Static Analysis

Ralf Huuck

Introduction

What is Static Analysis?

Static Analysis subsumes all methods which derive information from programs without actually executing them.

Static analysis is the term applied to the analysis of computer software that is performed without actually executing programs.

- wikipedia
The state space of programs is in theory infinite. Computation depends on
- integers
- reals
- etc.

In reality it is finite, e.g., 32-bit representations, which is still practically infinite when exploring all combinations.

Decidability

Properties on infinite state spaces are typically undecidable, i.e., there is no general algorithm to decide if they are true or false.

Rice's theorem: Any nontrivial property about the language recognized by a Turing machine is undecidable.

But we can still attempt to give useful approximate solutions.
Approximations

false negative, no violation detected due to under approx.

Definitions

Lattices

partial order: \( L=(S, \sqsubseteq) \), set \( S \), binary relation \( \sqsubseteq \) s.t.
- reflexive: \( \forall x \in S : x \sqsubseteq x \)
- transitive: \( \forall x,y,z \in S : x \sqsubseteq y \land y \sqsubseteq z \Rightarrow x \sqsubseteq z \)
- anti-symmetric: \( \forall x,y \in S : x \sqsubseteq y \land y \sqsubseteq x \Rightarrow x=y \)

% y upper bound of \( X \subseteq S \) (\( X \sqsubseteq y \)) if \( \forall x \in X : x \sqsubseteq y \)
% least upper bound \( \sqcup X \) if \( \forall y \in S : X \sqsubseteq y \Rightarrow \sqcup X \sqsubseteq y \)

Conversely, define lower bound and greatest lower bound.

Lattice is partial order where \( \sqcup X \) and \( \sqcap X \) exists for all \( X \subseteq S \).

Example: Finite Lattices

Lattices:

But not:

Show: every lattice has a unique largest (\( \top \)) and smallest (\( \bot \)) element.
Monotone functions

**f monotone:** \( \forall x, y \in S: x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y) \)

**Note:** compositions of monotone functions are monotone

**Theorem:** In lattice \( L \) with finite height every monotone function has a least fixed-point as:

\[
\text{fix}(f) = \bigcup \{ x \mid f(x) = x \}
\]

for which \( f(\text{fix}(f)) = \text{fix}(f) \).

**Closure**

If \( L_1, \ldots, L_n \) lattices of finite height, so is the product

\[
L_1 \times \cdots \times L_n = \{ (x_1, \ldots, x_n) \mid x_i \in L_i \}
\]

where \( \sqsubseteq \) is defined pointwise.

The height of the product is the sum of heights of its components.

Map: finite set \( A \), lattice \( L \) with \( |A| \leq |L| \) then with a pointwise order

\[
\text{Map}: L \mapsto L = \{ (a_1, \ldots, a_n) \mid a_i \in L \}
\]

is a lattice of height \( |A| \times \text{height}(L) \).

Other compositions: +, lift, flat, are also lattices of finite height again.

**Equation Systems (ES)**

Equation System \( F: L^n \to L^n \)

\[
\begin{align*}
x_1 &= F_1(x_1, \ldots, x_n) \\
x_2 &= F_2(x_1, \ldots, x_n) \\
\vdots & \quad \vdots \\
x_n &= F_n(x_1, \ldots, x_n)
\end{align*}
\]

where \( x \) variables and \( F_i: L^n \to L \) collection of monotone functions

has a least fixed point for the function

\[
F(x_1, \ldots, x_n) = (F_1(x_1, \ldots, x_n), F_2(x_1, \ldots, x_n), \ldots, F_n(x_1, \ldots, x_n))
\]

**Inequation System (IS)**

Inequation System \( F: L^n \to L^n \)

\[
\begin{align*}
x_1 &\subseteq F_1(x_1, \ldots, x_n) \\
x_2 &\subseteq F_2(x_1, \ldots, x_n) \\
\vdots & \quad \vdots \\
x_n &\subseteq F_n(x_1, \ldots, x_n)
\end{align*}
\]

We can show \( x \sqsubseteq y \Leftrightarrow x = x \cup y \).

Thus, IS \( F \) equivalent to following ES:

\[
\begin{align*}
x_1 &= x_1 \cap F_1(x_1, \ldots, x_n) \\
x_2 &= x_2 \cap F_2(x_1, \ldots, x_n) \\
\vdots & \quad \vdots \\
x_n &= x_n \cap F_n(x_1, \ldots, x_n)
\end{align*}
\]
A **control flow graph** is a directed graph where nodes are program points and the edges represent the flow of control between these points.

### Example CFG

```plaintext
fun(n) {
    var f;
    f=1;
    while (n>0) {
        f=f*n;
        n=n-1
    }
    return f;
}
```

**Data Flow Analysis**

Classical **Data Flow Analysis** is concerned about which data reaches which program point. The analysis is performed on program’s CFG and expressed as an (in)equality system over a finite lattice.

Typical examples are:
- liveness
- available expressions
- very busy expressions
- reaching definitions

We will go through each of them.
A variable is **live** before a program point if it will be read in the remaining program execution without being written first.

```java
fun(n) {
    var f;
    f=1;
    while (n>0) {
        f=f*n;
        n=n-1
    }
    return f;
}
```

Which variables are live at which locations? Which are not?

Any analysis must find out the set of live variables for each location. We model the domain of live variables by the lattice \((\mathcal{2}^{\text{Vars}}, \subseteq)\).

```java
fun(n) {
    var f;
    f=1;
    while (n>0) {
        f=f*n;
        n=n-1
    }
    return f;
}
```

How does the lattice \((\mathcal{2}^{\text{Vars}}, \subseteq)\) look like? Why is it a lattice?

**Vars** is the set of variables occurring in a program.

---

**Questions**

- Can we approximate (without executing the program) the set of live variables algorithmically?
- How does the set depend on our syntax?
- Can we define rules for each construct?
- How can rules lead to something that we can compute?

---

**Question**

Let \(\text{live}(v)\) denote the set of variables live before location \(v\).

Can we define rules just depending on the current node and its predecessors or successors?
Equation System

let \( \text{live}(v) \) denote the set of variables live before location \( v \).

\[
\begin{align*}
\text{live}(1) &= \text{live}(2) \\
\text{live}(2) &= \text{live}(3) \setminus \{f\} \\
\text{live}(3) &= (\text{live}(4) \cup \text{live}(6)) \cup \{n\} \\
\text{live}(4) &= (\text{live}(5) \setminus \{f\}) \cup \{f, n\} \\
\text{live}(5) &= (\text{live}(3) \setminus \{n\}) \cup \{n\} \\
\text{live}(6) &= \{\}
\end{align*}
\]

Gen/Kill

let \( \text{live}(v) \) denote the set of variables live before location \( v \).

\[
\begin{align*}
\text{live}(1) &= \text{live}(2) \\
\text{live}(2) &= \text{live}(3) \setminus \{f\} \\
\text{live}(3) &= (\text{live}(4) \cup \text{live}(6)) \cup \{n\} \\
\text{live}(4) &= (\text{live}(5) \setminus \{f\}) \cup \{f, n\} \\
\text{live}(5) &= (\text{live}(3) \setminus \{n\}) \cup \{n\} \\
\text{live}(6) &= \{\}
\end{align*}
\]

Join

let \( \text{live}(v) \) denote the set of variables live before location \( v \).

use the abbreviation
\[
\text{join}(v) = \bigcup_{w \in \text{succ}(v)} \text{live}(w)
\]

Join
let \( \text{live}(v) \) denote the set of variables live before location \( v \).

\[
\begin{align*}
\text{live}(1) & = \text{join}(1) \setminus \{f\} \\
\text{live}(2) & = \text{join}(2) \setminus \{f\} \\
\text{live}(3) & = \text{join}(3) \cup \{n\} \\
\text{live}(4) & = (\text{join}(4) \setminus \{f\}) \cup \{f, n\} \\
\text{live}(5) & = (\text{join}(3) \setminus \{n\}) \cup \{n\} \\
\text{live}(6) & = \{\}
\end{align*}
\]

The right hand-side of each equation is monotone, i.e., we can compute the fixed point of ES.

We are interested in the least fixed point.

Least fixed point: start with \( \{\} \) greatest start with Vars.

The right hand-side of each equation is monotone, i.e., we can compute the fixed point of ES.

We are interested in the least fixed point.

Least fixed point: start with \( \{\} \) greatest start with Vars.

How/in which order to we iterate through the equations?

- following equation order
- randomly (requires fairness)
- worklists/priority queues
- strongly connected components (program topology)
- Gauß-Seidel elimination
- etc. etc.

There is no general best way, but some are better than others.
Intermediate Summary

Seen so far:
• data flow analysis problem can be expressed in terms of fixed point over equations
• equations depend on syntax of program points and what is coming in/out
• many ways to compute fixed point

We have seen Join as being union of successors and computation of least fixed point. This is not always so ...

Available Expressions

An expression is available after a program point if its current value has been evaluated before and none of its variables are overwritten. (Good for optimizations)

fun(n) {
  var f;
  f=1;
  while (n>0) {
    f=f*n;
    n=n-1
  }
  return f;
}

Which expressions are available at which locations? Which are not?

Available Expressions

Our analysis must find out the set of available expressions for each location. We model the domain of available expression by the lattice $(2^{Expr}, \subseteq)$.

fun(n) {
  var f;
  f=1;
  while (n>0) {
    f=f*n;
    n=n-1
  }
  return f;
}

How does the lattice $(2^{Expr}, \subseteq)$ look like? Why is it a lattice?

Expr is the set of expressions occurring in a program.

Equation System

let avail(v) denote the set of expressions available after location $v$:

avail(1) = {} 

avail(2) = (avail(1) \ {f*n}) \cup {1} 

avail(3) = (avail(2) \cap avail(5)) \cup {n>0} 

avail(4) = (avail(3) \ {f*n}) \cup {f*n} 

avail(5) = (avail(4) \ {n-1}) \cup {n>0} 

avail(6) = avail(3)
Join/Fixed Point

let \( \text{avail}(v) \) denote the set of variables available after location \( v \).

We can use the abbreviation \( \text{join}(v) = \bigcup_{w \in \text{pred}(v)} \text{avail}(w) \) to see we have again a monotone framework.

This time we compute the greatest fixed point, as we like to have the maximum number of available expressions.

Very Busy Expressions

An expression is **very busy** before a program point if it definitely will be evaluated again before its value changes.

```plaintext
fun(n) {
    var f;
    f=1;
    while (n>0) {
        f=f*n;
        n=n-1
    }
    return f;
}
```

Which expressions are very busy at which locations? Which are not?

Greatest Fixed Point Solution

let \( \text{avail}(v) \) denote the set of expressions available after location \( v \).

\[
\begin{align*}
\text{avail}(1) &= \emptyset \\
\text{avail}(2) &= \{1\} \\
\text{avail}(3) &= \{n>0\} \\
\text{avail}(4) &= \{n>0, f*n\} \\
\text{avail}(5) &= \{f*n, n-1\} \\
\text{avail}(6) &= \{n>0\}
\end{align*}
\]

Very Busy Expressions

Our analysis must find out the set of very busy expressions for each location. We model the domain of very busy expression by the lattice \( (2^{\text{Expr}}, \subseteq) \).

```plaintext
fun(n) {
    var f;
    f=1;
    while (n>0) {
        f=f*n;
        n=n-1
    }
    return f;
}
```

How does the lattice \( (2^{\text{Expr}}, \subseteq) \) look like? Why is it a lattice?

Expr is the set of expressions occurring in a program.
Equation System

let busy(v) denote the set of expressions very busy before location v.

\[ \begin{align*}
busy(1) &= \text{busy}(2) \\
busy(2) &= \left( \text{busy}(3) \setminus \{f \cdot n\} \right) \cup \{1\} \\
busy(3) &= \left( \text{busy}(2) \cap \text{busy}(5) \right) \cup \{n \geq 0\} \\
busy(4) &= \left( \text{busy}(5) \setminus \{f \cdot n\} \right) \cup \{f \cdot n\} \\
busy(5) &= \left( \text{busy}(3) \setminus \{n-1\} \right) \cup \{n-1\} \\
busy(6) &= \{\} 
\end{align*} \]

Join/Fixed Point

let busy(v) denote the set of variables very busy before location v.

We can use the abbreviation \(\text{join}(v) = \bigcup_{w \in \text{succ}(v)} \text{busy}(w)\) to see we have again a monotone framework.

This time we compute the least fixed point again.

Least Fixed Point Solution

let busy(v) denote the set of expressions very busy before location v.

\[ \begin{align*}
busy(1) &= \{n \geq 0, 1\} \\
busy(2) &= \{n \geq 0, 1\} \\
busy(3) &= \{n \geq 0\} \\
busy(4) &= \{n \geq 0, n-1, f \cdot n\} \\
busy(5) &= \{n \geq 0, n-1\} \\
busy(6) &= \{\} 
\end{align*} \]

Reaching Definition

The effect of an assignment is reaching after a program point if it might have defined the current values of variables.

Which assignments are reaching at which locations? Which are not?
Our analysis must find out the set of reaching assignments for each location. We model the domain of reaching definitions by the lattice \( \mathcal{L} \).

**Fun(n) Function**

```plaintext
fun(n) {
    var f;
    f=1;
    while (n>0) {
        f=f*n;
        n=n-1
    }
    return f;
}
```

How does the lattice \( \mathcal{L} \) look like? Why is it a lattice?

Assign is the set of assignments occurring in a program.

**Equation System**

\[ \text{reach}(v) \text{ denote the set of assignments may define variable values after location } v. \]

\[ \text{reach}(1) = \{ f=1 \} \]

\[ \text{reach}(2) = \text{reach}(1) \setminus \{ f=1 \} \cup \{ f=1 \} \]

\[ \text{reach}(3) = \text{reach}(2) \cup \text{reach}(5) \]

\[ \text{reach}(4) = \text{reach}(3) \setminus \{ f=f*n \} \cup \{ f=f*n \} \]

\[ \text{reach}(5) = \text{reach}(4) \cup \{ n=n-1 \} \cup \{ n=n-1 \} \]

\[ \text{reach}(6) = \text{reach}(3) \]

**Join/Fixed Point**

We can use the abbreviation \( \text{join}(v) = \bigcup_{w \in \text{pred}(v)} \text{reach}(w) \) to see we have again a monotone framework.

**Least Fixed Point Solution**

This time we compute the least fixed point again.
Summary Data Flow Analysis

- we have seen how to compute approximate solutions (we do not know if all the paths are executed!) to data flow problems
- Join can be intersection or union
- analysis has forward or backward nature
- depending on the problem least or greatest fixed point

Rules of Thumb

- **forward analysis:** computes information about past behavior
- **backward analysis:** computes information about future behavior
- **must analysis:** information that must be true (on all paths) and computes under-approximation
- **may analysis:** information that may be true (on at least one paths) and computes over-approximation

Abstract Interpretation

The Rough Guide

The idea

Abstract Interpretation (AI) provides appropriate means to relate some concrete world (domain) with an abstract world.

The applications and implications

Far reaching, ranging from comparing program semantics to program analysis.

Origin

First developed by Cousot & Cousot (1977/78).
Abstract Interpretation (simplified)

AI is more than abstraction as it relates abstract with concrete world and vice versa.

We require:
1. ∀x: abs(x) ⊆ y ⇔ x ⊆ conc(y)
   (Galois connection)

Operations should be safely approximated:
2. abs ◦ c-op ◦ conc ◦ a-op

Example (Interval AI)
1. Structure
Sets of integers
- (2)
- (2,3,4)
- (1,3,9)

Intervals
- [2,2]
- [2,4]
- [1,9]

x∈ Z^{m^2} smallest interval comprising x

Intervals over approximate sets.

2. Operations
Concrete world: (2,5,6) + (2,3) = (4,7,8,9)
Abstract world: [2.6] + [2.3] = [4.9]
a) Introduce matching operator in abstract world for every operator in concrete world.
b) Check it satisfies safe approximation.
We like to compute all the possible values variables can take (collecting semantics).

```javascript
var f, n;
n=3;
f=1;
while (n>0) {
    f=f*n;
    n=n-1;
}
return f;
```

before each location the variables might be as follows:

1. We might need infinite space to store values.
2. We might not be able to compute them due to non-termination.

Solution:

1. can be overcome by using interval AI (over-approximation of values)
Problems:
1. We might need infinite space to store values.
2. We might not be able to compute them due to non-termination.

Solution:
1. Can be overcome by using interval AI (over-approximation of all values)
2. Not that easy: interval lattice has infinite width (not a problem) and infinite height (problem!).

Application

<table>
<thead>
<tr>
<th>var f,n</th>
</tr>
</thead>
<tbody>
<tr>
<td>n=3</td>
</tr>
<tr>
<td>f=1</td>
</tr>
<tr>
<td>n=1</td>
</tr>
<tr>
<td>n&lt;0</td>
</tr>
<tr>
<td>f=f*n</td>
</tr>
<tr>
<td>n=n-1</td>
</tr>
</tbody>
</table>

Acceleration

AI:
- We have relation between abstract and concrete domain.
- We have relation between abstract and concrete operators.

In order to deal with infinite lattices (i.e. to compute a fixed point in finite time) we introduce an extra operator that can “jump” infinitely high.

Such an acceleration operator is called widening operator. Sometimes people speak of dynamic approximation operator.

Result:

Application

<table>
<thead>
<tr>
<th>var f,n</th>
</tr>
</thead>
<tbody>
<tr>
<td>n=3</td>
</tr>
<tr>
<td>f=1</td>
</tr>
<tr>
<td>n=1</td>
</tr>
<tr>
<td>n&lt;0</td>
</tr>
<tr>
<td>f=f*n</td>
</tr>
<tr>
<td>n=n-1</td>
</tr>
</tbody>
</table>

Idea:
1. If variable value increased compared to previous iteration jump to $+\infty$, if decreased to $-\infty$.
2. Goto 2

Since our operations are monotone this is safe.
Application

Result:

1: var f,n

2: n=3

3: f=1

4: n>0

5: f=f*n

6: n=n-1

7

Application

Result:

1: var f,n

2: n=3

3: f=1

4: n>0

5: f=f*n

6: n=n-1

7

Summary

- AI relates abstract and concrete worlds (structure + operations)
- termination/safe approximation can be enforced by acceleration techniques

There are domains that capture more information, e.g., polyhedra.

AI is good for range approximation, i.e., array access, range check of operations, general buffer overflows.

Model Checking Syntax

Just some ideas ...
Introduction

Syntax gives us some information:
• when a variable is used
• when a variable is declared
• when a variable is modified
• etc.

Can we make use of it to find bugs in programs?

Syntactical Information

```
var f, n
f=1
n>0
f=f*n
n=n-1
```

1. declaration_f declaration_n
2. modified_n
3. modified_f
4. used_n
5. used_n used_f modified_f
6. used_n used_f modified_f
7. used_n modified_n

CFG is a transition system

So can we model check syntax?
Model Checking Syntax

```
1 var f,n
2 n=3
3 i=1
4 r>0
5 f=n
6 n=n-1
```

Yes, e.g.:
AG (declaration₁ → EF used₁)
AG (modified₁ → EF used₁)
also if variables are initialized, certain protocols are respected, locks are released etc.

But: Abstraction is sometimes neither sound nor complete.

Summary

- model checking syntax is good for finding bugs
- not so good for showing the absence of bugs/verification
- very efficient
- easy to use

Lessons learnt

- “looking” at programs without executing them can reveal a lot of information.
- different static analysis techniques
- application depends on purpose
- combinations with semantic analysis possible (constraint solving, model checking semantics)
Next Week

Model Checking Real-Time Systems