Second-Order Abstract Interpretation via Kleene Algebra

Dexter Kozen
Cornell University

AVM 2015
Attersee, Austria
4 May 2015

Joint work with
Lucja Kot
CS Department
Cornell University
Abstract Interpretation
Cousot & Cousot 79

- Static derivation of information about the execution state at various points in a program
- Comes in various flavors
  - type inference
  - dataflow analysis
  - set constraints
- Applications
  - code optimization
  - verification
  - generating proof artifacts for PCC
Standard Approach

- Start with the control flow graph of the program to be analyzed
- Propagate known information forward – possible values of variables or types
- Compute a join at confluence points
- Standard method is called the worklist algorithm
- The process is a bit like running the program on abstract values, hence the name abstract interpretation
Types or Abstract Values

- Represent sets of values
  - statically derivable
  - conservative approximation
- Form a partial semilattice
  - higher = less specific
  - join does not exist = type error
- Often, abstract values are associated with invariants
This Talk

- A general mechanism for abstract interpretation and dataflow analysis based on Kleene algebra
- May improve performance over standard worklist algorithm when the semilattice of types is small
- Illustration of the method in the context of Java bytecode verification
Kleene Algebra (KA)


\[(0 + 1(01^*0)^*1)^*\]
\[\{\text{multiples of 3 in binary}\}\]

\[(ab)^* a = a(ba)^*\]
\[\{a, aba, ababa, \ldots\}\]

\[(a + b)^* = a^*(ba^*)^*\]
\[\{\text{all strings over } \{a, b\}\}\]

Diagram of Kleene Algebra

Diagram of a,b transitions
Foundations of the Algebraic Theory


John Horton Conway (1937–)
Axioms of KA

Idempotent Semiring Axioms

\[ p + (q + r) = (p + q) + r \]
\[ p + q = q + p \]
\[ p + 0 = p \]
\[ p + p = p \]
\[ p(q + r) = pq + pr \]
\[ (p + q)r = pr + qr \]
\[ p(qr) = (pq)r \]
\[ 1p = p1 = p \]
\[ p0 = 0p = 0 \]

Axioms for \( \ast \)

\[ 1 + pp^\ast \leq p^\ast \]
\[ 1 + p^\ast p \leq p^\ast \]

\[ q + px \leq x \Rightarrow p^\ast q \leq x \]
\[ q + xp \leq x \Rightarrow qp^\ast \leq x \]
Significance of the $*$ Axioms

\[ 1 + pp^* \leq p^* \implies q + pp^* q \leq p^* q \]
\[ q + px \leq x \implies p^* q \leq x \]

$p^* q$ is the least $x$ such that $q + px \leq x$
Regular sets of strings over $\Sigma$

\[
A + B = A \cup B \\
AB = \{xy \mid x \in A, \ y \in B\} \\
A^* = \bigcup_{n \geq 0} A^n = A^0 \cup A^1 \cup A^2 \cup \ldots \\
1 = \{\epsilon\} \\
0 = \emptyset
\]

This is the free KA on generators $\Sigma$
Relational Models

Binary relations on a set $X$

For $R, S \subseteq X \times X$,

\[ R + S = R \cup S \]
\[ RS = R \circ S = \{(u, v) | \exists w (u, w) \in R, (w, v) \in S\} \]
\[ R^* = \text{reflexive transitive closure of } R \]
\[ = \bigcup_{n \geq 0} R^n = R^0 \cup R^1 \cup R^2 \cup \cdots \]
\[ 1 = \text{identity relation} = \{(u, u) | u \in X\} \]
\[ 0 = \emptyset \]

KA is complete for the equational theory of relational models
Other Models

- Trace models used in semantics
- \((\min, +)\) algebra used in shortest path algorithms
- \((\max, \cdot)\) algebra used in coding
- Convex sets used in computational geometry [Iwano & Steiglitz 90]
Matrices over a KA form a KA

\[
\begin{bmatrix}
a & b \\
c & d \\
\end{bmatrix}
+ \begin{bmatrix}
e & f \\
g & h \\
\end{bmatrix}
= \begin{bmatrix}
a + e & b + f \\
c + g & d + h \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
a & b \\
c & d \\
\end{bmatrix}
\cdot \begin{bmatrix}
e & f \\
g & h \\
\end{bmatrix}
= \begin{bmatrix}
ae + bg & af + bh \\
ce + dg & cf + dh \\
\end{bmatrix}
\]

\[
0 = \begin{bmatrix}
0 & 0 \\
0 & 0 \\
\end{bmatrix}
\quad 1 = \begin{bmatrix}
1 & 0 \\
0 & 1 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
a & b \\
c & d \\
\end{bmatrix}^* = \begin{bmatrix}
(a + bd^*c)^* & (a + bd^*c)^*bd^* \\
(d + ca^*b)^*ca^* & (d + ca^*b)^* \\
\end{bmatrix}
\]
Systems of Affine Linear Inequalities

**Theorem**

Any system of \( n \) linear inequalities in \( n \) unknowns has a unique least solution

\[
q_1 + p_{11}x_1 + p_{12}x_2 + \cdots + p_{1n}x_n \leq x_1
\]

\[
\vdots
\]

\[
q_n + p_{n1}x_1 + p_{n2}x_2 + \cdots + p_{nn}x_n \leq x_n
\]

Least solution is \( P^*q \)
Proof Artifacts

An independently verifiable representation of the proof

\( x \leq y \Rightarrow x^* \leq y^* \)

\( \lambda x, y. \lambda P0. (\text{trans}< [y=x^*; 1 \ x=x^* \ z=y^*] (\Rightarrow [x=x^* \ y=x^*; 1]) \) \\
(\text{sym} [x=x^*; 1 \ y=x^*] (\text{id.R} [x=x^*])) \) \ *R [x=x \ y=1 \ z=y^*] \\
(\text{trans}< [y=1 + y; y^* \ x=x; y^* + 1 \ z=y^*] \\
(\text{trans}< [y=y; y^* + 1 \ x=x; y^* + 1 \ z=1 + y; y^*] \\
(\text{mono+R} [x=x; y^* = y; y^* z=1] (\text{mono.R} [x=x \ y=y \ z=y^*] \ P0), \\
\Rightarrow [x=y; y^* + 1 \ y=1 + y; y^*] (\text{commut+} [x=y; y^* y=1] ))), \\
\Rightarrow [x=1 + y; y^* y=y^*] (\text{unwindL} [x=y]))) \)
Example: Java Bytecode Verification

**Java class hierarchy**

Object
- Interface
  - implements

Useless
- Integer
  - int, short, byte, boolean, char

Continuations
- Array[]
- Array[]
- Null
Example: Java Bytecode Verification

Typical bytecode instructions:

- iload 3: load an int from local 3, push on the operand stack
- istore 3: pop an int from the operand stack, store in local 3
- iadd: add the two ints on top of the stack, leave result on stack
- aload 4: load a ref from local 4, push on the operand stack
- astore 4: pop a ref from the operand stack, store in local 4
- swap: swap the two values on top of the stack (polymorphic)
Example: Java Bytecode Verification

**local variable array**

- String
- Hash-table
- p0
- p1
- p2
- Object
- this

parameters

other locals

**operand stack**

- String-Buffer
- User-Class
- int[
- maxStack

maxLocals

reference

integer

continuation

useless
A Directed Graph

- Vertices are instruction instances
- Edges to successor instructions, statically determined
  - fallthrough
  - jump targets
  - exception handlers
- Edges labeled with transfer functions
  - partial functions types \(\rightarrow\) types
  - models abstract effect of instruction
  - domain of definition gives precondition for safe execution
  - different successors may have different transfer functions
Example of a Transfer Function

- **Preconditions for safe execution**
  - local 3 is an integer
  - stack is not full

- **Effect**
  - push integer in local 3 on stack
Different exiting edges ⇒ different transfer functions

- pop object;
- pop field reference;
- push value

Object ≠ null

Object = null

- dump stack;
- push NullPointerException

- exception handler

- fallthrough instruction
Abstract Interpretation

- Annotate each vertex with a type
  - reflects best knowledge of the state immediately prior to execution of the instruction
  - must satisfy preconditions of exiting transfer functions
- Annotation of the entry instruction is determined by the declared type of the method
- Annotation of other instructions = join of values of transfer functions applied to predecessors annotations
- Want least fixpoint = best conservative approximation
Example

locals

stack

iloader 3

locals

stack

iloader 4

locals

stack

iadd

locals

stack

istore 3

locals

stack

goto
Example

locals
stack

iload 3

locals
stack

iload 4

iadd

locals
stack

istore 3

stack
locals

goto

reference

useless

integer
Example

locals stack

-iload 3

locals stack

-iload 4

locals stack

-iadd

String

locals stack

istore 3

locals stack

goto

StringBuffer

locals stack

Object

locals stack
Basic Worklist Algorithm

- Annotate entry instruction according to declared type of the method, put on worklist
  - first \( n + 1 \) locals contain this, method parameters
  - stack is empty
- Repeat until worklist is empty:
  - remove next instruction from worklist
  - for each exiting edge:
    - apply transfer function on that edge to current annotation
    - update successor annotation – join of transfer function value and current successor annotation
    - join does not exist \( \Rightarrow \) type error
    - if successor changed, put on worklist
An Application of Kleene Algebra

- Idea: avoid retracing of long cycles by symbolic composition of transfer functions
- Elements of the Kleene algebra are (typed) transfer functions
  - multiplication = typed composition
  - addition = join in the type semilattice
- Least fixpoint calculation involves computing the * of an $m \times m$ matrix, where $m$ is the size of a cutset (set of vertices breaking all cycles)
Semilattices and the ACC

- Let \((L, +, \bot)\) be a semilattice satisfying the ascending chain condition (ACC)

\[
x + (y + z) = (x + y) + z \quad x + \bot = x
\]
\[
x + y = y + x \quad x + x = x
\]

- ACC = no infinite ascending chains in \(L\)
- Implies that \(L\) contains a maximum element \(\top\)
- Elements of \(L\) represent dataflow information
  - lower = more information
  - higher = less information
  - \(\top\) = no information
A Partial Order

- There is a natural partial order
  \[ x \leq y \iff x + y = y \]
- \( x + y \) is the least upper bound of \( x \) and \( y \) with respect to \( \leq \)
Transfer Functions

- Transfer functions are modeled as strict, monotone functions $f : L \rightarrow L$
  - monotone: $x \leq y \Rightarrow f(x) \leq f(y)$
  - strict: $f(\bot) = \bot$
- Examples: $0 = \lambda x.\bot$, $1 = \lambda x.x$
- The domain of $f$ is
  \[ \text{dom } f = \{ x \in L \mid f(x) \neq \top \} \]
- monotonicity implies dom($f$) closed downward under $\leq$
Join

- Define a join operation on transfer functions:

\[ (f + g)(x) = f(x) + g(x) \]

- \( 0 = \lambda x.\bot \) is a two-sided identity for +

\[ ((\lambda x.\bot) + g)(x) = \bot + g(x) = g(x) \]

- idempotent \( f + f = f \), thus we have a natural partial order

\[ f \leq g \overset{\text{def}}{\iff} f + g = g \]

- upper semilattice with least element \( 0 = \lambda x.\bot \)
Composition

Write $f;g$ for the ordinary functional composition $g \circ f = \lambda x.g(f(x))$

- $x \in \text{dom}(f;g)$ iff $x \in \text{dom } f$ and $f(x) \in \text{dom } g$, and
  
  $$(f;g)(x) = g(f(x))$$

- $\lambda x.x$ is a two-sided identity for composition
  
  $$f;(\lambda x.x) = (\lambda x.x);f = f$$

- Composition is monotone
  
  $$f \leq g \Rightarrow f;h \leq g;h \quad f \leq g \Rightarrow h;f \leq h;g$$

- $0 = \lambda x.\bot$ is a two-sided annihilator
  
  $$(\lambda x.\bot);f = f;(\lambda x.\bot) = \lambda x.\bot$$
Distbutive Laws

Composition distributes over $+$ on the left

$$f; (g + h) = f; g + f; h$$

but not on the right; however

$$f; h + g; h \leq (f + g); h$$

due to monotonicity
Star

$f^* : L \to L$ is the function

$$f^*(x) = \text{the least } y \text{ such that } x + f(y) \leq y$$

This exists, since $f$ is monotone and the ACC holds, so the monotone sequence

$$x, \ x + f(x), \ x + f(x + f(x)), \ldots$$

converges after a finite number of steps

The convergence is not necessarily uniformly bounded in $x$

Counterexample: take $L = \mathbb{N} \cup \{\infty\}$, join = min, $f(x) = \infty$ if $x = \infty$, $x - 1$ if $x \geq 1$, and 0 if $x = 0$
We define a left-handed Kleene algebra to be a structure that satisfies all the axioms of Kleene algebra, except

- we only require the left-handed $*$ axioms and
- only right subdistributivity

Let $K$ be the set of monotone strict functions $L \to L$.

**Theorem**

The structure $(K, +, \cdot, *, 0, 1)$ is a left-handed Kleene algebra.

**Theorem**

The set of $n \times n$ matrices over a left-handed Kleene algebra with the usual matrix operations is again a left-handed Kleene algebra.
Dataflow as Matrix *

- Let $S = \{\text{vertices of the dataflow graph}\}$
- Let $E = \text{the } S \times S \text{ matrix whose } (s, t)^{\text{th}} \text{ entry is the transfer function labeling edge } (s, t)$
- Let $s_0$ be the entry point of the method, $\theta_0 \in L$ its initial label
- $E^*(s, t)$ is the join of all labels on paths from $s$ to $t$

Theorem

$E^*(s_0, t)(\theta_0)$ is the least fixpoint dataflow annotation of $t$. It is the same labeling as that produced by the worklist algorithm.
An Example

if (b) x = y + 1;
else x = z;

(if b then $\alpha$)
  iload 5  //load z
  istore 3  //save x
  goto $\beta$
else
  \(\alpha\): iload 4  //load y
  icodest 1  //load 1
  iadd
  istore 3  //save x
\(\beta\): ...

else
An Example

if (b) x = y + 1;
else x = z;

(if b then α)
iload 5 //load z
istore 3 //save x
goto β

α: iload 4 //load y
iconst 1 //load 1
iadd
istore 3 //save x

β: ...

else

then

(iload 5;
istore 3)
+

(iload 4;
iconst 1;
iadd;
istore 3)
An Example

\[ x = z; \quad \text{precondition} \quad \text{effect} \]

\[
\begin{aligned}
\text{iload 5 } & \quad 5: \text{int} & \quad \text{stack} = \text{int::\cdots}, \partial = 1 \\
\text{depth} < \text{maxStack}-1 & \\
\text{istore 3 } & \quad \text{int::stack} & \quad \partial = -1 \\
& \quad 3: \text{int}
\end{aligned}
\]
An Example

\[ x = z; \quad \text{precondition} \quad \text{effect} \]

\begin{align*}
\text{iload 5} & \quad \text{5:int} \quad \text{stack} = \text{int::\cdots}, \partial = 1 \\
& \quad \text{depth < maxStack-1} \\
\text{istore 3} & \quad \text{int::stack} \quad \partial = -1 \\
& \quad 3:\text{int} \\
\end{align*}

\text{compose}

\begin{align*}
\text{iload 5} & \quad 5:\text{int} \quad \partial = 0 \\
\text{istore 3} & \quad \text{depth < maxStack-1} \quad 3:\text{int} \\
\end{align*}
An Example

\[ x = y + 1; \quad \text{precondition} \quad \text{effect} \]

store 4 \( 4: \text{int} \) \( \text{stack} = \text{int}::\cdots, \partial = 1 \)

depth < maxStack-1

store 1 \( \text{depth} < \text{maxStack}-1 \) \( \text{stack} = \text{int}::\cdots, \partial = 1 \)

store 3 \( \text{int}::\text{stack} \) \( \partial = -1 \)

store 3 \( \text{int}::\text{stack} \) \( \partial = -1 \)

1: \text{int}
An Example

\[ x = y + 1; \quad \text{precondition} \quad \text{effect} \]

\begin{align*}
\text{iload 4} & \quad \text{4:int} \quad \text{stack} = \text{int::\cdots, } \partial = 1 \\
\text{depth} & \quad < \text{maxStack-1} \\
\text{iconst 1} & \quad \text{depth} \quad < \text{maxStack-1} \quad \text{stack} = \text{int::\cdots, } \partial = 1 \\
\text{iadd} & \quad \text{int::int::stack} \quad \partial = -1 \\
\text{istore 3} & \quad \text{int::stack} \quad \partial = -1 \\
& \quad \text{3:int} \\
\end{align*}

---

\[ \text{compose} \]

\begin{align*}
\text{iload 4} & \quad \text{4:int} \quad \partial = 0 \\
\text{iconst 1} & \quad \text{depth} \quad < \text{maxStack-2} \quad \text{3:int} \\
\text{iadd} & \\
\text{istore 3} & \\
\end{align*}
### An Example

<table>
<thead>
<tr>
<th>Precondition</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>iload 5</td>
<td>5:int</td>
</tr>
<tr>
<td>istore 3</td>
<td>depth $&lt;$ maxStack$–1$</td>
</tr>
<tr>
<td>iload 4</td>
<td>4:int</td>
</tr>
<tr>
<td>iconst 1</td>
<td>depth $&lt;$ maxStack$–2$</td>
</tr>
<tr>
<td>iadd</td>
<td></td>
</tr>
<tr>
<td>istore 3</td>
<td></td>
</tr>
</tbody>
</table>
### An Example

<table>
<thead>
<tr>
<th>Precondition</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>iload 5 5:int</td>
<td>$\partial = 0$</td>
</tr>
<tr>
<td>istore 3 depth &lt; maxStack–1</td>
<td>3:int</td>
</tr>
<tr>
<td>iload 4 4:int</td>
<td>$\partial = 0$</td>
</tr>
<tr>
<td>iconst 1 depth &lt; maxStack–2</td>
<td>3:int</td>
</tr>
<tr>
<td>iadd</td>
<td></td>
</tr>
<tr>
<td>istore 3</td>
<td></td>
</tr>
</tbody>
</table>

---

**Join**

<table>
<thead>
<tr>
<th>Precondition</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>iload 5 5:int</td>
<td>$\partial = 0$</td>
</tr>
<tr>
<td>istore 3 depth &lt; maxStack–1</td>
<td>3:int</td>
</tr>
<tr>
<td>iload 4 4:int, 5:int</td>
<td>$\partial = 0$</td>
</tr>
<tr>
<td>iconst 1 depth &lt; maxStack–2</td>
<td>3:int</td>
</tr>
<tr>
<td>iadd</td>
<td></td>
</tr>
<tr>
<td>istore 3</td>
<td></td>
</tr>
</tbody>
</table>
Dataflow as Matrix

Theorem

$E^*(s_0, t)(\theta_0)$ is the least fixpoint dataflow annotation of $t$. It is the same labeling as that produced by the worklist algorithm.

- Problem: $E$ is huge (but sparse)
- Solution: find a small cutset
A cutset (a.k.a. feedback vertex set) is a set $M$ of vertices breaking all directed cycles.

To compute the least fixpoint labeling efficiently, need to identify a small cutset.

Finding a minimal cutset is NP-complete, but polynomial time for reducible graphs.

In practice, take $M = \{\text{targets of back edges}\}$.
Partition $E$ into submatrices indexed by $M$ and $S - M$, where $M$ is the cutset.

That $M$ is a cutset is reflected algebraically by the property $D^n = 0$, where $n = |S - M|$.
Dataflow as Matrix

\[ F = (A + BD^* C)^* \]
\[ G = FBD^* \]
\[ H = D^* CF \]
\[ J = D^* + D^* CFBD^* \]
Dataflow as Matrix

- \( D^n = 0 \Rightarrow D^* = (I + D)^{n-1} \)
- The \( M \times M \) submatrix of \( E^* \) is
  \[
  (A + BD^* C)^* = (A + B(I + D)^{n-1} C)^*
  \]
- If \( s, t \) are cutpoints, the \( (s, t)^{th} \) entry of \( B(I + D)^{n-1} C \) is the join of all paths \( s \rightarrow t \) containing no other cutpoint
- Compute by repeated squaring or a variant of Dijkstra
$F = (A + B(I + D)^{n-1}C)^*$ is much smaller than $E$

The other submatrices of $E^*$ can be described in terms of this matrix

\[
G = FBD^*
\]
\[
H = D^* CF
\]
\[
J = D^* + HG
\]
Finding Small Cutsets

Efficiency depends on finding a small cutset = set of nodes intersecting every directed cycle

- finding a minimum cutset is $NP$-complete
- Ptime for reducible graphs [Garey & Johnson 79]
- bytecode programs compiled from Java source are typically reducible
- in practice, take targets of back edges

How big are cutsets in practice?

- analyzed 537 Java programs
- median cutset size = 2.1% of total program size
- all except 5 programs < 5%
- largest program analyzed was 2668 instructions with 5 cutpoints = 0.2%
Finding Small Cutsets

Efficiency depends on finding a small \textit{cutset} = set of nodes intersecting every directed cycle

- finding a minimum cutset is $\text{NP}$-complete
- Ptime for reducible graphs [Garey & Johnson 79]
- bytecode programs compiled from Java source are typically reducible
- in practice, take targets of back edges

How big are cutsets in practice?

- analyzed 537 Java programs
- median cutset size = 2.1% of total program size
- all except 5 programs < 5%
- largest program analyzed was 2668 instructions with 5 cutpoints = 0.2%
A Pipe Dream

Many instructions have preconditions for safe execution (e.g., array, pointer dereference). Compilers should either:

- insert a runtime type check, or
- optimize away the check, but provide a proof of correctness of the optimization

Programmer should be able to specify such preconditions, and they should behave the same way as the built-in ones
if (h.containsKey(key)) {
    data = h.get(key);
} else {
    data = new Data();
    h.put(key, data);
}

data = h.get(key);
if (data == null) {
    data = new Data();
    h.put(key, data);
}

data = h.get(key);
if (h.containsKey(key)) {
    data = h.get(key);
} else {
    data = new Data();
    h.put(key, data);
}

data = h.get(key);
if (data == null) {
    data = new Data();
    h.put(key, data);
}

assert h.containsKey(key);
data = h.get(key);
Built-in Preconditions

\[
x = \text{obj.data};
\]

\[
x = a[i];
\]

Compiler will either
- omit runtime check but supply a proof, or
- insert runtime check and throw exception on failure
  (NullPointerException or ArrayIndexOutOfBoundsException, resp.)
assert obj != null;
x = obj.data;

assert 0 <= i && i < a.length;
x = a[i];

Compiler will either

- omit runtime check but supply a proof, or
- insert runtime check and throw exception on failure
  (NullPointerException or ArrayIndexOutOfBoundsException, resp.)
assert h.containsKey(key);
data = h.get(key);

Compiler will either

- omit runtime check but supply a proof, or
- insert runtime check and throw InvalidAssertionException on failure
Conclusion

Summary

- A general mechanism for second-order abstract interpretation based on Kleene algebra
  - may improve performance over standard worklist algorithm when the semilattice of types is small - $O(m^3 + nm)$ vs $O(nd)$
- Proved soundness and completeness of the method
- Illustrated the method in the context of Java bytecode verification

Possible next steps

- Implement and compare experimentally to the standard worklist algorithm as specified in the Java VM specification
- Second-order method is amenable to parallelization, whereas the standard worklist method is inherently sequential
  - application of a transfer function requires knowledge of its inputs
  - compositions can be computed without knowing their inputs
Thanks!