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).

D. Kozen (Ed.): MPC 2004, LNCS 3125, pp. 110–128, 2004.

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.

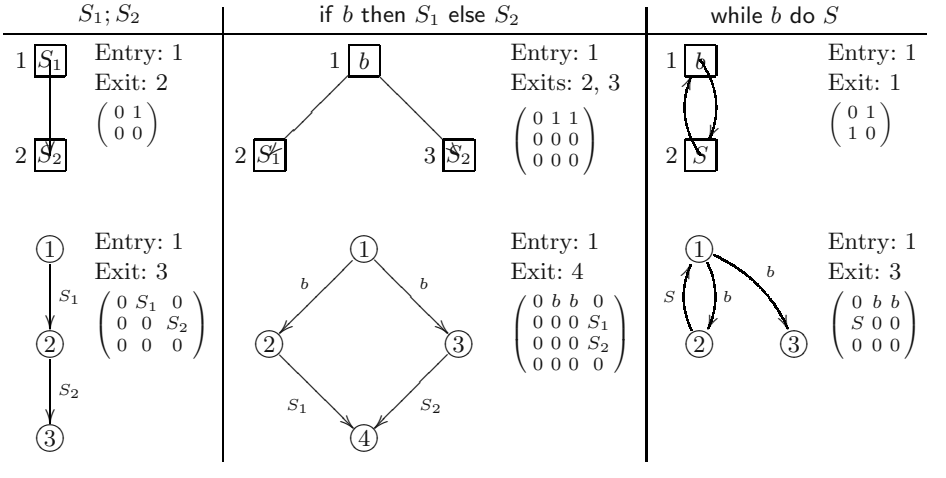


Fig. 1. CFGs and LTSs for compound statements

The four instances of gen/kill analysis 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 [13], 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, and possibly also information about the environment of the program if it appears inside a larger program. 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 analysis. 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.

RD The set D is a set of *definitions*. A definition is a pair (x, l), where l is the label of an assignment $x := E$. The assignment $x := E$ at label l generates the definition (x, l) and kills all other definitions of x. 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 1. Gen/kill values for atomic statements and tests. The symbol l denotes a label and the symbol b a test

	$l : x := E$		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	$\mathsf{Exp}(E)$	$\{E' \in D \mid x \in \mathsf{Var}(E')\} - \mathsf{Exp}(E)$	$\emptyset$	$\emptyset$	$\mathsf{Exp}(b)$	$\emptyset$

Table 1 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 2). The forward/backward duality is apparent when comparing the values of $\mathsf{gen}(S_1; S_2)$ 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 gen).

Finally, Table 3 shows how $\mathsf{out}(S)$ and $\mathsf{in}(S)$ can be recursively calculated. 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 along 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 $\{(x, 1), (x, 3), (y, 4)\}$. Assume that this program is embedded in a larger program that contains the definitions $(x, 5)$ and $(y, 6)\}$ (they may appear before label 1, even if they have a larger number as label) and that these definitions reach the entry point of the example program. Using Table 1 for RD, we get the following gen/kill values for the atomic statements, where S_l denotes the atomic statement at label l:

Table 2. 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	$\mathsf{gen}(S_1)$	$\emptyset$
LV	$\mathsf{gen}(b) \cup \mathsf{gen}(S_1)$	$\emptyset$
AE, VBE	$\mathsf{gen}(b)$	$\mathsf{kill}(S_1) - \mathsf{gen}(b)$

Table 3. Linking in and out ("imm." abbreviates "immediately")

	$\mathsf{in}(S)$	$\mathsf{out}(S)$
RD	$\bigcup(S' \mid S' \text{ imm. precedes } S : \mathsf{out}(S'))$	$\mathsf{gen}(S) \cup (\mathsf{in}(S) - \mathsf{kill}(S))$
LV	$\mathsf{gen}(S) \cup (\mathsf{out}(S) - \mathsf{kill}(S))$	$\bigcup(S' \mid S' \text{ imm. follows } S : \mathsf{in}(S'))$
AE	$\bigcap(S' \mid S' \text{ imm. precedes } S : \mathsf{out}(S'))$	$\mathsf{gen}(S) \cup (\mathsf{in}(S) - \mathsf{kill}(S))$
VBE	$\mathsf{gen}(S) \cup (\mathsf{out}(S) - \mathsf{kill}(S))$	$\bigcap(S' \mid S' \text{ imm. follows } S : \mathsf{in}(S'))$

$$\begin{array}{ll} \mathsf{gen}(S_1) = \{(x,1)\}, & \mathsf{kill}(S_1) = \{(x,3),(x,5)\}, \\ \mathsf{gen}(S_2) = \emptyset, & \mathsf{kill}(S_2) = \emptyset, \\ \mathsf{gen}(S_3) = \{(x,3)\}, & \mathsf{kill}(S_3) = \{(x,1),(x,5)\}, \\ \mathsf{gen}(S_4) = \{(y,4)\}, & \mathsf{kill}(S_4) = \{(y,6)\}. \end{array} \tag{1}$$

Using Table 2 for RD, we get

$$\begin{array}{lcl} \mathsf{gen}(S_1; \mathsf{if}\ S_2\ \mathsf{then}\ S_3\ \mathsf{else}\ S_4) & = & \{(x,3),(y,4)\} \cup (\{(x,1)\} - \emptyset) \\ & = & \{(x,1),(x,3),(y,4)\}, \\ \mathsf{kill}(S_1; \mathsf{if}\ S_2\ \mathsf{then}\ S_3\ \mathsf{else}\ S_4) & = & \emptyset \cup (\{(x,3),(x,5)\} - \{(x,3),(y,4)\}) \\ & = & \{(x,5)\}. \end{array}$$

Finally, Table 3 for RD yields

$$\begin{array}{lcl} \mathsf{in}(S_1; \mathsf{if}\ S_2\ \mathsf{then}\ S_3\ \mathsf{else}\ S_4) & = & \{(x,5),(y,6)\}, \\ \mathsf{out}(S_1; \mathsf{if}\ S_2\ \mathsf{then}\ S_3\ \mathsf{else}\ S_4) & = & \{(x,1),(x,3),(y,4)\} \cup \\ & & (\{(x,5),(y,6)\} - \{(x,5)\}) \\ & = & \{(x,1),(x,3),(y,4),(y,6)\}. \end{array}$$

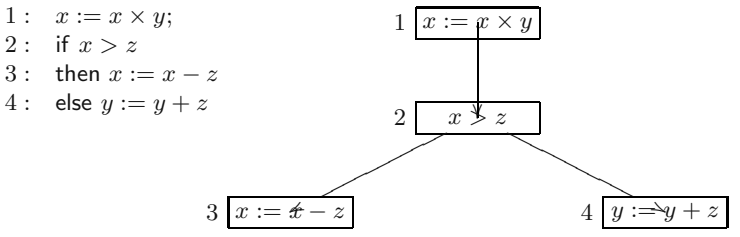


Fig. 2. CFG for running example

3 Kleene Algebra with Tests

In this section, we first introduce Kleene algebra [6, 9] and a specialization of it, namely Kleene algebra with tests [10]. 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) [9] 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:*

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

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

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

A Kleene algebra with tests *[10] 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.

Models of KA with tests include algebras of languages over an alphabet, algebras of path sets in a directed graph [14], algebras of relations over a set and abstract relation algebras with transitive closure [15, 16].

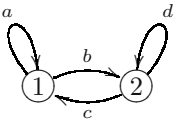
A very simple model of KA with tests is obtained by taking K to be the powerset of some set D and defining, for every $a, b \subseteq D$,

$$0 \stackrel{\text{def}}{=} \emptyset, \quad 1 \stackrel{\text{def}}{=} D, \quad a^* \stackrel{\text{def}}{=} D, \quad \neg a \stackrel{\text{def}}{=} \overline{a}, \quad a + b \stackrel{\text{def}}{=} a \cup b, \quad a \cdot b \stackrel{\text{def}}{=} a \cap b. \qquad (2)$$

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 $\mathbf{A}[i, j]$ refers to the entry in row i and column j of $\mathbf{A}$.

1. $\mathbf{0}$: matrix whose entries are all 0, i.e., $\mathbf{0}[i,j] = 0$,
2. $\mathbf{1}$: identity matrix (square), i.e., $\mathbf{1}[i,j] = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j, \end{cases}$
3. $(\mathbf{A} + \mathbf{B})[i,j] \stackrel{\text{def}}{=} \mathbf{A}[i,j] + \mathbf{B}[i,j]$,
4. $(\mathbf{A} \cdot \mathbf{B})[i,j] \stackrel{\text{def}}{=} \sum(k \mid : \mathbf{A}[i,k] \cdot \mathbf{B}[k,j])$,
5. The Kleene star of a square matrix is defined recursively [9]. If $\mathbf{A} = (\, a \,)$, for some $a \in K$, then $\mathbf{A}^* \stackrel{\text{def}}{=} (\, a^* \,)$. If

$$\mathbf{A} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \qquad \text{(with graphic representation } \quad \text{)} \ ,$$



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

$$\mathbf{A}^* \stackrel{\text{def}}{=} \begin{pmatrix} f^* & f^* \cdot b \cdot d^* \\ d^* \cdot c \cdot f^* & d^* + d^* \cdot c \cdot f^* \cdot b \cdot d^* \end{pmatrix} \ , \tag{3}$$

where $f = a + b \cdot d^* \cdot c$; the automaton corresponding to $\mathbf{A}$ helps understand that f^* corresponds to paths from state 1 to state 1. If $\mathbf{A}$ is a larger matrix, it is decomposed as a 2×2 matrix of submatrices: $\mathbf{A} = \begin{pmatrix} \mathbf{B} & \mathbf{C} \\ \mathbf{D} & \mathbf{E} \end{pmatrix}$, where $\mathbf{B}$ and $\mathbf{E}$ are square and nonempty. Then $\mathbf{A}^*$ is calculated recursively using (3). For our simple application below, $\mathbf{A}^* = \sum(n \mid n \geq 0 : \mathbf{A}^n)$, where $\mathbf{A}^0 = \mathbf{1}$ and $\mathbf{A}^{n+1} = \mathbf{A} \cdot \mathbf{A}^n$.

By setting up an appropriate type discipline, one can define *heterogeneous Kleene algebras* as is done for heterogeneous relation algebras [17–19]. 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 appropriate size constraints.

In Sect. 4, we will only use matrices whose entries are all tests. Such matrices are relations [20]; 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 $\mathbf{S}$ to represent the structure of programs and consider only matrices below the reflexive transitive closure $\mathbf{S}^*$ of $\mathbf{S}$. Complementation can then be defined as complementation relative to $\mathbf{S}^*$:

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

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

As a final remark in this section, we point out that a (square) matrix $\mathbf{T}$ is a test iff it is a diagonal matrix whose diagonal contains tests (this implies $\mathbf{T} \leq \mathbf{1}$). For instance, if t_1, t_2 and t_3 are tests,

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

4 Gen/Kill Analysis with KA

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

4.1 RD Analysis

We first explain how the data and programs to analyze are modelled. 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

$$D \stackrel{\text{def}}{=} \{(x,1),(x,3),(y,4),(x,5),(y,6)\} \ . \tag{4}$$

We consider the powerset of D as a Kleene algebra, as explained in Sect. 3 (see (2)).

The input to the analysis consists of three matrices $\mathbf{S}, \mathbf{g}$ and $\mathbf{k}$ representing respectively the structure of the program, what is generated and what is killed at each label. Here is an abstract definition of these matrices, where $g_i \subseteq D$ and $k_i \subseteq D$, for $i = 1, \ldots, 4$.

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

Recall that $1 = D$ (see (2)). Using the values already found in (1), we get the following as concrete instantiations for the example program (with g_i for $\mathsf{gen}(S_i)$ and k_i for $\mathsf{kill}(S_i)$):

$$\begin{array}{llll} g_1 \stackrel{\text{def}}{=} \{(x,1)\}, & g_2 \stackrel{\text{def}}{=} \emptyset, & g_3 \stackrel{\text{def}}{=} \{(x,3)\}, & g_4 \stackrel{\text{def}}{=} \{(y,4)\}, \\ k_1 \stackrel{\text{def}}{=} \{(x,3),(x,5)\}, & k_2 \stackrel{\text{def}}{=} \emptyset, & k_3 \stackrel{\text{def}}{=} \{(x,1),(x,5)\}, & k_4 \stackrel{\text{def}}{=} \{(y,6)\}. \end{array} \tag{5}$$

Note that $\mathbf{g}$ and $\mathbf{k}$ are tests. The entries in their diagonal represent what is generated or killed by the atomic instruction at each node. Table 1 imposes the condition $\mathsf{gen}(S) \subseteq \overline{\mathsf{kill}(S)}$ for any atomic statement S; this translates to $\mathbf{g} \leq \neg\mathbf{k}$ for the matrices given above.

Table 4 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 from an entry point (S^*), generate that something (g), then move to an exit point 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 4. RD existential ("may") gen/kill parameters for CFGs. Complementation is relative to S^*

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

3. O: To output something, generate it or, if it comes from the environment (the test i), do not kill it. Note how the expression for O is close to that for out given in Table 3, namely $\mathsf{gen}(S) \cup (\mathsf{in}(S) - \mathsf{kill}(S))$.

We use the table to calculate $\mathbf{G}$ and $\overline{\mathbf{K}}$ for RD [1]. It is a simple task to verify the following result (one has to use $\mathbf{g} \leq \neg\mathbf{k}$, 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.

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

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

Consider the entry $\mathbf{G}[1,3]$, 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 path, this means 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, $\overline{\mathbf{K}}[1,2]$ 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 either of labels 1 and 2.

To compare these results with those of the classical approach of Sect. 2, it suffices to collect from $\mathbf{G}$ the data generated and from $\overline{\mathbf{K}}$ the data not killed between the entry point of the program (label 1) and its exit points (labels 3 and 4). This can always be done by means of a row vector $\mathbf{s}$ selecting the entry points and a column vector $\mathbf{t}$ selecting the exit points. For our program,

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

so that

$$\mathsf{gen} \;=\; \mathbf{s} \cdot \mathbf{G} \cdot \mathbf{t} \;=\; g_3 + g_4 + (g_2 + g_1 \cdot \neg k_2) \cdot (\neg k_3 + \neg k_4) \tag{6}$$

[1] We use bold letters for matrices. Because the concepts of Table 4 may apply to other kinds of KAs, the variables in the table are typeset in the usual mathematical font.

and

$$\neg\mathsf{kill} \;=\; \mathbf{s}\cdot\overline{\mathbf{K}}\cdot\mathbf{t} \;=\; g_3 + g_4 + (g_2 + \neg k_1\cdot\neg k_2)\cdot(\neg k_3 + \neg k_4) \;\;.$$

Negating yields

$$\mathsf{kill} \;=\; k_3\cdot k_4 + k_2\cdot\neg g_3\cdot\neg g_4 + k_1\cdot\neg g_2\cdot\neg g_3\cdot\neg g_4 \;\;.$$

This is easily read and understood by looking at the CFG. Using the concrete values in (4) and (5) provides $\mathsf{gen} = \{(x,1),(x,3),(y,4)\}$ and $\mathsf{kill} = \{(x,5)\}$, just like in Sect. 2.

One can get $\mathbf{K}$ from the above matrix $\overline{\mathbf{K}}$ by complementing, but complementation must be done relatively to $\mathbf{S}^*$; 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

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

One might think that $\mathsf{kill} = \mathbf{s}\cdot\mathbf{K}\cdot\mathbf{t}$, but this is not the case. It is easy to see that $\mathbf{s}\cdot\mathbf{K}\cdot\mathbf{t} = \mathbf{K}[1,3] + \mathbf{K}[1,4]$, whereas the value of kill that we obtained above is $\neg(\overline{\mathbf{K}}[1,3] + \overline{\mathbf{K}}[1,4]) = \mathbf{K}[1,3]\cdot\mathbf{K}[1,4]$, and the latter is the right value. The reason for this behavior is that for RD, *not killing*, like *generating*, is existential, 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 $\mathbf{s}\cdot\mathbf{K}\cdot\mathbf{t}$ is to join all entries of the form $\mathbf{K}$[entry point, exit point] ($\mathbf{K}[1,3] + \mathbf{K}[1,4]$ for the example). Note that if the program has only one entry and one exit point, one may use either $\mathbf{s}\cdot\mathbf{K}\cdot\mathbf{t}$ or $\neg(\mathbf{s}\cdot\overline{\mathbf{K}}\cdot\mathbf{t})$; the equivalence follows from the fact that $\mathbf{s}$ is then a total function, while $\mathbf{t}$ is injective and surjective.

There remains one value to find for our example, that of $\mathbf{O}$. In Table 4, the equation for $\mathbf{O}$ is $O = G + i\cdot\overline{K}$. The symbol i denotes a test that characterizes the data that come from the environment of the program (a larger program containing it). For our example, i has the form

$$\mathbf{i} = \begin{pmatrix} i_1 & 0 & 0 & 0 \\ 0 & i_2 & 0 & 0 \\ 0 & 0 & i_3 & 0 \\ 0 & 0 & 0 & i_4 \end{pmatrix} \;\;.$$

Using the same $\mathbf{s}$ and $\mathbf{t}$ as above, we calculate $\mathbf{s}\cdot\mathbf{O}\cdot\mathbf{t}$, which is the information that gets out at the exit points as a function of what gets in at the entry point. We get

$$\begin{aligned} \mathbf{s}\cdot\mathbf{O}\cdot\mathbf{t} \;=\; \mathbf{s}\cdot(\mathbf{G} + \mathbf{i}\cdot\overline{\mathbf{K}})\cdot\mathbf{t} \;=\; & g_3 + g_4 + \\ & g_2\cdot(\neg k_3 + \neg k_4) + \\ & g_1\cdot\neg k_2\cdot(\neg k_3 + \neg k_4) + \\ & i_1\cdot\neg k_1\cdot\neg k_2\cdot(\neg k_3 + \neg k_4) \;\;. \end{aligned}$$

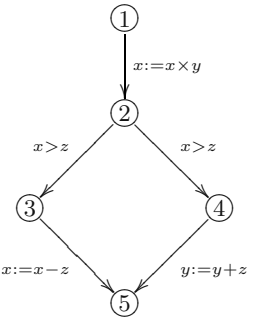
Thus, what gets out at exit labels 3 and 4 is what is generated at labels 3 and 4, plus what is generated at labels 1 and 2 and not killed after, plus what comes from the environment at label 1 and is not killed after. With the instantiations given in (4) and (5),

$$\mathbf{s}\cdot\mathbf{O}\cdot\mathbf{t} \;=\; \{(x,1),(x,3),(y,4)\} + i_1\cdot\{(x,1),(y,4),(y,6)\} \;.$$

If – as in Sect. 2 – we assume that the data coming from the environment at label 1 is $i_1 \stackrel{\text{def}}{=} \{(x,5),(y,6)\}$, then $\mathbf{s}\cdot\mathbf{O}\cdot\mathbf{t} = \{(x,1),(x,3),(y,4),(y,6)\}$ – as in Sect. 2.

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 [19]. Our example program and its LTS graph are given in Fig. 3. As mentioned in Sect. 2, we do not distinguish the two possible run-time results of the test $x > z$, since this does not change any of the four analyses.

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



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

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

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

Fig. 3. LTS for running example

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 (4) and (5) still apply.

Table 5 can then be used in the same manner as Table 4 to derive G, K and O. In this table, as usual, a^+ denotes $a\cdot a^*$. The variable i still denotes a test. The operator $\tilde{}$ denotes complementation with respect to S^+, so that $K = \overline{\tilde{K}} \sqcap S^+$. For CFGs, complementation is done with respect to S^*, because the instructions are on the nodes. For LTSs, the instructions are on the arcs, so that no killing or generation can occur at a node, unless it occurs via a nonnull circular path; this explains why complementation is done with respect to S^+.

Table 5. RD existential ("may") gen/kill parameters for LTSs. The operator $\tilde{}$ is complementation relative to S^+

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

The calculation of $\mathbf{G}$ gives

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

Using

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

we obtain for $\mathsf{gen} = \mathbf{s} \cdot \mathbf{G} \cdot \mathbf{t}$ the same result as for the CFG (see (6)).

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 6). The following remarks are in order.

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, except that o is used instead of i to denote what comes from the environment and I is used instead of O for the result. This was done simply to have a more exact correspondence with Table 3. Although o is input data, the letter o is used because the data is provided at the exit points; similarly, the letter I is used for the output data because it is associated with the entry points.
2. For all atomic and compound statements and all equations of Table 6, one can do the same kind of abstract comparison with the results given by Tables 1 and 2 as we have done for the example in Sect. 4.1. The results are the same.
3. The forward/backward and may/must dualities are apparent in the tables of Sect. 2, but they are much more visible and clear here.
   (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: $i \leftrightarrow o$ and $O \leftrightarrow I$. One can also use the relational converse operator $\breve{}$; then, for LV, $G = (\neg k \cdot S)^* \cdot g \cdot S^* =$

Table 6. Existential ("may") gen/kill parameters for CFGs. Complementation is relative to S^*

	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	$G + i \cdot \overline{K}$			
I		$G + \overline{K} \cdot o$		
$\overline{O}$			$K + \neg i \cdot \overline{G}$	
$\overline{I}$				$K + \overline{G} \cdot \neg o$

$(S^{\smile *} \cdot g^{\smile} \cdot (S^{\smile} \cdot \neg k^{\smile})^*)^{\smile}$. The same can be done for $\overline{K}$ and I. Thus, to make an LV analysis, one can switch i, o, 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 correspondences RD $\leftrightarrow$ AE and LV $\leftrightarrow$ VBE are much deeper and can in fact be obtained simply by switching gen and kill, and complementing in and out: $g \leftrightarrow k$, $G \leftrightarrow K$, $i \leftrightarrow \neg i$, $o \leftrightarrow \neg o$, $I \leftrightarrow \overline{I}$, $O \leftrightarrow \overline{O}$.

These dualities mean that only one kind of analysis is really necessary, since the other three can be obtained by simple substitutions and simple additional operations (converse and complementation).

4. All nonempty entries in Table 6 correspond to existential cases; collecting data with entry and exit vectors as we have done in Sect. 4.1 should be done with the parameters as given in Table 6 and not on their negation (unless there is only one entry and one exit point).
5. Table 6 can be applied to programs with goto statements to fixed labels without any additional machinery.

The equations for LTSs are given in Table 7. Similar comments can be made about this table as for Table 6.

The formulae in Tables 6 and 7 are obviously related, but it is interesting to see how the connection can be described formally and this is what we now do. We will also show the following: If P denotes any of the parameters in the left column of either Table 6 or Table 7 and if s and t denote the entry and exit vectors appropriate for the representation (CFG or LTS), then the value of $s \cdot P \cdot t$ is the same for both approaches, provided only the existential equations given in the tables are used (no complementation before merging the data with s and t). Note that the main goal of the analyses is to calculate $s \cdot O \cdot t$ or $s \cdot I \cdot t$.

Table 7. Existential ("may") gen/kill parameters for LTSs. The operator $\tilde{}$ is complementation relative to S^+

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

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 a LTS. The following matrices $\mathbf{S}$ and $\mathbf{S}_{\mathrm{L}}$ represent the structure of the CFG of Fig. 2 and that of the LTS of Fig. 3, respectively.

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

The matrix $\mathbf{S}_{\mathrm{L}}$ is structured as a matrix of four submatrices, one of which is the CFG matrix $\mathbf{S}$ and another is the column vector $\mathbf{t}$ that was used in (6) to select the exit nodes of the CFG. The rôle of this vector in $\mathbf{S}_{\mathrm{L}}$ is to add links from the exit nodes of the CFG to the new node corresponding to column 5.

Now consider the matrices of Sect. 4.1. The CFG matrix $\mathbf{g}$ can be converted to the LTS matrix $\mathbf{g}$, here called $\mathbf{g}_{\mathrm{L}}$, as follows.

$$\mathbf{g}_{\mathrm{L}} = \begin{pmatrix} \mathbf{g} \cdot \mathbf{S} & \mathbf{g} \cdot \mathbf{t} \\ \mathbf{0} & \mathbf{0} \end{pmatrix} = \begin{pmatrix} \mathbf{g} & \mathbf{x} \\ \mathbf{0} & \mathbf{y} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{S} & \mathbf{t} \\ \mathbf{0} & \mathbf{0} \end{pmatrix} = \begin{pmatrix} \mathbf{g} & \mathbf{x} \\ \mathbf{0} & \mathbf{y} \end{pmatrix} \cdot \mathbf{S}_{\mathrm{L}}$$

The value of the submatrices $\mathbf{x}$ and $\mathbf{y}$ does not matter, since these disappear in the result of the composition. One can use the concrete values given in Sect. 4.1 and check that indeed in that case $\mathbf{g}_{\mathrm{L}} = \begin{pmatrix} \mathbf{g} \cdot \mathbf{S} & \mathbf{g} \cdot \mathbf{t} \\ \mathbf{0} & \mathbf{0} \end{pmatrix}$. The same holds for $\mathbf{G}$ and $\overline{\mathbf{K}}$. The matrix $\begin{pmatrix} \mathbf{g} & \mathbf{x} \\ \mathbf{0} & \mathbf{y} \end{pmatrix}$ is an embedding of the CFG $\mathbf{g}$ in a larger graph with

an additional node. Composition with $\mathbf{S}_{\mathrm{L}}$ "pushes" the information provided by $\mathbf{g}$ on the appropriate arrows of $\mathbf{g}_{\mathrm{L}}$.

We now abstract from the matrix context. We assume an heterogeneous Boolean KA such that, given correctly typed elements a, b, c, d, it is possible to form matrices like $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$. To distinguish expressions related to CFGs and LTSs, we add a subscript $_{\mathrm{L}}$ to variables in the latter expressions.

We will show that the RD CFG expressions given in Table 6 can be transformed into the corresponding LTS expressions given in Table 7. An analogous treatment can be done for the other analyses.

We begin with G, whose expression is $S^* \cdot g \cdot (S \cdot \neg k)^*$, and show that it transforms to $S_{\mathrm{L}}^* \cdot g_{\mathrm{L}} \cdot (\overline{k_{\mathrm{L}}} \sqcap S_{\mathrm{L}})^*$. We first note the following two properties:

$$\begin{pmatrix} a & ? \\ 0 & ? \end{pmatrix} \cdot \begin{pmatrix} b & ? \\ 0 & ? \end{pmatrix} = \begin{pmatrix} a \cdot b & ? \\ 0 & ? \end{pmatrix} \qquad \text{and} \qquad \begin{pmatrix} a & ? \\ 0 & ? \end{pmatrix}^* = \begin{pmatrix} a^* & ? \\ 0 & ? \end{pmatrix} \ , \tag{7}$$

where "?" means that the exact value is not important for our purpose (we use the same convention below). Now let $f(a) \stackrel{\text{def}}{=} \begin{pmatrix} a & t \\ 0 & 0 \end{pmatrix}$. We will use the assumptions $S_{\mathrm{L}} = f(S)$, $g_{\mathrm{L}} = f(g) \cdot f(S)$, $k_{\mathrm{L}} = f(k) \cdot f(S)$ and $G_{\mathrm{L}} = \begin{pmatrix} G \cdot S & G \cdot t \\ 0 & 0 \end{pmatrix}$, which hold for matrices, as noted above. Before giving the main derivation, we prove the auxiliary result

$$f(\neg k) \cdot f(S) = \overline{k_{\mathrm{L}}} \sqcap S_{\mathrm{L}} \ . \tag{8}$$

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

$$
\begin{array}{ll}
= & \langle \text{ Assumptions } \rangle \\
& \overline{k_{\mathrm{L}}} \sqcap S_{\mathrm{L}}
\end{array}
$$

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

$$
\begin{array}{ll}
& G_{\mathrm{L}} \\
= & \langle \text{ Assumption } \rangle \\
& \begin{pmatrix} G \cdot S & G \cdot t \\ 0 & 0 \end{pmatrix} \\
= & \langle \text{ Matrix composition } \rangle \\
& \begin{pmatrix} G & ? \\ 0 & ? \end{pmatrix} \cdot \begin{pmatrix} S & t \\ 0 & 0 \end{pmatrix} \\
= & \langle \text{ Expression for } G \text{ and definition of } f \rangle \\
& \begin{pmatrix} S^* \cdot g \cdot (S \cdot \neg k)^* & ? \\ 0 & ? \end{pmatrix} \cdot f(S) \\
= & \langle \text{ Definition of } f \text{ and induction using (7) } \rangle \\
& (f(S))^* \cdot f(g) \cdot (f(S) \cdot f(\neg k))^* \cdot f(S) \\
= & \langle \text{ KA sliding rule: } (a \cdot b)^* \cdot a = a \cdot (b \cdot a)^* \rangle \\
& (f(S))^* \cdot f(g) \cdot f(S) \cdot (f(\neg k) \cdot f(S))^* \\
= & \langle \text{ Assumptions and (8) } \rangle \\
& S_{\mathrm{L}}^* \cdot g_{\mathrm{L}} \cdot (\overline{k_{\mathrm{L}}} \sqcap S_{\mathrm{L}})^*
\end{array}
$$

The transformation of the CFG subexpression $\neg k \cdot (S \cdot \neg k)^*$ (appearing in the definition of $\overline{K}$ in Table 6) is done in a similar fashion, except that the last steps are

$$
\begin{array}{ll}
& f(\neg k) \cdot (f(S) \cdot f(\neg k))^* \cdot f(S) \\
= & \langle \text{ KA sliding rule: } (a \cdot b)^* \cdot a = a \cdot (b \cdot a)^* \rangle \\
& f(\neg k) \cdot f(S) \cdot (f(\neg k) \cdot f(S))^* \\
= & \langle \ a \cdot a^* = a^+ \text{ and (8) } \rangle \\
& (\overline{k_{\mathrm{L}}} \sqcap S_{\mathrm{L}})^+
\end{array}
$$

What remains to establish is the correspondence between i for CFGs and i_{L} for LTSs. Since we want i_{L} to be a test just like i, we cannot take $i_{\mathrm{L}} = f(i) \cdot f(S)$ like for the other matrices. It turns out that $i_{\mathrm{L}} \stackrel{\text{def}}{=} \begin{pmatrix} i & 0 \\ 0 & 0 \end{pmatrix}$ is convenient and is indeed what we would choose using intuition, because it does not make sense to feed information to the additional exit node, since it is past all the instructions. One then has

$$
O_{\mathrm{L}} = \begin{pmatrix} (G + i \cdot \overline{K}) \cdot S & (G + i \cdot \overline{K}) \cdot t \\ 0 & 0 \end{pmatrix} = G_{\mathrm{L}} + \begin{pmatrix} i & 0 \\ 0 & 0 \end{pmatrix} \cdot (\overline{K})_{\mathrm{L}} = G_{\mathrm{L}} + i_{\mathrm{L}} \cdot \tilde{K_{\mathrm{L}}} \ .
$$

Finally, we show that the calculation of the information along paths between entry and exit nodes gives the same result for CFGs and LTSs. For CFGs, this information is obtained by calculating $s \cdot P \cdot t$, where s is the vector of entry nodes, t is the vector of exit nodes and P is any of the existential parameters $(G, \overline{K}, O, \ldots)$, depending on the analysis. For LTSs, the corresponding expression is $s_{\mathrm{L}} \cdot P_{\mathrm{L}} \cdot t_{\mathrm{L}}$. As above, we assume $P_{\mathrm{L}} = \begin{pmatrix} P \cdot S & P \cdot t \\ 0 & 0 \end{pmatrix}$, and $s_{\mathrm{L}} = \begin{pmatrix} s & 0 \end{pmatrix}$ 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. We then have

$$s_{\mathrm{L}} \cdot P_{\mathrm{L}} \cdot t_{\mathrm{L}} = \begin{pmatrix} s & 0 \end{pmatrix} \cdot \begin{pmatrix} P \cdot S & P \cdot t \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} s & 0 \end{pmatrix} \cdot \begin{pmatrix} P \cdot t \\ 0 \end{pmatrix} = s \cdot P \cdot t \ ,$$

so that using either a CFG or an LTS gives the same result.

Note that no property of s or t has been used in the explanation of the transformation from CFGs to LTSs.

5 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. This has revealed the symmetries between the analyses much more clearly than the classical approach.

We have in fact used relations for the formalization, so that the framework of relation algebra with transitive closure [15, 16] could have been used instead of that of Kleene algebra. Note however that converse has been used only to explain the forward/backward dualities, but is used nowhere in the calculations. We prefer Kleene algebra or Boolean Kleene algebra because the results have wider applicability. It is reasonable to expect to find examples where the equations of Tables 6 and 7 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 Kleene algebra is already used. For instance, Kleene algebra is already employed to analyze sequences of abstract program actions for security properties [11]. 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. Is it then 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 that used in [21]. Another intriguing question is whether the set-based program analysis framework of [22] is related to our approach.

Acknowledgements

The authors thank Bernhard Möller and the MPC referees for thoughtful comments and additional pointers to the literature.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools. Addison-Wesley (1989)
2. Overstreet, C.M., Cherinka, R., Sparks, R.: Using bidirectional data flow analysis to support software reuse. Technical Report TR-94-09, Old Dominion University, Computer Science Department (1994)
3. Moonen, L.: Data flow analysis for reverse engineering. Master's thesis, Programming Research Group, University of Amsterdam (1996)
4. Lo, R.W., Levitt, K.N., Olsson, R.A.: MCF: A malicious code filter. Computers and Security **14** (1995) 541–566
5. Bergeron, J., Debbabi, M., Desharnais, J., Erhioui, M.M., Lavoie, Y., Tawbi, N.: Static detection of malicious code in executable programs. In: 1st Symposium on Requirements Engineering for Information Security, Indianapolis, IN (2001)
6. Conway, J.H.: Regular Algebra and Finite Machines. Chapman and Hall, London (1971)
7. Desharnais, J., Möller, B., Struth, G.: Kleene algebra with domain. Technical Report 2003-7, Institut für Informatik, Universität Augsburg, D-86135 Augsburg (2003)
8. Desharnais, J., Möller, B., Struth, G.: Modal Kleene algebra and applications – A survey. Technical Report DIUL-RR-0401, Département d'informatique et de génie logiciel, Université Laval, D-86135 Augsburg (2004)
9. Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events. Information and Computation **110** (1994) 366–390
10. Kozen, D.: Kleene algebra with tests. ACM Transactions on Programming Languages and Systems **19** (1997) 427–443
11. Kozen, D.: Kleene algebra with tests and the static analysis of programs. Technical Report 2003-1915, Department of Computer Science, Cornell University (2003)
12. Fischer, C.N., LeBlanc, Jr., R.J.: Crafting a Compiler. Benjamin/Cummings Publishing Company, Menlo Park, CA (1988)
13. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer (1999)
14. Möller, B.: Derivation of graph and pointer algorithms. In Möller, B., Partsch, H.A., Schuman, S.A., eds.: Formal Program Development. Volume 755 of Lecture Notes in Computer Science. Springer, Berlin (1993) 123–160
15. Ng, K.C.: Relation algebras with transitive closure. PhD thesis, University of California, Berkeley (1984)
16. Ng, K.C., Tarski, A.: Relation algebras with transitive closure. Notices of the American Mathematical Society **24** (1977) A29–A30
17. Kozen, D.: Typed Kleene algebra. Technical Report 98-1669, Computer Science Department, Cornell University (1998)
18. Schmidt, G., Hattensperger, C., Winter, M.: Heterogeneous relation algebra. In Brink, C., Kahl, W., Schmidt, G., eds.: Relational Methods in Computer Science. Springer (1997)

19. Schmidt, G., Ströhlein, T.: Relations and Graphs. EATCS Monographs in Computer Science. Springer, Berlin (1993)
20. Desharnais, J.: Kleene algebra with relations. In Berghammer, R., Möller, B., Struth, G., eds.: Relational and Kleene-Algebraic Methods. Volume 3051 of Lecture Notes in Computer Science., Springer (2004) 8–20
21. Kozen, D., Patron, M.C.: Certification of compiler optimizations using Kleene algebra with tests. In Lloyd, J., Dahl, V., Furbach, U., Kerber, M., Lau, K.K., Palamidessi, C., Pereira, L.M., Sagiv, Y., Stuckey, P.J., eds.: Proc. 1st Int. Conf. on Computational Logic (CL2000), London. Volume 1861 of Lecture Notes in Artificial Intelligence., London, Springer (2000) 568–582
22. Heintze, N.: Set Based Program Analysis. PhD thesis, School of Computer Science, Computer Science Division, Carnegie Mellon University, Pittsburgh, PA (1992) CMU-CS-92-201.