Kleene Algebra and Bytecode Verification

Łucja Kot Dexter Kozen

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501, USA

Abstract

Most standard approaches to the static analysis of programs, such as the popular worklist method, are first-order methods that inductively annotate program points with abstract values. In [6] we introduced a second-order approach based on Kleene algebra. In this approach, the primary objects of interest are not the abstract data values, but the transfer functions that manipulate them. These elements form a Kleene algebra. The dataflow labeling is not achieved by inductively labeling the program with abstract values, but rather by computing the star (Kleene closure) of a matrix of transfer functions. In this paper we show how this general framework applies to the problem of Java bytecode verification. We show how to specify transfer functions arising in Java bytecode verification in such a way that the Kleene algebra operations (join, composition, star) can be computed efficiently. We also give a hybrid dataflow analysis algorithm that computes the closure of a matrix on a cutset of the control flow graph, thereby avoiding the recalculation of dataflow information along long paths. This method could potentially improve the performance over the standard worklist algorithm when a small cutset can be found.

Key words: Java, bytecode, verification, static analysis, abstract interpretation, Kleene algebra

1 Introduction

Dataflow analysis and abstract interpretation are concerned with the static derivation of information about the execution state at various points in a program. There is typically a semilattice L of *types* or *abstract values*, each describing a larger set of runtime values. Each instruction has one or more

Email addresses: lucja@cs.cornell.edu (Łucja Kot), kozen@cs.cornell.edu (Dexter Kozen).

associated *transfer functions* $f : L \to L$ that describe how the abstract state is transformed by the instruction.

The *worklist algorithm* for dataflow analysis is a standard method for computing a least fixpoint labeling of the nodes of the control flow graph G with elements of L [5]. Starting with initial information at the start node, dataflow information is propagated in a forward direction by applying a transfer function to the current dataflow information at a node and updating successor nodes until a fixpoint is achieved.

One disadvantage of the worklist approach is that long paths in the graph may be analyzed multiple times. For example, if a node s is labeled with $\ell \in L$, then later revisited and relabeled with $\ell' > \ell$, then any long paths out of s may be traversed again. The running time could be as bad as dn, where n is the size of the program and d is the depth of the semilattice, although in practice this worst-case bound is probably rarely attained. Thus the worklist algorithm remains a popular method for many practical program analysis tasks.

The worklist method is a first-order method in the sense that the primary objects of interest are the elements of the semilattice L. In [6] we introduced a second-order functional approach based on Kleene algebra. In this approach, the primary objects of interest are not the abstract data values, but the transfer functions that manipulate them. These elements form a Kleene algebra, a well-known family of algebraic structures with a rich theory and many applications in computer science. In the second-order approach, the dataflow labeling is not achieved by inductively labeling the program with abstract values, but rather by computing the star (Kleene closure) of a matrix of transfer functions.

In this paper we demonstrate how this general framework applies to the problem of bytecode verification. For concreteness, we focus on Java. The contributions of this paper are twofold: (i) we give an explicit specification mechanism for transfer functions that allows the Kleene algebra operations (join, composition, star) to be computed efficiently (Section 3); and (ii) we present a dataflow algorithm that computes the closure of a matrix of transfer functions on a cutset of the control flow graph, thereby avoiding the recalculation of dataflow information along long paths (Section 4). This method could potentially improve the performance over the standard worklist algorithm when a small cutset can be found.

2 Background

2.1 Upper Semilattices

Our abstract data values will form an upper semilattice L with join $+$ and bottom element $\bot$. The operation $+$ is associative, commutative, idempotent ($x + x = x$), and $x \leq y$ iff $x + y = y$. The element $\bot$ is the least element of the semilattice and is an identity for $+$. We also assume the *ascending chain condition* (ACC): no infinite ascending chains in L. This is a standard assumption that ensures that dataflow computations converge. It follows from this assumption that there exists a maximum element $\top$.

Intuitively, lower elements in the semilattice represent more specific information, and the join operation represents disjunction of information. For example, in the Java class hierarchy, the join of `String` and `StringBuffer` is `Object`, their least common ancestor in the hierarchy.

The element $\top$ represents a type error. In practice, any attempt by a dataflow analysis computation to form a join $x+y$ that does not make sense indicates a fatal type error, and the analysis will be aborted. We represent this situation mathematically by $x + y = \top$.

The element $\bot$ represents "unlabeled". For example, the initial labeling in the worklist algorithm is a map $w_0 : V \to L$, where V is the set of vertices of the control flow graph, such that $w_0(s_0)$ is the initial dataflow information available at the start node s_0, and $w_0(u) = \bot$ for all other nodes $u \in V$.

2.2 Semilattice Homomorphisms

We model transfer functions as semilattice homomorphisms $f : L \to L$, where L is an upper semilattice satisfying the ascending chain condition. The maps f must satisfy

$$\begin{aligned} f(x + y) &= f(x) + f(y) \\ f(\bot) &= \bot. \end{aligned} \tag{1}$$

There are particular semilattice homomorphisms

$$0 \stackrel{\text{def}}{=} \lambda x.\bot \qquad 1 \stackrel{\text{def}}{=} \lambda x.x.$$

The *domain* of f is the set

$\mathsf{dom}\, f \stackrel{\text{def}}{=} \{x \in L \mid f(x) \neq \top\}$.

The property (1) implies that $\mathsf{dom}\, f$ is closed downward under $\leq$.

2.3 Kleene Algebra

A *Kleene algebra* (KA) is a structure $(K, +, \cdot, {}^*, 0, 1)$, where $(K, +, \cdot, 0, 1)$ is an idempotent semiring, p^*q is the least x such that $q + px \leq x$, and qp^* the least x such that $q + xp \leq x$. Here "least" refers to the natural partial order $p \leq q \leftrightarrow p+q = q$. The operation $+$ gives the supremum with respect to $\leq$. An important fact is that the $n \times n$ matrices over a Kleene algebra again form a Kleene algebra under the appropriate definitions of the operators. We refer the reader to [7] for a more complete introduction.

In our application, the family of semilattice homomorphisms $f : L \to L$, used to model transfer functions, form a Kleene algebra in which $+$ is interpreted pointwise, $\cdot$ is interpreted as function composition, 0 and 1 are as defined in Section 2.2, and f^* is defined as the join of all finite powers of f. The ACC on L guarantees that this join exists.

For further details, we refer the reader to [6].

3 Application to Java

The Java bytecode verification algorithm, as described in the Java Virtual Machine specification [9], is a worklist algorithm. The official specification of the algorithm in [9] is operational, but there have been numerous attempts at a more mathematical treatment [1–3,8,10].

In this application, the elements of L describe the current state of the local variables and operand stack, which comprise the stack frame of the currently executing method. The top element $\top$ and bottom element $\bot$ of L are artificial elements representing a type error and an unlabeled state, respectively. Every other element of L consists of

(i) an assignment of types from a semilattice L_0, described below, to a local variable array, and
(ii) a bounded-depth operand stack containing values from L_0.

These assignments must satisfy certain constraints, as described below.

The semilattice L_0 describes the types of local variables and operand stack

elements. In the JVM, local variables do not have a fixed type, but are allowed to contain different types at different points of the program. The semilattice L_0 has top element `Useless` representing uninitialized or otherwise unusable values (not to be confused with the top element $\top$ of L).

When merging the state of the local variable array and operand stack at the confluence of two or more control flow paths, the resulting state is the join in L of the states produced by the different paths. The states must satisfy certain compatibility conditions, or they cannot be merged; in that case, the join in L is $\top$, representing a type error. For example, the stack depths must be the same, and the join in L_0 of corresponding stack entries may not be `Useless`. However, the join of corresponding local variables may very well be `Useless`.

Just below `Useless` in L_0 are several incomparable type hierarchies. The first is the Java class hierarchy with top element `Object` representing all reference values, including interfaces and arrays. Array types below `Object` consist of dimension and component type information. There is a least reference type `Null`, representing the null reference. The type `Null` is a subtype of all other reference types.

Also directly below `Useless` are the types `Int`, `Float`, `Long`, and `Double`. The type `Int` represents the Java primitive types `int`, `byte`, `char`, `short`, and `boolean`. In the Java virtual machine, all these values are represented as integers.

Finally, there is a collection of incomparable type hierarchies representing return addresses from embedded `jsr` subroutines used in the implementation of the Java try-catch-finally construct. These subroutines are well known to cause special problems for bytecode verification [2,3,8,10]. Given the extent of the additional complications introduced by `jsr/ret`, and given that they are a feature specific to Java rather than to bytecode in general, we have chosen to forego their treatment in this paper.

For $p, q \in L$, the join $p + q$ is defined iff

- the current stack depths of p and q are the same,
- the join in L_0 of corresponding local variable array elements in p and q is defined,
- the join in L_0 of corresponding stack elements in p and q is defined and is not `Useless`.

If $p + q$ is defined, its value is obtained by taking the join in L_0 of the corresponding local variable array elements in p and q and the same stack as in p and q. If $p + q$ is undefined, we take $p + q = \top$.

A transfer function $f : L \to L$ can be specified in terms of its *preconditions* and *effects*. The preconditions are a set of constraints that specify the domain of f, and the effects describe how f changes the abstract state.

The preconditions and effects can be encoded by triples

$$P = (\mathsf{oldD}, \mathsf{oldS}, \mathsf{oldL})$$
$$E = (\mathsf{newD}, \mathsf{newS}, \mathsf{newL}),$$

where:

- oldS is an array of assertions $\alpha \leq t$, where α is a variable and $t \in L_0$, or just an unconstrained variable α. Each α occurs at most once in oldS. These specify abstract values that are expected to occupy the top few positions on the stack just before execution, and constitute the precondition for typesafe execution.

  For example, the `iadd` (integer addition) instruction would have $\mathsf{oldS} = [\alpha \leq \texttt{Int}, \beta \leq \texttt{Int}]$, indicating that the instruction expects two integers on top of the stack. The `astore 3` instruction (store a reference value in local variable 3) would have $\mathsf{oldS} = [\alpha \leq \texttt{Object}]$, indicating that the instruction expects a reference value on top of the stack. The `swap` instruction would have $\mathsf{oldS} = [\alpha, \beta]$, indicating that the instruction expects two values of arbitrary type on top of the stack.

  The array oldS does not normally specify the entire stack, just a few of the topmost items. We denote the size of oldS by $|\mathsf{oldS}|$.
- oldD is the maximum allowed depth of the stack below oldS. This specifies how much free stack space must be available to execute f without stack overflow. For example, if f requires 5 free stack locations and $|\mathsf{oldS}|$ is 3, then $\mathsf{oldD} = \mathsf{maxS} - 8$, indicating that there may be at most $\mathsf{maxS} - 8$ additional elements on the stack below those specified by oldS. The number oldD may be any number between 0 and maxS, inclusive.
- oldL is an array of assertions $\alpha \leq t$, where α is a variable and $t \in L_0$, or just an unconstrained variable α, specifying the type constraints on local variables that are necessary for typesafe execution of f. Each α occurs at most once in oldL, and the variables in oldS and oldL must be disjoint. For example, the oldL array of the `aload 3` instruction (load of a reference type from local variable 3) would contain $\alpha \leq \texttt{Object}$ for local variable 3.
- newS is an array of expressions involving type values and variables representing the effect of the execution of f on the stack. For example, the `iadd` instruction would have $\mathsf{newS} = [\texttt{Int}]$, indicating that the instruction returns an integer on top of the stack. If the `swap` instruction had $\mathsf{oldS} = [\alpha, \beta]$, then it would have $\mathsf{newS} = [\beta, \alpha]$. If the `aload 3` instruc-

tion had $\alpha \leq$ `Object` for local variable 3, then it would have $\mathsf{newS} = [\alpha]$. We denote the size of newS by $|\mathsf{newS}|$.

- newD is a number that is either the same as oldD or 0. In most cases, it is the same as oldD, indicating that the stack below oldS is unmodified by the instruction. One exception to this is the `athrow` instruction, which empties the stack before pushing the exception object. For this instruction, or for any exception thrown by other means, newD will be 0.
- newL describes the explicit effects of f on the local variables. For example, the newL array of the `istore 2` instruction (integer store to local variable 2) would specify that local variable 2 contains α after execution of f, where $\mathsf{oldS} = [\alpha \leq$ `Int`$]$. Equivalently, local variable 2 of newL might just as well contain the constant `Int`, since `Int` is minimal in L_0, therefore $\alpha \leq$ `Int` $\Rightarrow$ $\alpha =$ `Int`.

  In addition, newL contains the constraints of oldL that are unaffected by f. For example, for the instruction `aload 3`, if the oldL array specified $\alpha \leq$ `Object` for local variable 3, then the newL array would contain α for local variable 3.

The arrays newL and newS may contain the symbolic joins of abstract types and type variables.

These properties will hold for all transfer functions defined from individual bytecode instructions, and our definition of join and composition will preserve them. Thus we can expect them to hold for all functions in our analysis.

3.2 The Transfer Function Specified by P, E

In this section we show how a specification P, E uniquely describes a transfer function $f : L \to L$.

The domain of f is the set of $p \in L$ such that (i)–(iii) below hold:

(i) For each of the topmost $|\mathsf{oldS}|$ elements of the stack in p, if the corresponding element of oldS is $\alpha \leq t$, then that element must be less than or equal to t. If the corresponding element of oldS is a variable α, the type is not constrained.

(ii) For each local variable x, if the x^{th} element of oldL is $\alpha \leq t$, then the x^{th} local variable of p must be less than or equal to t. If the x^{th} element of oldL is a variable α, then the x^{th} local variable of p is not constrained.

(iii) The stack depth at p is no greater than $\mathsf{oldD} + |\mathsf{oldS}|$.

Finally, we specify the value of $f(p)$, where $p \in \mathsf{dom}\, f$. For each local variable x, if the x^{th} element of oldL is $\alpha \leq t$ or α, unify α with the x^{th} element of p. Similarly, for each element of oldS, if that element is either $\alpha \leq t$ or α, unify α

with the corresponding element of the stack of p. Now for each local variable x, evaluate the x^{th} element of newL, which is a symbolic join of variables and constants in L_0, under this substitution. That will be the x^{th} element of the local variable array of $f(p)$. The values of the stack of $f(p)$ corresponding to newS are obtained similarly. If oldD = newD, the remaining elements on the stack at $f(p)$ are unchanged. Otherwise, if newD = 0, the stack contents at $f(p)$ will be just newS.

3.3 Operations on Transfer Functions

3.3.1 Lengthening

In this section we describe the Kleene algebra operations on specifications of transfer functions. Before doing so, however, we present an auxiliary operation that is of use when comparing two specifications with different oldS or newS lengths. Given a specification P, E such that oldD = newD ≥ 1, we can *lengthen* the stacks by adding a new unconstrained variable α to both oldS and newS immediately under the elements already represented there and decrementing oldD and newD by 1. The resulting specification P', E' represents the same transfer function f as P, E with the added restriction that the stacks are constrained to have at least one additional element below oldS and newS.

In case oldD ≥ 1 but newD $= 0$, as for example with the `athrow` instruction, we can lengthen just oldS by adding a new unconstrained variable α to oldS immediately under the elements already represented there and decrementing oldD by 1. The resulting specification P', E' represents the same transfer function f as P, E with the added restriction that oldS must have at least one more element than previously required.

3.3.2 Join

Given specifications P_f, E_f and P_g, E_g defining transfer functions f and g, respectively, we wish to define P_{f+g} and E_{f+g}. Intuitively, we would like P_{f+g} to be the weakest set of constraints implying both P_f and P_g, and we would like E_{f+g} to be the join of E_f and E_g.

The constraints that P_f and P_g place on stack depth must not be so strong as to prevent the merging of the stacks. Thus, all of the following properties must hold:

$$\begin{aligned}
\mathsf{oldD}_f + |\mathsf{oldS}_f| &\geq |\mathsf{oldS}_g| \\
\mathsf{oldD}_g + |\mathsf{oldS}_g| &\geq |\mathsf{oldS}_f| \\
\mathsf{newD}_f + |\mathsf{newS}_f| &\geq |\mathsf{newS}_g| \\
\mathsf{newD}_g + |\mathsf{newS}_g| &\geq |\mathsf{newS}_f|.
\end{aligned}$$

First, if $|\mathsf{oldS}_f| \neq |\mathsf{oldS}_g|$, say $|\mathsf{oldS}_f| < |\mathsf{oldS}_g|$, we lengthen oldS_f as described in Section 3.3.1 until they are the same length. If this is impossible because $\mathsf{oldD}_f = 0$, it is a type error. Thus we can assume without loss of generality that $|\mathsf{oldS}_f| = |\mathsf{oldS}_g|$.

To define P_{f+g}, we first set

$$\mathsf{oldD}_{f+g} \stackrel{\text{def}}{=} \min(\mathsf{oldD}_f, \mathsf{oldD}_g).$$

This sets oldD_{f+g} to the stricter of the two constraints imposed by oldD_f and oldD_g.

The contents of the array oldS_{f+g} are the weakest constraints that imply the constraints imposed by both oldS_f and oldS_g. To define element i in oldS_{f+g}, locate the corresponding elements in oldS_f and oldS_g, counting from the top of the stack. Call these items i_f and i_g. The value of element i in oldS_{f+g} is defined as follows.

- If one of i_f, i_g is $\alpha \leq s$ and the other is either $\beta \leq t$ with $s \leq t$ or just β, then the corresponding constraint in oldS_{f+g} is $\alpha \leq s$, since it is the stricter constraint. Unify α and β in P_f, E_f, P_g, E_g.
- If i_f is α and i_g is β, unify the two variables in P_f, E_f, P_g, E_g. The corresponding element of oldS_{f+g} is just α.
- If i_f is $\alpha \leq s$ and i_g is $\beta \leq t$ with neither $s \leq t$ nor $t \leq s$, it is a type error.

We define oldL_{f+g} similarly from oldL_f and oldL_g. If the variables in the two arrays are both constrained, say by s and t with $s \leq t$, then unify the two variables in P_f, E_f, P_g, E_g and constrain it with s in oldL_{f+g}. If one of the elements is unconstrained, take the other constraint and unify the two variables.

For E_{f+g}, we must have $|\mathsf{newS}_f| = |\mathsf{newS}_g|$, otherwise it is a type error. Set

$$\mathsf{newD}_{f+g} \stackrel{\text{def}}{=} \min(\mathsf{newD}_f, \mathsf{newD}_g).$$

The intuition behind this is the same as for oldD_{f+g}.

Define newL_{f+g} to be the join of newL_f and newL_g. That is, to obtain a particular element in newL_{f+g}, take the join in L_0 of the corresponding elements

in newL_f and newL_g. The resulting expression can be simplified if necessary using associativity, commutativity, and idempotence. If any join of two type values in this process is `Useless`, it is not a type error.

Similarly, define newS_{f+g} to be the join of newS_f and newS_g, except that a `Useless` value is a type error.

3.3.3 Composition

Say we are given specifications P_f, P_g, E_f, E_g of transfer functions f and g. We wish to define P_{fg} and E_{fg}. For the composition to be legal, the following conditions must hold:

$$\begin{aligned} \mathsf{newD}_f + |\mathsf{newS}_f| &\geq |\mathsf{oldS}_g| \\ \mathsf{oldD}_g + |\mathsf{oldS}_g| &\geq |\mathsf{newS}_f|. \end{aligned}$$

If $|\mathsf{newS}_f| \neq |\mathsf{oldS}_g|$, we first lengthen the shorter one as described in Section 3.3.1. If this is not possible because one of oldD_g or newD_f is 0, it is a type error. Thus we can assume without loss of generality that $|\mathsf{newS}_f| = |\mathsf{oldS}_g|$.

First we define oldD_{fg}. There are two cases, depending on f:

$$\mathsf{oldD}_{fg} \stackrel{\text{def}}{=} \begin{cases} \min(\mathsf{newD}_f, \mathsf{oldD}_g), & \text{if } \mathsf{oldD}_f = \mathsf{newD}_f \\ \mathsf{oldD}_f, & \text{otherwise.} \end{cases}$$

To construct oldL_{fg}, we start with oldL_f and modify it as follows.

If local variable x of newL_f contains an expression e with type constant $s \in L_0$ and one or more type variables, and if local variable x of oldL_g is of the form $\alpha \leq t$, we must have $s \leq t$, otherwise it is a type error. Intuitively, the type produced by f in that position can be at least s, thus g must not place a stronger constraint on that element.

Moreover, for all variables β in e, if the constraint $\beta \leq u$ appears in oldL_f or oldS_f and $t \leq u$, or if β appears unconstrained in oldL_f or oldS_f, replace the constraint $\beta \leq u$ or the unconstrained occurrence of β in oldL_f or oldS_f with $\beta \leq t$. Intuitively, the stronger constraint $\beta \leq t$ imposed by oldL_g propagates backward through f. If $u \leq t$, we do not alter the constraint $\beta \leq u$. If neither $u \leq t$ nor $t \leq u$, it is a type error.

When this has been done for all local variables x, the resulting array is oldL_{fg}.

A similar construction holds for oldS_{fg}. We start with oldS_f. If any element of newS_f is an expression e with type constant $s \in L_0$ and one or more type variables, and if the corresponding element of oldS_g is of the form $\alpha \leq t$, we must have $s \leq t$, otherwise it is a type error. Moreover, as described above, for all variables β in e, we propagate the constraint $\beta \leq t$ backwards through f if necessary.

Define E_{fg} as follows. Again, there are two cases for newD_{fg}, depending on f:

$$\mathsf{newD}_{fg} \stackrel{\text{def}}{=} \begin{cases} \min(\mathsf{newD}_f, \mathsf{oldD}_g), & \text{if } \mathsf{oldD}_g = \mathsf{newD}_g \\ \mathsf{newD}_g, & \text{otherwise.} \end{cases}$$

We compute newL_{fg} and newS_{fg} as follows. Start with newL_g and newS_g, respectively. For each local variable with α or $\alpha \leq t$ in oldL_g, unify α with the expression occurring in the corresponding location in newL_f, and apply this substitution to newL_g and newS_g, evaluating and simplifying expressions if necessary. Similarly, for each stack entry α or $\alpha \leq t$ in oldS_g, unify α with the expression occurring in the corresponding location in newS_f, and apply this substitution to newL_g and newS_g, evaluating and simplifying if necessary. A type error is signaled if `Useless` appears in the evaluation of expressions in newS_g. The resulting arrays are newL_{fg} and newS_{fg}, respectively.

3.3.4 Identity

The identity function $1 \stackrel{\text{def}}{=} \lambda p.p$ is specified by:

$$P_1, E_1 \stackrel{\text{def}}{=} (\mathsf{maxS}, [\,], A),$$

where $[\,]$ denotes the empty stack and A is an array of maxL distinct unconstrained variables.

3.3.5 Star

Given a specification P, E of a transfer function f, a specification of f^* can be computed by taking the join of sufficiently many finite powers of f. For this not to result in a type error, we had better have $|\mathsf{oldS}_f| = |\mathsf{newS}_f|$: if $|\mathsf{oldS}_f| < |\mathsf{newS}_f|$, then some power of f will result in a stack overflow, and if $|\mathsf{oldS}_f| > |\mathsf{newS}_f|$, then some power of f will result in a stack underflow.

It suffices to take the join of powers f^k up to $k = |\mathsf{oldS}_f| + \mathsf{maxL}$, since this is an upper bound on the number of steps needed for any variable or constant appearing in oldS_f or oldL_f to propagate to an expression in newS_{f*} or newL_{f*}.

Thus $f^* = (1+f)^k$ for $k = |\mathsf{oldS}_f| + \mathsf{maxL}$, which we can compute by repeated squaring in $\log k$ steps.

4 An Algorithm

In this section we present a hybrid algorithm for dataflow analysis that may give an improvement in performance over the standard worklist algorithm when a small cutset can be found. The algorithm exploits the ability to compute the Kleene algebra operations on transfer functions as defined above.

We are given a program with n instructions, and we wish to label the underlying control flow graph G of the program with elements of the semilattice L. Let E be the $n \times n$ matrix with rows and columns indexed by the vertices of G such that if (s,t) is an edge of G, then $E\,[s,t]$ is the transfer function labeling the edge (s,t), and $E\,[s,t] = 0$ if (s,t) is not an edge of G. This matrix is easily constructed in a single pass thorough the program.

Recall from Section 2.3 that the $n \times n$ matrices over a Kleene algebra again form a Kleene algebra. We can thus speak of the matrix E^*. The entry $E^*\,[u,v]$ is the join of the composition of transfer functions along all paths from u to v. If we can compute E^*, then we can obtain the desired fixpoint dataflow labeling at any node u of G by evaluating $E^*\,[s_0,u](\ell_0)$, where $\ell_0 \in L$ is the initial label of the start node s_0. The label ℓ_0 consists of an empty stack, the types of the arguments to the method (including the object itself if it is an instance method) in the first few local variables, and `Useless` for the remaining local variables. The value of a transfer function given by its specification P, E on an element $\ell \in L$ can be computed by unifying the variables in oldS and oldL with the corresponding values in ℓ, checking that all constraints $\alpha \leq t$ in oldS and oldL are satisfied, then evaluating the expressions in newS and newL under this substitution.

It is shown in [6] that an abstracted version of this method and the standard worklist algorithm produce the same fixpoint labeling on all type-correct programs.

4.1 Small Cutsets

We do not compute E^* directly, because it is too big. Instead, we propose the following hybrid method that uses the preceding ideas in conjunction with the worklist algorithm to avoid recalculating dataflow information along long paths.

Let M be a *cutset* (also known as a *feedback vertex set*) in G; that is, a set of nodes such that every directed cycle of G contains at least one node in M. We also include the start node s_0 in M, even though s_0 may not be a cutpoint. Let $m = |M|$. Finding a minimal cutset is known to be *NP*-complete, but solvable in polynomial time for reducible graphs [4]. Flowgraphs of bytecode programs compiled from Java source would ordinarily be reducible. In practice, simply taking M to be the set of all targets of back edges should give a very small cutset.

Let A, B, C, and D be the $M \times M$, $M \times (V - M)$, $(V - M) \times M$, and $(V - M) \times (V - M)$ submatrices of E, respectively. Let $F \stackrel{\text{def}}{=} A + BD^*C$. By Kleene algebra,

$$E^* = \begin{bmatrix} F^* & F^*BD^* \\ D^*CF^* & D^* + D^*CF^*BD^* \end{bmatrix}. \tag{2}$$

The fact that M is a cutset is reflected algebraically by the property $D^{n-m} = 0$. This is because D^{n-m} describes the labels of paths of length $n - m$ through $V - M$; but by the pigeonhole principle, any such path would have a repeated node, thus would contain a cycle, which must intersect M. Therefore no such path can exist.

It follows from this and the theorem $x^* = (1 + x)^{k-1}(x^k)^*$ of Kleene algebra that $D^* = (I + D)^{n-m-1}$, hence

$$F = A + BD^*C = A + B(I + D)^{n-m-1}C.$$

The $m \times m$ matrix F describes the labels of paths from a cutpoint to another cutpoint that do not go through an intermediate cutpoint. Since the subgraph on $V - M$ is acyclic, F can be computed in time $O(mn)$ using the traditional worklist algorithm starting from every cutpoint. In each such computation, each vertex of $V - M$ is visited at most once. Alternatively, we could topologically sort the subgraph and compute the compositions in sorted order.

As a byproduct of this computation, we also obtain the matrix

$$G \stackrel{\text{def}}{=} BD^* = B(I + D)^{n-m-1},$$

which describes the labels of paths from a cutpoint to a non-cutpoint that do not go through any other cutpoint.

Now we need to compute the star of F, but this matrix will typically be much

smaller than E. We can do this by repeated squaring, which achieves the fixpoint F^* after at most $\log d$ steps, where d is the depth of the semilattice L, or by a recursive divide-and-conquer method using the recursive definition of the star of a matrix (2). The first method requires time $O(m^3 \log d)$ and the second requires time $O(m^3)$ in the worse case.

Now to achieve the final dataflow labeling, we observe that the s_0^{th} row of F^* is a vector of transfer functions $F^*[s_0, u]$, one for each cutpoint u, which when applied to ℓ_0 yields the final dataflow labeling of u. Similarly, the s_0^{th} row of F^*G is a vector of transfer functions $F^*G[s_0, v]$, one for each non-cutpoint v, which when applied to ℓ_0 yields the final dataflow labeling of v. Each of the m values $F^*[s_0, u](\ell_0)$ can be calculated in constant time, or $O(m)$ in all. Once we have this vector of values, we can calculate

$$F^*G[s_0, v](\ell_0) = \sum_{u \in M} G[u, v](F^*[s_0, u](\ell_0)),$$

which takes time $O(m)$ for each $v \in V$, or $O(nm)$ in all.

4.2 Complexity

The worst-case complexity of this algorithm is $O(nm + m^3)$. Compared with the worst-case complexity of the worklist algorithm, namely $O(nd)$ where d is the depth of the semilattice L, our algorithm may give an improvement when m is small.

One other advantage of the second-order method is that it is amenable to parallelization. The worklist method is inherently sequential, since each application of a transfer function requires knowledge of its inputs, whereas compositions can be computed without knowing their inputs. Such questions remain for future investigation.

Acknowledgments

We are indebted to Stephen Chong, Andrew Myers, and Radu Rugina for valuable discussions. This work was supported in part by NSF grant CCR-0105586 and ONR Grant N00014-01-1-0968. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the US Government.

References

[1] M. Abadi and R. Stata. A type system for Java bytecode subroutines. In *Proc. 25th Symp. Principles of Programming Languages*, pages 149–160. ACM SIGPLAN/SIGACT, January 1998.

[2] Alessandro Coglio. Simple verification technique for complex java bytecode subroutines. *Concurrency and Computation: Practice and Experience*, 16(7):647–670, June 2004.

[3] Stephen N. Freund and John C. Mitchell. A type system for the java bytecode language and verifier. *J. Automated Reasoning*, 30:271–321, 2003.

[4] M. R. Garey and D. S. Johnson. *Computers and Intractibility: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[5] Gary A. Kildall. A unified approach to global program optimization. In *Proc. Conf. Principles of Programming Languages (POPL'73)*, pages 194–206. ACM, 1973.

[6] Łucja Kot and Dexter Kozen. Second-order abstract interpretation via Kleene algebra. Technical Report 2004-1971, Computer Science Department, Cornell University, December 2004.

[7] Dexter Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Infor. and Comput.*, 110(2):366–390, May 1994.

[8] Xavier Leroy. Java bytecode verification: an overview. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. Conference on Computer Aided Verification (CAV 2001)*, volume 2102 of *Lect. Notes in Computer Science*, pages 265–285. Springer-Verlag, 2001.

[9] Tim Lindholm and Frank Yellin. *The JAVA virtual machine specification*. Addison Wesley, 1996.

[10] Zhenyu Qian. Standard fixpoint iteration for java bytecode verification. *Transactions on Programming Languages and Systems*, 22(4):638–672, July 2000.