the addition of 1 in the equation for $\overline{K_{\mathrm{L}}}$ in (9). Treating $\overline{K_{\mathrm{L}}}$ like G_{L} is possible; however, adding 1 in the equation for $\overline{K_{\mathrm{L}}}$ saves us from using complementation with respect to S^+ in LTS expressions, and this is cleaner.

We are now ready to derive the expressions for RD given in the LTS table (Table 8). We begin with G, whose CFG expression is $S^*\cdot g\cdot(S\cdot\neg k)^*$, and show that it transforms to $G_{\mathrm{L}} = S_{\mathrm{L}}^*\cdot g_{\mathrm{L}}\cdot(\overline{k_{\mathrm{L}}}\sqcap S_{\mathrm{L}})^*$. We first note the following two properties:

$$\begin{pmatrix} a & ? \\ 0 & ? \end{pmatrix}\cdot\begin{pmatrix} b & ? \\ 0 & ? \end{pmatrix} = \begin{pmatrix} a\cdot b & ? \\ 0 & ? \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} a & ? \\ 0 & ? \end{pmatrix}^* = \begin{pmatrix} a^* & ? \\ 0 & ? \end{pmatrix}, \tag{10}$$

where '?' means that the exact value is not important for our purpose (we use the same convention below; it is always possible to calculate what an instance of '?' is, but this just clutters the picture). Before giving the main derivation, we prove the auxiliary result

$$f(\neg k)\cdot f(S) = \overline{k_{\mathrm{L}}}\sqcap S_{\mathrm{L}}. \tag{11}$$

$$\begin{array}{ll} & f(\neg k)\cdot f(S) \\ = & \quad \langle\ \text{Definition of } f\ \rangle \\ & \begin{pmatrix} \neg k & t \\ 0 & 0 \end{pmatrix}\cdot\begin{pmatrix} S & t \\ 0 & 0 \end{pmatrix} \\ = & \quad \langle\ \text{The two matrix multiplications give the same result}\ \rangle \\ & \begin{pmatrix} \neg k & 0 \\ 0 & 0 \end{pmatrix}\cdot\begin{pmatrix} S & t \\ 0 & 0 \end{pmatrix} \\ = & \quad \langle\ \text{The left matrix is a test}\ \rangle \\ & \neg\begin{pmatrix} k & 0 \\ 0 & 0 \end{pmatrix}\cdot\begin{pmatrix} S & t \\ 0 & 0 \end{pmatrix} \\ = & \quad \langle\ \text{In a Boolean KA, for any } a \text{ and test } p,\ \neg p\cdot a = \overline{p\cdot a}\sqcap a\ \rangle \\ & \overline{\begin{pmatrix} k & 0 \\ 0 & 0 \end{pmatrix}\cdot\begin{pmatrix} S & t \\ 0 & 0 \end{pmatrix}}\sqcap\begin{pmatrix} S & t \\ 0 & 0 \end{pmatrix} \\ = & \quad \langle\ \text{The two matrix multiplications give the same result}\ \rangle \\ & \overline{\begin{pmatrix} k & t \\ 0 & 0 \end{pmatrix}\cdot\begin{pmatrix} S & t \\ 0 & 0 \end{pmatrix}}\sqcap\begin{pmatrix} S & t \\ 0 & 0 \end{pmatrix} \end{array}$$

$$
\begin{array}{ll}
= & \langle \text{ Definition of } f \rangle \\
& \overline{f(k)\cdot f(S)} \sqcap f(S) \\
= & \langle \text{ Assumptions (9)} \rangle \\
& \overline{k_{\mathrm{L}}} \sqcap S_{\mathrm{L}}.
\end{array}
$$

And now comes the proof that $G_{\mathrm{L}} = S_{\mathrm{L}}^{*}\cdot g_{\mathrm{L}}\cdot(\overline{k_{\mathrm{L}}} \sqcap S_{\mathrm{L}})^{*}$.

$$
\begin{array}{ll}
& G_{\mathrm{L}} \\
= & \langle \text{ Assumptions (9)} \rangle \\
& \begin{pmatrix} G\cdot S & G\cdot t \\ 0 & 0 \end{pmatrix} \\
= & \langle \text{ Matrix multiplication} \rangle \\
& \begin{pmatrix} G & ? \\ 0 & ? \end{pmatrix}\cdot\begin{pmatrix} S & t \\ 0 & 0 \end{pmatrix} \\
= & \langle \text{ Expression for } G \text{ and definition of } f \rangle \\
& \begin{pmatrix} S^{*}\cdot g\cdot(S\cdot\neg k)^{*} & ? \\ 0 & ? \end{pmatrix}\cdot f(S) \\
= & \langle \text{ Definition of } f \text{ and induction on the structure of the} \\
& \quad \text{expression using (10)} \rangle \\
& (f(S))^{*}\cdot f(g)\cdot(f(S)\cdot f(\neg k))^{*}\cdot f(S) \\
= & \langle \text{ (2(d))} \rangle \\
& (f(S))^{*}\cdot f(g)\cdot f(S)\cdot(f(\neg k)\cdot f(S))^{*} \\
= & \langle \text{ Assumptions (9), and (11)} \rangle \\
& S_{\mathrm{L}}^{*}\cdot g_{\mathrm{L}}\cdot(\overline{k_{\mathrm{L}}} \sqcap S_{\mathrm{L}})^{*}\ .
\end{array}
$$

The transformation of the CFG subexpression $\neg k\cdot(S\cdot\neg k)^{*}$ (appearing in the definition of $\overline{K}$ in Table 7) is done in a similar fashion, except that the last steps are:

$$
\begin{array}{ll}
& f(\neg k)\cdot(f(S)\cdot f(\neg k))^{*}\cdot f(S) \\
= & \langle \text{ (2(d))} \rangle \\
& f(\neg k)\cdot f(S)\cdot(f(\neg k)\cdot f(S))^{*} \\
= & \langle\ a\cdot a^{*} = a^{+} \text{ and (11)} \rangle \\
& (\overline{k_{\mathrm{L}}} \sqcap S_{\mathrm{L}})^{+}\ .
\end{array}
$$

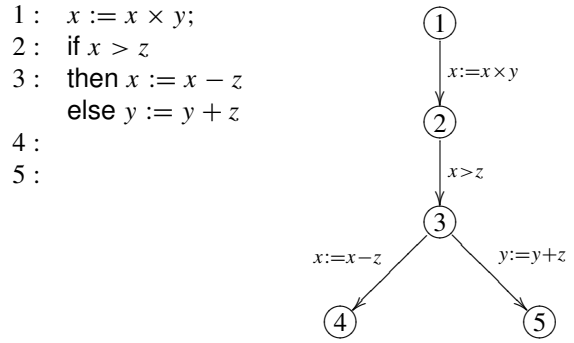
Using this, plus the definition of $\overline{K}$ (Table 7) and the assumption about $\overline{K_{\mathrm{L}}}$ (9), one easily arrives at $\overline{K_{\mathrm{L}}} = (\overline{k_{\mathrm{L}}} \sqcap S_{\mathrm{L}})^{*} + G_{\mathrm{L}}$.

Finally, we show that the calculation of the 'out' information along paths between entry and exit nodes gives the same result for CFGs and LTSs, that is, $s_{\mathrm{L}}\cdot O_{\mathrm{L}}\cdot t_{\mathrm{L}} = s\cdot O\cdot t$. We use $s_{\mathrm{L}} = \begin{pmatrix} s & 0 \end{pmatrix}$ and $t_{\mathrm{L}} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$, meaning essentially that the additional node cannot be an entry point and must be the only exit point. The following derivation uses (9) and simple matrix laws.

$$
s_{\mathrm{L}}\cdot O_{\mathrm{L}}\cdot t_{\mathrm{L}} = \begin{pmatrix} s & 0 \end{pmatrix}\cdot\begin{pmatrix} O\cdot S & O\cdot t \\ 0 & 0 \end{pmatrix}\cdot\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} s & 0 \end{pmatrix}\cdot\begin{pmatrix} O\cdot t \\ 0 \end{pmatrix} = s\cdot O\cdot t.
$$

But we cannot claim $s_{\mathrm{L}}\cdot I_{\mathrm{L}}\cdot t_{\mathrm{L}} = s\cdot I\cdot t$, since $O_{\mathrm{L}} = I_{\mathrm{L}}$, while in general $s\cdot O\cdot t \neq s\cdot I\cdot t$.

The transformation from CFGs to LTSs for AE analysis is similar. However, the backward analyses require a special treatment. We explain that with LV analysis.



$$S = \left(\begin{array}{c|cccc} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 \end{array}\right)$$

$$g = \left(\begin{array}{c|cccc} 0 & g_1 & 0 & 0 & 0 \\ 0 & 0 & g_2 & 0 & 0 \\ 0 & 0 & 0 & g_3 & g_4 \\ 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 \end{array}\right)$$

$$k = \left(\begin{array}{c|cccc} 0 & k_1 & 0 & 0 & 0 \\ 0 & 0 & k_2 & 0 & 0 \\ 0 & 0 & 0 & k_3 & k_4 \\ 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 \end{array}\right)$$

Fig. 4. CFG to LTS: Adding a new entry node.

5.2. Relationship between CFG and LTS expressions for LV analysis

To care for the backward analysis LV, one could take a completely dual approach to the one we have used for the forward analysis RD. To convert the CFG into an LTS, it suffices to add a new entry node (instead of an exit node). Fig. 4 shows how the CFG of Fig. 2 is transformed (compare to the LTS of Fig. 3). The structure of the new matrices is also shown (compare with those of Fig. 3). The results we have proved for the CFG–LTS conversion for RD then immediately apply to the CFG–LTS conversion for LV by duality.

So, there is an LTS such that the expressions of Table 8 for LV correspond to those of Table 7 for LV. This does not completely answer the question, though, because the LTS in Fig. 4 is not the standard LTS for the corresponding program. Fortunately, the results we have shown in the previous subsection for RD hold for LV, with I and O switched, when the LTS is obtained by adding an exit node (like that of Fig. 3, which is the standard LTS). The reason is that Assumptions (9) still hold, as can be checked.

6. Soundness of the KA approach

The goal of this section is to prove that the data flow information provided by the classical approach is also provided by the KA approach. We give the proof for the CFG representation and RD analysis. Section 6.1 deals with the correspondence between gen, kill and G, K, while Section 6.2 deals with that between in, out and I, O.

6.1. Correspondence of gen, kill with G, K

We will show that calculating what is generated and what is killed by a program gives the same result with the KA approach as with the classical approach. The following notation will be used:

(1) S, S_1 and S_2 denote programs or statements and also the matrices that represent the structure of these programs. It will be easy to tell from the context if a matrix or a program is meant;
(2) s, s_1 and s_2 denote the row vectors that select the entry node of programs S, S_1 and S_2;
(3) t, t_1 and t_2 denote the column vectors that select the exit nodes of programs S, S_1 and S_2.

It is assumed that (CFGs representing) programs have a single entry node but may have many exit nodes, due to conditional instructions. It is also assumed that there is a path from the entry node to any node; this is always the case for structured programs.

We will show by induction on the structure of a program S that

$$s \cdot G \cdot t = \mathsf{gen}(S) \quad \text{and} \quad s \cdot \overline{K} \cdot t = \neg\mathsf{kill}(S), \tag{12}$$

where G and $\overline{K}$ are given in Table 7.

The base case is when S is an atomic statement or a test and it is trivial, since all matrices have only one entry, so that

$$s \cdot G \cdot t = 1 \cdot S^* \cdot g \cdot (S \cdot \neg k)^* \cdot 1 = 1 \cdot 1 \cdot g \cdot 1 \cdot 1 = g = \mathsf{gen}(S)$$

and

$$s \cdot \overline{K} \cdot t = 1 \cdot (\neg k \cdot (S \cdot \neg k)^* + G) \cdot 1 = \neg k \cdot 1 + g = \neg k = \neg\mathsf{kill}(S),$$

where $g \leq \neg k$ has been used, as well as the fact that g and k are by definition equal to $\mathsf{gen}(S)$ and $\mathsf{kill}(S)$.

For the inductive step, there are three cases to consider: sequences, conditionals and loops.

Let the sequence be $S \stackrel{\text{def}}{=} S_1; S_2$. For the proof, we use the matrices

$$\begin{array}{lll} S \stackrel{\text{def}}{=} \begin{pmatrix} S_1 & t_1 \cdot s_2 \\ 0 & S_2 \end{pmatrix}, & g \stackrel{\text{def}}{=} \begin{pmatrix} g_1 & 0 \\ 0 & g_2 \end{pmatrix}, & k \stackrel{\text{def}}{=} \begin{pmatrix} k_1 & 0 \\ 0 & k_2 \end{pmatrix}, \\ s \stackrel{\text{def}}{=} \begin{pmatrix} s_1 & 0 \end{pmatrix}, & t \stackrel{\text{def}}{=} \begin{pmatrix} 0 \\ t_2 \end{pmatrix} & \end{array} \tag{13}$$

and the following abbreviation and laws (due to the definition of K in Table 7):

$$K_2' \stackrel{\text{def}}{=} \neg k_2 \cdot (S_2 \cdot \neg k_2)^*, \qquad \overline{K_2} = K_2' + G_2, \qquad K_2' \leq \overline{K_2}. \tag{14}$$

The vector s selects the entry node of S_1 as the entry node of S and vector t selects the exit nodes of S_2 as the exit nodes of S. In matrix S, the submatrix $t_1 \cdot s_2$ connects the exit nodes of S_1 to the entry node of S_2. Hence, matrix S represents the structure of the sequence $S_1; S_2$. We calculate $s \cdot G \cdot t$.

$$\begin{array}{ll} & s \cdot G \cdot t \\ = & \quad \langle \text{ Definition of } G \text{ in Table 7 } \rangle \\ & s \cdot S^* \cdot g \cdot (S \cdot \neg k)^* \cdot t \\ = & \quad \langle \text{ (13) and multiplying } S \text{ with } \neg k \rangle \\ & s \cdot S^* \cdot g \cdot \begin{pmatrix} S_1 \cdot \neg k_1 & t_1 \cdot s_2 \cdot \neg k_2 \\ 0 & S_2 \cdot \neg k_2 \end{pmatrix}^* \cdot t \\ = & \quad \langle \text{ (13) and definition of } {}^* \text{ for matrices } \rangle \\ & s \cdot S^* \cdot \begin{pmatrix} g_1 & 0 \\ 0 & g_2 \end{pmatrix} \cdot \begin{pmatrix} (S_1 \cdot \neg k_1)^* & (S_1 \cdot \neg k_1)^* \cdot t_1 \cdot s_2 \cdot \neg k_2 \cdot (S_2 \cdot \neg k_2)^* \\ 0 & (S_2 \cdot \neg k_2)^* \end{pmatrix} \cdot \begin{pmatrix} 0 \\ t_2 \end{pmatrix} \\ = & \quad \langle \text{ Matrix multiplication } \rangle \\ & s \cdot S^* \cdot \begin{pmatrix} g_1 \cdot (S_1 \cdot \neg k_1)^* \cdot t_1 \cdot s_2 \cdot \neg k_2 \cdot (S_2 \cdot \neg k_2)^* \cdot t_2 \\ g_2 \cdot (S_2 \cdot \neg k_2)^* \cdot t_2 \end{pmatrix} \\ = & \quad \langle \text{ (13), definition of } {}^* \text{ for matrices, and (14) } \rangle \\ & \begin{pmatrix} s_1 & 0 \end{pmatrix} \cdot \begin{pmatrix} S_1^* & S_1^* \cdot t_1 \cdot s_2 \cdot S_2^* \\ 0 & S_2^* \end{pmatrix} \cdot \begin{pmatrix} g_1 \cdot (S_1 \cdot \neg k_1)^* \cdot t_1 \cdot s_2 \cdot K_2' \cdot t_2 \\ g_2 \cdot (S_2 \cdot \neg k_2)^* \cdot t_2 \end{pmatrix} \\ = & \quad \langle \text{ Matrix multiplication } \rangle \\ & s_1 \cdot S_1^* \cdot g_1 \cdot (S_1 \cdot \neg k_1)^* \cdot t_1 \cdot s_2 \cdot K_2' \cdot t_2 \ + \ s_1 \cdot S_1^* \cdot t_1 \cdot s_2 \cdot S_2^* \cdot g_2 \cdot (S_2 \cdot \neg k_2)^* \cdot t_2 \\ = & \quad \langle \text{ Definition of } G \text{ in Table 7 } \rangle \\ & s_1 \cdot G_1 \cdot t_1 \cdot s_2 \cdot K_2' \cdot t_2 \ + \ s_1 \cdot S_1^* \cdot t_1 \cdot s_2 \cdot G_2 \cdot t_2 \\ = & \quad \langle \ s_1 \cdot S_1^* \cdot t_1 = 1, \text{ since there is a path from entry to exit of } S_1; \\ & \quad \text{distributivity and } s_1 \cdot G_1 \cdot t_1 \leq 1 \ \rangle \\ & s_1 \cdot G_1 \cdot t_1 \cdot s_2 \cdot (K_2' + G_2) \cdot t_2 \ + \ s_2 \cdot G_2 \cdot t_2 \end{array}$$

$$
\begin{array}{ll}
= & \langle\ (14) \text{ and induction hypothesis } (12)\ \rangle \\
& \mathsf{gen}(S_1)\cdot\neg\mathsf{kill}(S_2)\ +\ \mathsf{gen}(S_2) \\
= & \langle\ \text{Using set notation}\ \rangle \\
& (\mathsf{gen}(S_1) - \mathsf{kill}(S_2)) \cup \mathsf{gen}(S_2) \\
= & \langle\ \text{Table 3}\ \rangle \\
& \mathsf{gen}(S).
\end{array}
$$

The proof that $s\cdot\overline{K}\cdot t = \neg\mathsf{kill}(S)$ is similar. And the proofs for the cases of the conditional and the loop are just as simple. The matrices to use for the conditional $S \stackrel{\text{def}}{=} \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2$ are:

$$
S \stackrel{\text{def}}{=} \begin{pmatrix} 0 & s_1 & s_2 \\ 0 & S_1 & 0 \\ 0 & 0 & S_2 \end{pmatrix}, \qquad g \stackrel{\text{def}}{=} \begin{pmatrix} 0 & 0 & 0 \\ 0 & g_1 & 0 \\ 0 & 0 & g_2 \end{pmatrix}, \qquad k \stackrel{\text{def}}{=} \begin{pmatrix} 0 & 0 & 0 \\ 0 & k_1 & 0 \\ 0 & 0 & k_2 \end{pmatrix},
$$

$$
s \stackrel{\text{def}}{=} (\,1 \quad 0 \quad 0\,), \qquad t \stackrel{\text{def}}{=} \begin{pmatrix} 0 \\ t_1 \\ t_2 \end{pmatrix}
$$

and those for the loop $S \stackrel{\text{def}}{=} \mathsf{while}\ b\ \mathsf{do}\ S_1$ are

$$
S \stackrel{\text{def}}{=} \begin{pmatrix} 0 & s_1 \\ t_1 & S_1 \end{pmatrix}, \qquad g \stackrel{\text{def}}{=} \begin{pmatrix} 0 & 0 \\ 0 & g_1 \end{pmatrix}, \qquad k \stackrel{\text{def}}{=} \begin{pmatrix} 0 & 0 \\ 0 & k_1 \end{pmatrix},
$$

$$
s \stackrel{\text{def}}{=} (\,1 \quad 0\,), \qquad t \stackrel{\text{def}}{=} \begin{pmatrix} 1 \\ 0 \end{pmatrix}.
$$

6.2. *Correspondence of* in, out *with* I, O

We want to show that O and I satisfy the constraints of Table 4. More precisely, let S be a program whose entry node is α. Consider an arbitrary node α_2 of the CFG representing S; this node is the entry node of at least one statement inside S, namely the atomic instruction at node α_2. Now let S_2 be an arbitrary statement (not necessarily atomic) inside S that has α_2 as its entry node and call S_1 the part of S which is not S_2. Without loss of generality, we can represent program S by the matrix

$$
S \stackrel{\text{def}}{=} \begin{pmatrix} S_1 & t_1\cdot s_2 \\ t_2\cdot s_1 & S_2 \end{pmatrix}
$$

(rows and columns can always be permuted to get this shape). As in Section 6.1, s_2 is a row vector selecting the entry point of S_2 (i.e., α_2) and t_2 is a column vector selecting the exit nodes of S_2. We will also use the row vector s to select the entry node of S, i.e., α. We assume that S is well structured, so that α is either α_2 or a node of S_1 (i.e., the entry node of S cannot be strictly inside the inner statement S_2). The rôle of row vector s_1 is to select the point of return from S_2 to S_1, if any, and the rôle of column vector t_1 is to select the nodes of S_1 that have α_2 as successor. The entries $t_1\cdot s_2$ and $t_2\cdot s_1$ link S_1 to S_2 and S_2 to S_1.

The test matrices g and k are

$$
g \stackrel{\text{def}}{=} \begin{pmatrix} g_1 & 0 \\ 0 & g_2 \end{pmatrix} \quad \text{and} \quad k \stackrel{\text{def}}{=} \begin{pmatrix} k_1 & 0 \\ 0 & k_2 \end{pmatrix}.
$$

We will prove that

$$
I[\alpha, \alpha_2] = \textstyle\sum(\alpha' \mid \alpha' \in \mathsf{pred}(\alpha_2) : O[\alpha, \alpha']) \tag{15}
$$

and

$$
s\cdot O\cdot \begin{pmatrix} 0 \\ t_2 \end{pmatrix} = \mathsf{gen}(S_2) \cup (I[\alpha, \alpha_2] - \mathsf{kill}(S_2)). \tag{16}
$$

Because α is the entry node of S and α_2 that of S_2, $I[\alpha, \alpha_2]$ and $\alpha' \in \mathsf{pred}(\alpha_2)$ might as well be written as $\mathsf{in}(S_2)$ and $\alpha' \in \mathsf{pred}(S_2)$, respectively. Similarly, $s \cdot O \cdot \begin{pmatrix} 0 \\ t_2 \end{pmatrix}$ is what gets out of the exit nodes of program S_2, i.e., $\mathsf{out}(S_2)$. This means that (15) and (16) correspond to the classical equations given in Table 4. The correspondence can be made more formal by considering nodes (atomic statements) instead of arbitrary statements. Suppose that S_2 is an atomic statement. Then α_2 is also the exit node of S_2, so that $s \cdot O \cdot \begin{pmatrix} 0 \\ t_2 \end{pmatrix} = O[\alpha, \alpha_2]$. Hence, by (15) and (16), $I[\alpha, \alpha_2]$ and $O[\alpha, \alpha_2]$ satisfy exactly the classical equations of Table 4. Thus the row matrices $s \cdot I$ and $s \cdot O$ give exactly the classical results at each node.

The rest of the section is devoted to the proof of (15) and (16). We begin with (15).

$$
\begin{array}{ll}
 & I[\alpha, \alpha_2] \\
= & \quad \langle \text{ Definition of } I \text{ in Table 7 } \rangle \\
 & (O \cdot S)[\alpha, \alpha_2] \\
= & \quad \langle \text{ Definition of matrix composition } \rangle \\
 & \sum(\alpha' \mid: O[\alpha, \alpha'] \cdot S[\alpha', \alpha_2]) \\
= & \quad \langle \text{ Either } S[\alpha', \alpha_2] = 0 \text{ or } S[\alpha', \alpha_2] = 1, \\
 & \quad \text{and } S[\alpha', \alpha_2] = 1 \;\Leftrightarrow\; \alpha' \in \mathsf{pred}(\alpha_2) \; \rangle \\
 & \sum(\alpha' \mid \alpha' \in \mathsf{pred}(\alpha_2) : O[\alpha, \alpha']).
\end{array}
$$

The following property is needed later on:

$$G \cdot S \cdot \overline{K} \leq O. \tag{17}$$

$$
\begin{array}{ll}
 & G \cdot S \cdot \overline{K} \\
= & \quad \langle \text{ Definitions in Table 7 } \rangle \\
 & S^* \cdot g \cdot (S \cdot \neg k)^* \cdot S \cdot (\neg k \cdot (S \cdot \neg k)^* + S^* \cdot g \cdot (S \cdot \neg k)^*) \\
= & \quad \langle \text{ Distributivity of } \cdot \text{ over } + \; \rangle \\
 & S^* \cdot g \cdot (S \cdot \neg k)^* \cdot S \cdot \neg k \cdot (S \cdot \neg k)^* \;+\; S^* \cdot g \cdot (S \cdot \neg k)^* \cdot S \cdot S^* \cdot g \cdot (S \cdot \neg k)^* \\
\leq & \quad \langle \text{ (2(a)), } g \leq 1 \text{ and } \neg k \leq 1 \; \rangle \\
 & S^* \cdot g \cdot (S \cdot \neg k)^* \;+\; S^* \cdot S^* \cdot S \cdot S^* \cdot g \cdot (S \cdot \neg k)^* \\
= & \quad \langle \text{ (2(a)) } \rangle \\
 & S^* \cdot g \cdot (S \cdot \neg k)^* \\
= & \quad \langle \text{ Definition of } G \text{ and } O \text{ in Table 7 } \rangle \\
 & O.
\end{array}
$$

And now the proof of (16). In the proof, we use the column vector s_2', which is the transpose of s_2, and the abbreviations

$$
A \stackrel{\text{def}}{=} \begin{pmatrix} S_1 \cdot \neg k_1 & t_1 \cdot s_2 \cdot \neg k_2 \\ t_2 \cdot s_1 \cdot \neg k_1 & 0 \end{pmatrix}, \qquad B \stackrel{\text{def}}{=} \begin{pmatrix} 0 & 0 \\ 0 & S_2 \cdot \neg k_2 \end{pmatrix},
$$

$$
C \stackrel{\text{def}}{=} \begin{pmatrix} S_1 & t_1 \cdot s_2 \\ t_2 \cdot s_1 & 0 \end{pmatrix}, \qquad D \stackrel{\text{def}}{=} \begin{pmatrix} 0 & 0 \\ 0 & S_2 \end{pmatrix}.
$$

It is easy to check that

$$A + B = S \cdot \neg k, \qquad C + D = S, \qquad s_2 \cdot s_2' = 1. \tag{18}$$

$$
\begin{array}{ll}
 & s \cdot O \cdot \begin{pmatrix} 0 \\ t_2 \end{pmatrix} \\
= & \quad \langle \text{ Definition of } O \text{ and } G \text{ in Table 7, and (18) } \rangle
\end{array}
$$

$$
\begin{array}{ll}
& s \cdot S^* \cdot g \cdot (A + B)^* \cdot \begin{pmatrix} 0 \\ t_2 \end{pmatrix} \\
= & \quad \langle\ (2(\mathrm{b}))\ \rangle \\
& s \cdot S^* \cdot g \cdot (B^* \cdot A)^* \cdot B^* \cdot \begin{pmatrix} 0 \\ t_2 \end{pmatrix} \\
= & \quad \langle\ (2(\mathrm{c})) \text{ and calculating } B^*\ \rangle \\
& s \cdot S^* \cdot g \cdot (1 + (B^* \cdot A)^* \cdot B^* \cdot A) \begin{pmatrix} 1 & 0 \\ 0 & (S_2 \cdot \neg k_2)^* \end{pmatrix} \cdot \begin{pmatrix} 0 \\ t_2 \end{pmatrix} \\
= & \quad \langle\ \text{Matrix multiplication and distributivity of } \cdot \text{ over } +\ \rangle \\
& s \cdot S^* \cdot g \cdot \begin{pmatrix} 0 \\ (S_2 \cdot \neg k_2)^* \cdot t_2 \end{pmatrix} + s \cdot S^* \cdot g \cdot (B^* \cdot A)^* \cdot B^* \cdot A \cdot \begin{pmatrix} 0 \\ (S_2 \cdot \neg k_2)^* \cdot t_2 \end{pmatrix} \\
= & \quad \langle\ (2(\mathrm{b})), \text{ definition of } g, \text{ multiplying } g \text{ with right neighbour}\ \rangle \\
& s \cdot S^* \cdot \begin{pmatrix} 0 \\ g_2 \cdot (S_2 \cdot \neg k_2)^* \cdot t_2 \end{pmatrix} + s \cdot S^* \cdot g \cdot (A + B)^* \cdot A \cdot \begin{pmatrix} 0 \\ (S_2 \cdot \neg k_2)^* \cdot t_2 \end{pmatrix} \\
= & \quad \langle\ (18) \text{ and definition of } G \text{ in Table 7}\ \rangle \\
& s \cdot S^* \cdot \begin{pmatrix} 0 \\ g_2 \cdot (S_2 \cdot \neg k_2)^* \cdot t_2 \end{pmatrix} + s \cdot G \cdot A \cdot \begin{pmatrix} 0 \\ (S_2 \cdot \neg k_2)^* \cdot t_2 \end{pmatrix} \\
= & \quad \langle\ (18), (2(\mathrm{b})) \text{ and definition of } A\ \rangle \\
& s \cdot (D^* \cdot C)^* \cdot D^* \cdot \begin{pmatrix} 0 \\ g_2 \cdot (S_2 \cdot \neg k_2)^* \cdot t_2 \end{pmatrix} + \\
& s \cdot G \cdot \begin{pmatrix} S_1 \cdot \neg k_1 & t_1 \cdot s_2 \cdot \neg k_2 \\ t_2 \cdot s_1 \cdot \neg k_1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ (S_2 \cdot \neg k_2)^* \cdot t_2 \end{pmatrix} \\
= & \quad \langle\ \text{Calculating } D^* \text{ and matrix multiplication}\ \rangle \\
& s \cdot (D^* \cdot C)^* \cdot \begin{pmatrix} 1 & 0 \\ 0 & S_2^* \end{pmatrix} \cdot \begin{pmatrix} 0 \\ g_2 \cdot (S_2 \cdot \neg k_2)^* \cdot t_2 \end{pmatrix} + \\
& s \cdot G \cdot \begin{pmatrix} 0 & t_1 \cdot s_2 \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ \neg k_2 \cdot (S_2 \cdot \neg k_2)^* \cdot t_2 \end{pmatrix} \\
= & \quad \langle\ (2(\mathrm{c})), \text{ matrix multiplication, definition of } G \text{ and } (14)\ \rangle \\
& s \cdot (1 + (D^* \cdot C)^* \cdot D^* \cdot C) \cdot \begin{pmatrix} 0 \\ G_2 \cdot t_2 \end{pmatrix} + s \cdot G \cdot \begin{pmatrix} 0 & t_1 \cdot s_2 \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ K_2' \cdot t_2 \end{pmatrix} \\
= & \quad \langle\ \text{Distributivity of } \cdot \text{ over } +, \text{ definition of } C \text{ and } (18)\ \rangle \\
& s \cdot \begin{pmatrix} 0 \\ G_2 \cdot t_2 \end{pmatrix} + s \cdot (D^* \cdot C)^* \cdot D^* \cdot \begin{pmatrix} S_1 & t_1 \cdot s_2 \\ t_2 \cdot s_1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ G_2 \cdot t_2 \end{pmatrix} + \\
& s \cdot G \cdot \begin{pmatrix} 0 & t_1 \cdot s_2 \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ s_2' \end{pmatrix} \cdot (0 \quad s_2) \cdot \begin{pmatrix} 0 \\ K_2' \cdot t_2 \end{pmatrix} \\
= & \quad \langle\ (2(\mathrm{b})), (18) \text{ and matrix multiplication}\ \rangle \\
& s \cdot \begin{pmatrix} 0 \\ G_2 \cdot t_2 \end{pmatrix} + s \cdot S^* \cdot \begin{pmatrix} 0 & t_1 \cdot s_2 \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ G_2 \cdot t_2 \end{pmatrix} + \\
& s \cdot G \cdot \begin{pmatrix} 0 & t_1 \cdot s_2 \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ s_2' \end{pmatrix} \cdot (s_2 \cdot K_2' \cdot t_2) \\
\leq & \quad \left\langle \text{By definition of } S, \begin{pmatrix} 0 & t_1 \cdot s_2 \\ 0 & 0 \end{pmatrix} \leq S \right\rangle \\
& s \cdot \begin{pmatrix} 0 \\ G_2 \cdot t_2 \end{pmatrix} + s \cdot S^* \cdot \begin{pmatrix} 0 & t_1 \cdot s_2 \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ G_2 \cdot t_2 \end{pmatrix} + \\
& s \cdot G \cdot S \cdot \begin{pmatrix} 0 \\ s_2' \end{pmatrix} \cdot (s_2 \cdot K_2' \cdot t_2) \\
= & \quad \langle\ \text{Matrix multiplication and definition of } I \text{ in Table 7}\ \rangle \\
& s \cdot \begin{pmatrix} 0 \\ G_2 \cdot t_2 \end{pmatrix} + s \cdot S^* \cdot \begin{pmatrix} t_1 \\ 0 \end{pmatrix} \cdot (s_2 \cdot G_2 \cdot t_2) + s \cdot I \cdot \begin{pmatrix} 0 \\ s_2' \end{pmatrix} \cdot (s_2 \cdot K_2' \cdot t_2)
\end{array}
$$

$$
\begin{array}{ll}
 & \left\langle \text{It is assumed that there is a path from } \alpha \text{ to any node, hence} \right. \\
= & \left. \text{there is one to a predecessor of } \alpha_2, \text{ i.e., } s \cdot S^* \cdot \begin{pmatrix} t_1 \\ 0 \end{pmatrix} = 1 \right\rangle \\
 & s \cdot \begin{pmatrix} 0 \\ G_2 \cdot t_2 \end{pmatrix} \;+\; (s_2 \cdot G_2 \cdot t_2) \;+\; s \cdot I \cdot \begin{pmatrix} 0 \\ s_2' \end{pmatrix} \cdot (s_2 \cdot K_2' \cdot t_2) \\
= & \left\langle s \cdot \begin{pmatrix} 0 \\ G_2 \cdot t_2 \end{pmatrix} = \begin{cases} 0 & \text{if } \alpha \text{ is a node of } S_1 \\ s_2 \cdot G_2 \cdot t_2 & \text{if } \alpha = \alpha_2 \end{cases} \right\rangle \\
 & (s_2 \cdot G_2 \cdot t_2) \;+\; s \cdot I \cdot \begin{pmatrix} 0 \\ s_2' \end{pmatrix} \cdot (s_2 \cdot K_2' \cdot t_2) \\
= & \left\langle \text{Distributivity and } s \cdot I \cdot \begin{pmatrix} 0 \\ s_2' \end{pmatrix} \leq 1 \right\rangle \\
 & (s_2 \cdot G_2 \cdot t_2) \;+\; s \cdot I \cdot \begin{pmatrix} 0 \\ s_2' \end{pmatrix} \cdot s_2 \cdot (K_2' + G_2) \cdot t_2 \\
= & \langle \text{ (14), (12) and set notation } \rangle \\
 & \mathsf{gen}(S_2) \cup (I[\alpha, \alpha_2] - \mathsf{kill}(S_2)) \; .
\end{array}
$$

There is one weakening step $\leq$ in the previous derivation. To show that it can be strengthened to an equality, it suffices to prove

$$s \cdot G \cdot S \cdot \begin{pmatrix} 0 \\ s_2' \end{pmatrix} \cdot (s_2 \cdot K_2' \cdot t_2) \leq s \cdot O \cdot \begin{pmatrix} 0 \\ t_2 \end{pmatrix},$$

since $s \cdot G \cdot S \cdot \begin{pmatrix} 0 \\ s_2' \end{pmatrix} \cdot (s_2 \cdot K_2' \cdot t_2)$ is the only expression that has increased in the weakening step.

$$
\begin{array}{ll}
 & s \cdot G \cdot S \cdot \begin{pmatrix} 0 \\ s_2' \end{pmatrix} \cdot (s_2 \cdot K_2' \cdot t_2) \\
= & \langle \text{ Matrix multiplication } \rangle \\
 & s \cdot G \cdot S \cdot \begin{pmatrix} 0 \\ s_2' \end{pmatrix} \cdot \begin{pmatrix} 0 & s_2 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 \\ 0 & K_2' \end{pmatrix} \cdot \begin{pmatrix} 0 \\ t_2 \end{pmatrix} \\
\leq & \left\langle \text{It is easy to show } \begin{pmatrix} 0 & 0 \\ 0 & K_2' \end{pmatrix} \leq \overline{K}, \text{ and matrix multiplication} \right\rangle \\
 & s \cdot G \cdot S \cdot \begin{pmatrix} 0 & 0 \\ 0 & s_2' \cdot s_2 \end{pmatrix} \cdot \overline{K} \cdot \begin{pmatrix} 0 \\ t_2 \end{pmatrix} \\
\leq & \langle \text{ Since } s_2 \text{ selects the unique node } \alpha_2, \; s_2' \cdot s_2 \leq 1 \; \rangle \\
 & s \cdot G \cdot S \cdot \overline{K} \cdot \begin{pmatrix} 0 \\ t_2 \end{pmatrix} \\
\leq & \langle \text{ (17) } \rangle \\
 & s \cdot O \cdot \begin{pmatrix} 0 \\ t_2 \end{pmatrix}.
\end{array}
$$

7. Related work

Other authors [25–27,13,14,28] use Kleene algebra for the purpose of conducting static analysis of programs.

In [26], Kot and Kozen present an approach to abstract interpretation [29] of programs based on KA. The actual KA used in their approach is the set of $n \times n$ matrices over a KA K of (monotone) semilattice homomorphisms $f : L \to L$. Their approach is a second-order one, where the object of prime interest is not the information about the execution state (as in the standard approaches and the present paper) but the transfer functions that describe how the state is transformed by an instruction. They give in [26] a hybrid algorithm to compute the desired analysis whose worst-case complexity is $O(n \cdot m + m^3)$, where n is the size of the program (the number of instructions) and m is the size of a cutset of the program's CFG. Compared to the standard iterative algorithm [16,4], which is $O(n \cdot d)$, d being

the depth of the semilattice L, their algorithm may give an improvement when m is small compared to n. They also describe a concrete implementation of this method in the context of Java bytecode verification in [27].

In [14], Kozen proposes a general framework for the static analysis of programs based on KA with tests. He shows how KA with tests can be used to statically verify compliance with safety policies specified by security automata [30] and he proves soundness and completeness over relational interpretations.

In [28], Kozen and Patron use KA with tests to verify a wide assortment of common compiler optimizations, including dead code elimination, common subexpression elimination, copy propagation, loop hoisting and induction variable elimination. For each of the considered optimizations, they give a formal equational proof of the correctness of the optimizing transformation.

8. Conclusion

We have shown how four instances of gen/kill analysis can be described using Kleene algebra. This has been done for a CFG-like and an LTS-like representation of programs (using matrices). The result of this exercise is a very concise and very readable set of equations characterizing the four analyses.

We have in fact used relations for the formalization, so that the framework of relation algebra with transitive closure [19,20] could have been used instead of that of KA. Note however that converse has been used only to explain the forward/backward dualities, but is used nowhere in the calculations. We prefer KA or Boolean KA because the results have wider applicability. It is reasonable to expect to find examples where the equations of Tables 7 and 8 could be used for something else than relations. Also, we hope to connect the Kleene formulation of the gen/kill analyses with representations of programs where KA is already used. For instance, KA is already employed to analyze sequences of abstract program actions for security properties [14]. Instead of keeping only the name of an action (instruction), it would be possible to construct a triple (name, gen, kill) giving information about the name assigned to the instruction and what it generates and kills. Such triples can be elements of a KA by applying KA operations componentwise. Our plan is to determine whether it is possible to prove stronger security properties in the framework of KA, given that more information is available.

We plan to investigate other types of program analysis to see if the techniques presented in this paper could apply to them. We would also like to describe the analyses of this paper using a KA-based deductive approach in the style of [28].

Acknowledgements

The authors thank Bernhard Möller and the MPC and SCP referees for thoughtful comments, some challenging questions and additional pointers to the literature.

References

[1] A.V. Aho, J.D. Ullman, Principles of Compilers Design, Addison-Wesley, 1977.
[2] A.V. Aho, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques and Tools, Addison-Wesley, 1986.
[3] C.N. Fischer, R.J. LeBlanc Jr., Crafting a Compiler, Benjamin/Cummings Publishing Company, Menlo Park, CA, 1988.
[4] F. Nielson, H.R. Nielson, C. Hankin, Principles of Program Analysis, Springer, 1999.
[5] C.M. Overstreet, R. Cherinka, R. Sparks, Using bidirectional data flow analysis to support software reuse, Tech. Rep. TR-94-09, Old Dominion University, Computer Science Department, June 1994.
[6] L. Moonen, Data flow analysis for reverse engineering, Master's Thesis, Programming Research Group, University of Amsterdam, 1996.
[7] R.W. Lo, K.N. Levitt, R.A. Olsson, MCF: A malicious code filter, Computers and Security 14 (6) (1995) 541–566.
[8] J. Bergeron, M. Debbabi, J. Desharnais, M.M. Erhioui, Y. Lavoie, N. Tawbi, Static detection of malicious code in executable programs, in: 1st Symposium on Requirements Engineering for Information Security, Indianapolis, IN, 2001.
[9] J.H. Conway, Regular Algebra and Finite Machines, Chapman and Hall, London, 1971.
[10] J. Desharnais, B. Möller, G. Struth, Kleene algebra with domain, Tech. Rep. 2003-7, Institut für Informatik, Universität Augsburg, D-86135 Augsburg, May 2003.
[11] J. Desharnais, B. Möller, G. Struth, Modal Kleene algebra and applications — A survey, Tech. Rep. DIUL-RR-0401, Département d'informatique et de génie logiciel, Université Laval, D-86135 Augsburg, March 2004.
[12] D. Kozen, A completeness theorem for Kleene algebras and the algebra of regular events, Information and Computation 110 (2) (1994) 366–390.
[13] D. Kozen, Kleene algebra with tests, ACM Transactions on Programming Languages and Systems 19 (3) (1997) 427–443.

[14] D. Kozen, Kleene algebra with tests and the static analysis of programs, Tech. Rep. 2003-1915, Department of Computer Science, Cornell University, November 2003.
[15] J.B. Kam, J.D. Ullman, Monotone data flow analysis frameworks, Acta Informatica 7 (1977) 309–317.
[16] G. Kildall, A unified approach to global program optimization, in: Proceedings of the 1st Annual ACM Symposium on Principles of Programming Languages, 1973, pp. 194–206.
[17] E. Clarke, E. Emerson, Synthesis of synchronisation skeletons for branching time temporal logic, Logics of Programs 131 (71) (1981) 52–71.
[18] B. Möller, Derivation of graph and pointer algorithms, in: B. Möller, H.A. Partsch, S.A. Schuman (Eds.), Formal Program Development, in: Lecture Notes in Computer Science, vol. 755, Springer, Berlin, 1993, pp. 123–160.
[19] K.C. Ng, Relation algebras with transitive closure, Ph.D. Thesis, University of California, Berkeley, 1984.
[20] K.C. Ng, A. Tarski, Relation algebras with transitive closure, Notices of the American Mathematical Society 24 (1977) A29–A30.
[21] J. Desharnais, Kleene algebra with relations, in: R. Berghammer, B. Möller, G. Struth (Eds.), Relational and Kleene-Algebraic Methods, in: Lecture Notes in Computer Science, vol. 3051, Springer, 2004, pp. 8–20.
[22] D. Kozen, Typed Kleene algebra, Tech. Rep. 98-1669, Computer Science Department, Cornell University, March 1998.
[23] G. Schmidt, C. Hattensperger, M. Winter, Heterogeneous relation algebra, in: C. Brink, W. Kahl, G. Schmidt (Eds.), Relational Methods in Computer Science, Springer, 1997, pp. 39–53.
[24] G. Schmidt, T. Ströhlein, Relations and Graphs, in: EATCS Monographs in Computer Science, Springer, Berlin, 1993.
[25] A. Barth, D. Kozen, Equational verification of cache blocking in LU decomposition using Kleene algebra with tests, Tech. Rep. 2002-1865, Computer Science Department, Cornell University, June 2002.
[26] L. Kot, D. Kozen, Second-order abstract interpretation via Kleene algebra, Tech. Rep. 2004-1971, Computer Science Department, Cornell University, December 2004.
[27] L. Kot, D. Kozen, Kleene algebra and bytecode verification, in: Proc. 1st Workshop Bytecode Semantics, Verification, Analysis, and Transformation, Bytecode'05, Edinburgh, in: Electronic Notes in Theoretical Computer Science, vol. 141, 2005, pp. 221–236.
[28] D. Kozen, M.-C. Patron, Certification of compiler optimizations using Kleene algebra with tests, in: J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L.M. Pereira, Y. Sagiv, P.J. Stuckey (Eds.), Proc. 1st Int. Conf. on Computational Logic, CL2000, London, in: Lecture Notes in Artificial Intelligence, vol. 1861, Springer, London, 2000, pp. 568–582.
[29] P. Cousot, R. Cousot, Abstract interpretation: A unified lattice model for static analysis by construction or approximation of fixpoints, in: Proceedings of POPL'77, ACM Press, Los Angeles, California, 1977, pp. 238–252.
[30] F.B. Schneider, Enforceable security policies, ACM Transactions on Information and System Security 3 (1) (2000) 30–50.