Second-Order Abstract Interpretation via Kleene Algebra

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 this paper we introduce 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 introduce the method and prove soundness and completeness with respect to the standard worklist algorithm.

Key words: 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 possible runtime values. The natural partial order associated with L is the subtype relation. The objective of the analysis is to associate an element of L with each point of the program that represents what is known about the program state whenever control passes through that point. This may consist of type information, bounds on values of variables or

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

Preprint submitted to Elsevier Science *20 December 2004*

registers, operand stack depth, the shape of data structures, whether pointers are null, etc. It is usually only an approximation; the lower in the semilattice, the better the approximation.

Each instruction has one or more associated *transfer functions* $f : L \to L$ that describe how the state is transformed by the instruction. The domain of f is determined by the type of the instruction. Membership in the domain of f may be considered a precondition for the safe execution of the instruction; attempting to apply f to an element of L not in its domain signals a type error. For example, an empty stack should not be popped. Transfer functions may be composed, provided there is no type mismatch. Transfer functions may be polymorphic, as for example in the case of a swap operation that interchanges the top two elements of a stack.

Each instruction has zero or more *successors*. Data manipulation instructions such as loads, stores, and arithmetic operations have the fallthough instruction as successor. Conditional jumps have the fallthrough as well as the jump target, and unconditional jumps have only the jump target. Any instruction that can raise an exception has the entry point of an exception handler as a successor. Successors are determined solely by statically available information. Thus the program can be modeled by a directed control flow graph G whose nodes are the instructions and whose edges go from each instruction to the successors of that instruction. The edges are labeled with the transfer functions associated with that instruction. In most cases the transfer function is the same for all edges exiting a particular node, but in some cases it is different. For example, in Java bytecode, if an instruction throws an exception, then the operand stack is cleared and the exception object pushed onto the stack before invoking the handler. The successor state corresponding to the exception handler thus reflects a different stack configuration than that corresponding to the fallthrough instruction.

The *worklist algorithm* for dataflow analysis is a standard method for computing a least fixpoint labeling of the nodes of G with elements of L [3]. It works as follows. First, the entry point of the method is labeled with the element of L describing the initial state of the computation. For example, in Java bytecode, the initial label consists of an empty operand 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 a special undefined marker for the remaining local variables. This node is marked as changed and placed on a worklist. Then, as long at the worklist is nonempty, the procedure repeatedly removes the next element s of the worklist, and for each exiting edge (s, t), applies the transfer function f associated with that edge to the label $x \in L$ of s to get $f(x)$, then updates the label of the successor t with $f(x)$. If t is unlabeled, then it is labeled with $f(x)$. If t is already labeled, then it is relabeled with the join of $f(x)$ and its current label in the semilattice, if the join exists.

This indicates the best information that is known about the program state at t from the various control flow paths into t that have been analyzed so far. If the join of $f(x)$ and the current label of t does not exist in the semilattice, then it is a type error. If t is successfully labeled and the new label is different from the old, then t is marked as changed and placed back on the worklist. When the worklist becomes empty, the resulting labeling is the least fixpoint of a monotone mapping on labelings defined in terms of the transfer functions.

One disadvantage of the worklist approach is that long paths in the graph may be analyzed several times. For example, if a node s is labeled with $x \in L$, then later revisited and relabeled with $y > x$, 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 this worst-case bound is probably rarely attained in practice. Thus the worklist algorithm remains a popular method for many practical program analysis tasks.

In this paper we describe an alternative approach that can be used to avoid the recalculation of dataflow information along long paths using a symbolic method based on Kleene algebra. The elements of the algebra are transfer functions. The novelty of this approach is that it is the transfer functions, not the data values, that are the objects of primary algebraic interest. The transfer functions are elements of a certain algebraic structure called a *Kleene algebra* with the operations of composition, join, and iteration. The control flow graph of a method with n instructions gives rise to an $n \times n$ matrix of transfer functions, and computing the star or Kleene closure of this matrix amounts to computing the dataflow information at all points of the program simultaneously.

In a companion paper [5], we describe a concrete implementation of these ideas in the context of Java bytecode. That implementation combines the second-order approach introduced in this paper with the standard worklist algorithm to obtain a hybrid algorithm with running time $O(nm + m^3)$, where m is the size of a cutset (a set of nodes breaking all cycles in G). The algorithm avoids recalculation of dataflow information along long paths by computing the star (closure) of an $m \times m$ matrix of transfer functions. This may give an improvement when m is small compared to n.

It is also apparent that our second-order method 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. The detailed development of a parallel static analysis algorithm based on these ideas is an intriguing research topic that we leave for future investigation.

In this paper, we lay the foundations of this approach and prove correctness with respect to the standard worklist algorithm [3]. This paper is organized as follows. In Section 2, we review the pertinent definitions of Kleene algebra, including the formation of Kleene algebras of matrices over another Kleene algebra, and describe how transfer functions can be modeled as semilattice homomorphisms on a semilattice of abstract values or types. In Section 3, we present an alternative approach to static analysis based on computing the star or Kleene closure of a matrix of transfer functions. Finally, in Section 4 we prove that our second-order method produces the same final dataflow labeling as the standard worklist algorithm on all type-correct programs.

2 Background

2.1 Upper Semilattices

An *upper semilattice* is a partially ordered set L in which every finite set has a least upper bound, which must be unique. The least upper bound of two elements x and y is denoted $x + y$. The least upper bound of the null set is denoted $\perp$. The operation $+$ is associative, commutative, idempotent ($x + x = x$), and $x \leq y$ iff $x + y = y$. The element $\perp$ is the least element of the semilattice and is an identity for $+$.

We also assume that there are no infinite ascending chains in L; this is known as the *ascending chain condition* (ACC). 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 $\perp$ 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 $w_0(u) = \perp$ for all other nodes $u \in V$.

The ascending chain condition (ACC) is a standard assumption that ensure that dataflow computations always converge.

2.2 Kleene Algebra

Kleene algebra was introduced by S. C. Kleene [4] (see also [2]). We define a *Kleene algebra* (KA) to be 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 $\leq$. This particular axiomatization is from [6].

We normally omit the $\cdot$, writing pq for $p \cdot q$. The precedence of the operators is $^* > \cdot > +$. Thus $p + qr^*$ should be parsed $p + (q(r^*))$.

Typical models include the family of regular sets of strings over a finite alphabet, the family of binary relations on a set, and the family of $n \times n$ matrices over another Kleene algebra.

The following are some elementary theorems of KA.

$$p^* = 1 + pp^* = 1 + p^*p = p^*p^* = p^{**} \tag{1}$$

$$p(qp)^* = (pq)^*p \tag{2}$$

$$p^*(qp^*)^* = (p+q)^* = (p^*q)^*p^* \tag{3}$$

$$qp = 0 \rightarrow (p+q)^* = p^*q^* \tag{4}$$

$$px = xq \rightarrow p^*x = xq^* \tag{5}$$

The identities (2) and (3) are called the *sliding rule* and the *denesting rule*, respectively. These rules are particularly useful in program equivalence proofs. The property (5) is a kind of bisimulation property. It plays a prominent role in the completeness proof of [6]. We refer the reader to [6] for further definitions and basic results.

2.3 Matrices over a Kleene Algebra

As mentioned, the $n \times n$ matrices over a Kleene algebra again form a Kleene algebra under the appropriate definitions of the operators. We mention a few important properties that are preserved by this construction.

A Kleene algebra is *star-continuous* if it satisfies the infinitary property $pq^*r = \sup_n pq^nr$, where the supremum is with respect to the natural order $\leq$. This axiom is strictly stronger than the axioms for * given above. A Kleene algebra is *finitary* if for every a there exists an m such that $a^* = (1+a)^m$. Every finitary algebra is star-continuous. Each of these properties holds of the matrix algebra if it holds of the underlying algebra.

In applications, we will be considering Kleene algebras of functions taking values in a semilattice satisfying the ascending chain condition (ACC): no infinite ascending chains. This property implies that the Kleene algebra is star-continuous, hence the algebra of matrices over this algebra is also star-continuous.

2.4 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

$$\begin{aligned} f(x+y) &= f(x) + f(y) \\ f(\bot) &= \bot. \end{aligned} \tag{6}$$

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

Let K denote the family of semilattice homomorphisms on L. We can impose a Kleene algebra structure on K as follows. First, order K pointwise:

$$f \leq g \stackrel{\text{def}}{\Longleftrightarrow} \forall x\ f(x) \leq g(x).$$

Under this definition, K forms an upper semilattice with pointwise join operation

$$(f+g)(x) \stackrel{\text{def}}{=} f(x) + g(x)$$

and least element 0.

Elements of K can be composed using ordinary functional composition. The operator is written $\cdot$ and the composition of f followed by g is written fg; thus $(fg)(x) = g(f(x))$. Note that $x \in \mathsf{dom}\, fg$ iff $x \in \mathsf{dom}\, f$ and $f(x) \in \mathsf{dom}\, g$. The identity function 1 is a two-sided identity for composition.

Also, composition distributes over $+$ on both sides. The right distributivity law $(f+g)h = fh + gh$ depends on the fact (6) that h preserves joins. Thus K forms an idempotent semiring under the operations $+, \cdot, 0, 1$.

The element f^* is defined as the join of all finite powers of f, including the 0^{th} power, which is defined as 1:

$$f^* \stackrel{\text{def}}{=} 1 + f + f^2 + f^3 + \cdots .$$

This join always exists because of the ascending chain condition on L. In fact, because of the ACC, $f^*(x)$ is always the join of finitely many finite powers of f applied to x; that is,

$$\forall x\ \exists n\ f^*(x) = x + f(x) + \cdots + f^n(x) = (1+f)^n(x). \tag{7}$$

The star-continuity condition follows from finite distributivity. However, there does not necessarily exist an n such that $f^* = 1 + f + \cdots + f^n$, since the n in (7) may depend on x. A counterexample is given by the semilattice consisting of $\mathbb{N} \cup \{\infty\}$ with min as join and the semilattice morphism f that on input x gives ∞ if $x = \infty$, $x - 1$ if $x \geq 1$, and 0 if $x = 0$. Thus K is not necessarily finitary.

Theorem 2.1 *The structure* $(K, +, \cdot, {}^*, 0, 1)$ *is a star-continuous Kleene algebra.*

3 A Second-Order Approach

In this section we present a general second-order approach to static analysis. The technique 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 2.2 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. The desired fixpoint dataflow labeling at any node u of G can be obtained by evaluating $E^* [s_0, u] (\ell_0)$, where $\ell_0 \in L$ is the initial label of the start node s_0. Thus the inductive labeling of the control flow graph is replaced with the computation of the matrix E^*.

A concrete implementation of this method is described in [5] in the context of Java bytecode. In that implementation, E^* is not computed directly, but rather used in conjunction with the worklist algorithm to obtain a hybrid method that uses matrix closure on a small cutset to avoid recalculation dataflow information along long paths in the control flow graph.

4 Soundness and Completeness

We argue in this section that the second-order algorithm proposed in Section 3 and the standard worklist algorithm produce the same final dataflow labeling for any type-correct program.

Let L be an upper semilattice with top satisfying the ACC as described in Section 2.1, and let K be the Kleene algebra of semilattice homomorphisms $L \to L$ as described in Section 2.4. Let G be a control flow graph with vertices V, $n = |V|$, start node $s_0 \in V$, and edges labeled with transfer functions $f \in K$. Let $\ell_0 \in L$ be the initial dataflow information at s_0.

Formally, the worklist algorithm computes a sequence of labelings $w_n : V \to L$, $n \geq 0$, as follows. We start with the initial labeling

$$w_0(u) \stackrel{\text{def}}{=} \begin{cases} \ell_0, \text{ if } u = s_0 \\ \perp, \text{ otherwise.} \end{cases}$$

At stage n, say we have constructed a labeling w_n. To get w_{n+1}, we take the next edge (u, v) from the worklist, apply the associated transfer function $E [u, v]$ to the current label $w_n(u)$ of u, and update the label of v with that value. Thus

$$w_{n+1}(t) = \begin{cases} E [u, v] (w_n(u)) + w_n(v), & \text{if } t = v, \\ w_n(t), & \text{if } t \neq v. \end{cases} \tag{8}$$

The sequence $w_0, w_1, \ldots$ is monotone and converges to a fixpoint

$$w_* \stackrel{\text{def}}{=} \sup_n w_n.$$

This labeling is the least labeling such that for all u reachable from the start node s_0,

$$E[u,v](w_*(u)) \le w_*(v) \tag{9}$$

[3]. For vertices u not reachable from s_0, the worklist algorithm will never see them, but (9) will still hold for those vertices, since $w_*(u) = \bot$ and $E[u,v](\bot) = \bot \le w_*(v)$. Note that v may still be reachable from s_0, even if u is not.

To compare this algorithm to our second-order algorithm of Section 3, it will be convenient to consider labelings of vertices with certain semilattice homomorphisms $L \to L$ instead of elements of L. We lift an element $x \in L$ to an almost-constant semilattice homomorphism $\widehat{x} \in K$ as follows:

$$\widehat{x}(y) \stackrel{\text{def}}{=} \begin{cases} x, & \text{if } y \ne \bot \\ \bot, & \text{otherwise.} \end{cases}$$

Note that $\widehat{\bot} = 0$. The value x can be recovered from $\widehat{x}$ by applying $\widehat{x}$ to any element of L besides $\bot$. The advantage of using lifted values is that function application becomes composition, which is a Kleene algebra operation:

$$\widehat{f(x)} = \widehat{x} f. \tag{10}$$

If $w : V \to L$ is a labeling, we can lift it to a second-order labeling $\widehat{w} : V \to K$ by taking

$$\widehat{w}[u] \stackrel{\text{def}}{=} \widehat{w(u)}.$$

For example, the lifted version of the initial labeling w_0 is

$$\widehat{w}_0[u] = \begin{cases} \widehat{\ell}_0, & \text{if } u = s_0 \\ 0, & \text{otherwise.} \end{cases}$$

Although both w and $\widehat{w}$ are functions on V, we write $\widehat{w}[u]$ with square brackets because we will be regarding it as a row vector of length n and using it in matrix-vector computations.

We are now ready to prove our main theorem.

Theorem 4.1 *Let $w_* : V \to L$ be the final dataflow labeling produced by the worklist algorithm. Then for all $u \in V$,*

$$E^* [s_0, u] (\ell_0) = w_*(u). \tag{11}$$

Proof. Let w_0 be the initial labeling, and consider the matrix-vector product $\widehat{w}_0 E^*$, which is a row vector $V \to K$. We first show that

$$\widehat{w}_0 E^* = \widehat{w}_*. \tag{12}$$

By Kleene algebra, the left-hand side is the least solution of

$$\widehat{w}_0 + XE \leq X, \tag{13}$$

so it suffices to show that $\widehat{w}_*$ is as well, since the least solution of (13) is unique. For all $u \in V$ and $x \neq \bot$,

$$\widehat{w}_0 [u] (x) = w_0(u) \leq \sup_n w_n(u) = w_*(u) = \widehat{w}_* [u] (x),$$

therefore $\widehat{w}_0 \leq \widehat{w}_*$. Similarly,

$$\begin{aligned}
(\widehat{w}_* E) [u] (x) &= (\sum_v \widehat{w}_* [v] E [v, u])(x) \\
&= \sum_v \widehat{w}_* [v] E [v, u] (x) \\
&= \sum_v E [v, u] (w_*(v)) \\
&\leq w_*(u) && \text{by (9)} \\
&= \widehat{w}_* [u] (x).
\end{aligned}$$

Since u and $x \neq \bot$ were arbitrary, $\widehat{w}_* E \leq \widehat{w}_*$. Since both $\widehat{w}_0 \leq \widehat{w}_*$ and $\widehat{w}_* E \leq \widehat{w}_*$, we have $\widehat{w}_0 + \widehat{w}_* E \leq \widehat{w}_*$, therefore $\widehat{w}_*$ is a solution to (13).

To show that it is the least solution, let X be any other solution. Then $\widehat{w}_0 \leq X$. Reasoning inductively, suppose that $\widehat{w}_n \leq X$. Let (s, t) be the edge selected by the worklist algorithm at stage n. For $x \neq \bot$,

$$
\begin{aligned}
\widehat{w}_{n+1}[u](x) &= w_{n+1}(u) \\
&= \begin{cases} E[v,u](w_n(v)) + w_n(u), & \text{if } (v,u) = (s,t) \\ w_n(u), & \text{otherwise} \end{cases} \\
&\le \sum_v E[v,u](w_n(v)) + w_n(u) \\
&= \sum_v \widehat{w}_n[v]\, E[v,u](x) + \widehat{w}_n[u](x) \\
&= (\widehat{w}_n E)[u](x) + \widehat{w}_n[u](x) \\
&= (\widehat{w}_n(E+1))[u](x),
\end{aligned}
$$

therefore $\widehat{w}_{n+1} \le \widehat{w}_n(E+1)$. It follows that

$$
\widehat{w}_{n+1} \le \widehat{w}_n(E+1) \le X(E+1) \le XE + X \le X.
$$

Then $\widehat{w}_* = \sup_n \widehat{w}_n \le X$, so $\widehat{w}_*$ is the least solution of (13).

Finally, we show how (11) follows from (12). Let $u \in V$. Since $\widehat{w}_0[v] = 0$ for $v \ne s_0$, we have

$$
\begin{aligned}
\widehat{w}_0[s_0]\, E^*[s_0,u] &= \widehat{w}_0[s_0]\, E^*[s_0,u] \ + \sum_{v \ne s_0} \widehat{w}_0[v]\, E^*[v,u] \\
&= \sum_v \widehat{w}_0[v]\, E^*[v,u] \\
&= (\widehat{w}_0 E^*)[u] \\
&= \widehat{w}_*[u] \qquad \text{by (12)},
\end{aligned}
$$

thus for any $x \in L - \{\perp\}$,

$$
\begin{aligned}
E^*[s_0,u](\ell_0) &= E^*[s_0,u](w_0(s_0)) = \widehat{w}_0[s_0]\, E^*[s_0,u](x) \\
&= \widehat{w}_*[u](x) = w_*(u).
\end{aligned}
$$

□

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] John Horton Conway. *Regular Algebra and Finite Machines.* Chapman and Hall, London, 1971.

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

[4] Stephen C. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–41. Princeton University Press, Princeton, N.J., 1956.

[5] Łucja Kot and Dexter Kozen. Kleene algebra and bytecode verification. Technical Report 2004-1972, Computer Science Department, Cornell University, December 2004.

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

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