THEORY of COMPILATION
LECTURE 08

Static Analysis

You are here

Compiler

Executable code

Up Until Now

AST

IR

Asm

9

(1) t = A – B
(2) u = A – C
(3) v = t + u
(4) A = D
(5) D = v + u

LD R1, @A
LD R2, @B
SUB R2,R1,R2
LD R3, @C
SUB R1,R1,R3
ADD R3,R2,R1
LD R2, @D
ADD R1,R3,R1
ST @A, R2
ST @D, R1

Syntax directed translation
Register allocation
Instruction selection

Today

• Dataflow analysis
• Lattices
• Chaotic Iterations
• Monotone framework (for dataflow analysis)
• A few example analyses

Static Analysis

“The algorithmic discovery of properties of a program by inspection of its source text”

— Manna, Pnueli

Reason statically — at compile time — about the possible runtime behaviors of a program
• Does not have to literally be the source text, just means w/o running it
• In a compiler, we mostly use IR
Static Analysis

- What for...

x = ?
if (x > 0) {
    y = 42;
} else {
    y = 73;
    foo();
}
assert (y == 42);

- Bad news: problem is generally undecidable

Central idea: use approximation

Over-Approximation

Conservative static analysis: assertion may be violated

Precision

/* My Awesome Static Analyzer */
main(...) {
    printf("assertion may be violated\n");
}

- Lose precision only when required
- Understand where precision is lost
Static Analysis

- Formalize software behavior using a mathematical model (semantics)
- Prove properties of the mathematical model
  - Automatically, typically with approximation of the formal semantics
- Develop theory and tools for program correctness and robustness

Static Analysis

- Spans a wide range from type checking to full functional verification
  - General safety specifications
  - Absence of resource leaks
  - Concurrency correctness conditions (e.g., progress, race-freedom)
  - Correct use of libraries (e.g., initialization)
- Under-approximations useful for bug-finding, test-case generation, ...

Static Analysis: Techniques

- Dataflow analysis
- Constraint-based analysis
- Type and effect systems
- Abstract Interpretation
  - ...

Example: Reaching Definitions

- Concept of definition and use:
  - \( x = y + z \)
    - is a definition of \( x \)
    - is a use of \( y \) and \( z \)
- A definition reaches a use if
  - value written by definition...
  - ...may be read by use

Example: Reaching Definitions

```
1 y := x
2 z := 1
3 while (y > 0) {
    4 y := y - 1
    5    y := z
} 6 y := 0
7 return y + z
```

(adapted from Nielson, Nielson & Hankin)
Example: Reaching Definitions

```
1. y := x
2. z := 1
3. while (y > 0) {
   4.   z := z * y
   5.   y := y – 1
4. }
6. y := 0
7. return y + z
```

(adapted from Nielson, Nielson & Hankin)

---

Dataflow Analysis: Overview

```
while (y > 0) {
   x := y
   y := y – 1
}
return y + z
```

---

Control-Flow Graph

```
1. y := x
2. z := 1
3. while (y > 0) {
   4.   z := z * y
   5.   y := y – 1
4. }
6. y := 0
7. return y + z
```

---

Transfer Functions

Given a program statement S, we can define a transfer function $T_S$ that relates the properties that are true before the statement to the properties that are true after the statement.
Partial Orders

- Set P
- Binary relation ⊑ such that ∀x,y,z ∈ P:
  - x ⊑ x (reflexive)
  - x ⊑ y and y ⊑ z implies x ⊑ z (transitive)
- Can use partial order to define:
  - Upper and lower bounds
  - Least upper bound
  - Greatest lower bound

Upper Bounds

- For S ⊆ P:
  - x ∈ P is an upper bound of S if ∀y ∈ S, y ⊑ x
  - x ∈ P is the least upper bound of S if
    - x is an upper bound of S, and
    - x ⊑ y for all upper bounds y of S
  - ⊔ – join, least upper bound, lub, supremum, sup
  - ⊔ S is the least upper bound of S
  - x ⊔ y = ⊔{x,y}
  - Often written as ∨ as well

Lower Bounds

- For S ⊆ P:
  - x ∈ P is a lower bound of S if ∀y ∈ S, x ⊑ y
  - x ∈ P is the greatest lower bound of S if
    - x is a lower bound of S, and
    - y ⊑ x for all greatest lower bounds y of S
  - ⊓ – meet, greatest lower bound, glb, infimum, inf
  - ⊓ S is the greatest lower bound of S
  - x ⊓ y = ⊓{x,y}
  - Often written as ∧ as well

Covering

- x ⊏ y if x ⊑ y and x ≠ y
- x is covered by y (y covers x) if
  - x ⊏ y, and
  - no z such that x ⊏ z ⊏ y
- Conceptually,
  - y covers x if there are no elements between x and y
Lattices

- If \( x \sqcup y \) and \( x \sqcap y \) exist for all \( x, y \in P \)
  then \( P \) is a lattice
- If \( \sqcup S \) and \( \sqcap S \) exist for all \( S \subseteq P \)
  then \( P \) is a complete lattice
- **Theorem:** all finite lattices are complete.
- Example of a lattice that is not complete:
  - Integers \( \mathbb{Z} \)
  - \( \sqcup = \max \), \( \sqcap = \min \)
  - But \( \sqcup \mathbb{Z} \) and \( \sqcap \mathbb{Z} \) do not exist \( \Rightarrow \) not complete
  - Conversely, \( \mathbb{Z} \cup \{+\infty, -\infty\} \) is a complete lattice

Example

- \( P = \{000, 001, 010, 011, 100, 101, 110, 111\} \)
- \( x \subseteq y \iff (x \& y) = x \) where \( \& \) is bitwise 'and'
  
  ![Hasse Diagram](image)

  If \( y \) covers \( x \):
  - Line from \( y \) to \( x \)
  - \( y \) above \( x \) in diagram

Top and Bottom

- Greatest element of \( P \) (if it exists) is top (\( \top \))
- Least element of \( P \) (if it exists) is bottom (\( \bot \))

\[ \top = \sqcup P \quad \bot = \sqcap P \]

Product Lattices

- Given two lattices \( L \) and \( Q \), the product can easily be made a lattice
  \[ (l_1, q_1) \subseteq (l_2, q_2) \iff l_1 \subseteq l_2 \text{ and } q_1 \subseteq q_2 \]

For vectors of \( L \), defining a lattice is also easy

\[ (l_1, l_2, \ldots, l_n) \subseteq (q_1, q_2, \ldots, q_n) \iff \forall i (l_i \subseteq q_i) \]

Lattices of Program Properties

- Properties of interest can often be arranged into a lattice
- **Example:** Lattices of values –

  ![Lattices of Program Properties](image)

  - When the value of each variable is a lattice, the state of the program is a product lattice of the states of all variables.
A lattice of predicates
\[ \langle x = \{ \perp, \text{even}, \text{odd} \}, \langle y = \{ \perp, \text{even}, \text{odd} \} \rangle \]

\[ \text{e.g. } \langle x = \text{even}, y = \text{odd} \rangle \sqsubseteq \langle x = \perp, y = \text{odd} \rangle \sqsubseteq \langle x = \perp, y = \perp \rangle \]

Example

```javascript
x := 0;
y := 6;
while (x < 10) {
x := x + 2;
y := y + x;
}
assert (y is even);
```

Product lattice of two individual lattices, one per variable

Lattices of program properties

- Lattice does not have to carry a direct relationship to program values
  - Example: Can an object escape from a function?

```
can-escape
```

Computing the Transfer Function

- We must hard-code a transfer function specific to the lattice
  - Occasionally, there would be a trade-off between how precise the transfer functions are and how easy it is to compute them
- We can build lattices for arbitrary facts about the program
  - Need to make sure our transfer functions are "well behaved" (we will define "good" behavior later)

From CFG to Equations

- For every block, define state variables in and out
  - If \( i \) is the only predecessor of \( j \):
    - \( \text{in}_j = \text{out}_i \)
  - Use join (\( \sqcup \)) when multiple edges enter the same block:
    - \( \text{out}_j = \text{out}_i \sqcup \text{out}_k \)
  - \( y \) is even

```
return y
```

From CFG to Equations

- For every block, define state variables in and out
  - \( \text{out}_i = T_i(\text{in}_i) \)
  - \( \text{in}_j = (x = v_1, y = v_2) \)
  - \( \text{out}_j = (x = v_1, y = v_2) \)
  - \( y := y + 1 \)
  - odd = even
  - even = odd
We define the following transfer function:

\[ \text{out}_i = \text{in}_i \setminus \{x,*,i\} \cup \{(x,i)\} \]

where

- \( x \) is the variable assigned to in \( i \)
- \( \{x,*,i\} = \{(x,l) | l \in \text{Lab}\} \)

Lab = set of all statement labels

\[ k: \text{statement...} \]

\[ j: \text{return y} \]

\[ i: \text{y := x} \]

\[ ii: \text{z := 1} \]

\[ iii: \text{y > 0} \]

\[ iv: \text{z := z \ast y} \]

\[ v: \text{y := y – 1} \]

\[ vi: \text{y := 0} \]

\[ vii: \text{return y + z} \]

We define the following transfer function:

\[ \text{out}_i = \text{in}_i \setminus \{x,*,i\} \cup \{(x,i)\} \]

where

- \( x \) is the variable assigned to in \( i \)
- \( \{x,*,i\} = \{(x,l) | l \in \text{Lab}\} \)

Lab = set of all statement labels

\[ k: \text{...statement...} \]

\[ j: \text{return y} \]

\[ i: \text{y := x} \]

\[ ii: \text{z := 1} \]

\[ iii: \text{y > 0} \]

\[ iv: \text{z := z \ast y} \]

\[ v: \text{y := y – 1} \]

\[ vi: \text{y := 0} \]

\[ vii: \text{return y + z} \]

For every program point, we compute the set of variable definitions that reach it.

\[ L = \mathcal{P} (\text{Var} \times \text{Lab}) \]

\[ \{ (x,1) \} \]

\[ \{ (x,2) \} \]

\[ \vdots \]

\[ \{ (x,n) \} \]

\[ \{ (y,1) \} \]

\[ \{ (y,2) \} \]

\[ \vdots \]

\[ \{ (y,n) \} \]

This is called a power set lattice.

\[ \subseteq \]

\[ \cup \]

\[ ∪ \]

\[ \{ (x,1), (x,2) \} \]

\[ \{ (x,1), (y,1) \} \]

\[ \{ (x,1), (y,n) \} \]

\[ \vdots \]

\[ \{ (x,n), (y,n) \} \]

\[ \{ (x,n), (y,n–1) \} \]

\[ \vdots \]

\[ \{ (x,n), (y,n) \} \]

\[ \{ y+y+z \} \]

\[ y := x \]

\[ z := 1 \]

\[ y > 0 \]

\[ z := z \ast y \]

\[ y := y – 1 \]

\[ y := 0 \]

\[ return y + z \]

\[ \{ (y,1), (y,2) \} \]

\[ \{ (y,1), (y,n) \} \]

\[ \{ (y,n), (y,n) \} \]

\[ \{ (y,n), (y,n–1) \} \]

\[ \vdots \]

\[ \{ (y,n), (y,n) \} \]

\[ \{ y+y+z \} \]

\[ y := x \]

\[ z := 1 \]

\[ y > 0 \]

\[ z := z \ast y \]

\[ y := y – 1 \]

\[ y := 0 \]

\[ return y + z \]

\[ \{ (x,1) \} \]

\[ \{ (x,2) \} \]

\[ \vdots \]

\[ \{ (x,n) \} \]

\[ \{ (y,1) \} \]

\[ \{ (y,2) \} \]

\[ \vdots \]

\[ \{ (y,n) \} \]

\[ \{ y+y+z \} \]
System of Equations

These equations define a function over 13 variables (in \(1..7\), out \(1..6\)).

Each variable represents a value from our lattice, \(\mathcal{P}(\text{Var} \times \text{Lab})\).

\[
\langle \emptyset, \quad v_8 v_9 v_{12} v_10 v_11 v_10 v_13 v_1 \ (y,*), \quad \cup \{ (y,1) \} \\
\quad v_2 \ (z,*), \quad \cup \{ (z,2) \} \\
\quad v_3 v_4 \ (z,*), \quad \cup \{ (z,4) \} \\
\quad v_5 \ (y,*), \quad \cup \{ (y,5) \} \\
\quad v_6 \ (y,*), \quad \cup \{ (y,6) \} \rangle
\]

A solution \(v\) satisfies \(F(v) = v\).

Solving the Equations

- Fixed Point Problem
  - Given a function \(F: L \rightarrow L\), find \(x \in L\) such that \(F(x) = x\).
  - With transfer functions, you will commonly find that \(\top\) is one such solution...
  - We would like the most precise solution

Knaster-Tarski Theorem

- Order preserving (monotonic) function:
  \(x \sqsubseteq y \Rightarrow F(x) \sqsubseteq F(y)\)

- Let \(L\) be a complete lattice and \(F: L \rightarrow L\) a monotonic function. Then the set of fixed points of \(F\) is also a complete lattice.

- **Definition.** the least fixed point \(x_\bot\) is a fixed point (\(F(x_\bot) = x_\bot\)), such that for any \(x\), if \(F(x) = x\), then \(x_\bot \sqsubseteq x\).

Kleene Fixed-point Theorem

- Order preserving (monotonic) function:
  \(x \sqsubseteq y \Rightarrow F(x) \sqsubseteq F(y)\)

Let \(x_\bot\) be its least fixed point.

- In particular, \(F(x_\bot) = x_\bot\).

- We build a chain \(x_0, x_1, \ldots\) by: \(x_0 = \bot\) and \(x_{i+1} = F(x_i)\)
  - by induction, \(x_i \sqsubseteq x_{i+1}\)
  - also, \(x_i \sqsubseteq x_{i+1}\)
  - if for some \(i, x_i = x_{i+1}\) \(x_i\) is a fixed point \(x_i \sqsubseteq x_{i+1} \sqsubseteq x_i \Rightarrow x_i = x_{i+1}\)

Same trick works for computing greatest fixed point

- in that case, start with \(x_0 = \top\)

Chains

- A set \(S \subseteq L\) is a chain if
  \(\forall x, y \in S. y \sqsubseteq x \lor x \sqsubseteq y\)

- \(L\) has no infinite chains if every chain in \(L\) is finite.

- In that case, we are guaranteed to find the least fixed point in a finite number of steps.
Least Fixed Point Solution

Chaotic Iterations
- To avoid recomputing values that do not change:
  - Keep a work list of CFG nodes to update
  - Pick one node at a time
  - Update out(u) from in(u)
  - If out(u) has changed, recompute in(v) for all successors v of u and add v to the work list

Chaotic Iterations: Example
Using Reaching-Definitions Information

- Remember: this is an over-approximation
  - A definition may be reaching use
- We may err, but always on the safe side
  - We may say that a definition may reach a program point when it doesn’t
  - We never miss a definition that may reach a point
- Usage examples
  - detecting possible use before any definition
  - useful for debugging
  - very simple constant folding

Available Expressions Analysis

```
x = a + b
y = a * b
while (y > a + b) {
  a = a + 1
  x = a + b
}
```

Some Required Notation

- Classes of expressions:
  - AExp – arithmetic expressions
  - BExp – boolean expressions
- FV: (BExp U AExp) → 𝒫(Var)
  - Variables used in an expression
- AExp(a) = all (non-atomic) arithmetic sub-expressions of an arithmetic expression a
  - AExp(b) for a boolean expression b

Available Expressions Analysis

- Property space
  - L = 𝒫(AExp) ; ⊑ = ⊇
  - in, out: Lab → L
    - Map a statement label to set of arithmetic expressions that are available at (before, after) that statement
- Dataflow equations
  - Flow equations – how to join incoming dataflow facts
  - Effect equations – given an input set of expressions in(i), what is the effect of the statement at i

Available Expressions Analysis

- in(ℓ) =
  - ∅ when ℓ is the initial label
  - ∩ (out(ℓ′) | ℓ′ ∈ pred(ℓ)) otherwise
- out(ℓ) =
  ```
  ∅(a) = nil | {x′ ∈ AExp | x = FV(a) U ∪ {x′ ∈ AExp(a) | x ∈ FV(x′)}
  skip = m(ℓ)
  cond = m(ℓ) U AExp(cond)
  ```
Transfer Functions

1: \( x := a + b \)
2: \( y := a \times b \)
3: \( y > a + b \)
4: \( a := a + 1 \)
5: \( x := a + b \)

\[ \text{out}(1) = \text{in}(1) \setminus \emptyset \cup \{ a + b \} \]
\[ \text{out}(2) = \text{in}(2) \setminus \emptyset \cup \{ a \times b \} \]
\[ \text{in}(1) = \emptyset \]
\[ \text{in}(2) = \text{out}(1) \]
\[ \text{in}(3) = \text{out}(2) \cap \text{out}(5) \]
\[ \text{in}(4) = \text{out}(3) \]
\[ \text{in}(5) = \text{out}(4) \]

Solution

\[ \text{out}(1) = \text{in}(1) \setminus \emptyset \cup \{ a + b \} \]
\[ \text{out}(2) = \text{in}(2) \setminus \emptyset \cup \{ a \times b \} \]
\[ \text{in}(1) = \emptyset \]
\[ \text{in}(2) = \text{out}(1) \]
\[ \text{in}(3) = \text{out}(2) \]
\[ \text{in}(4) = \text{out}(3) \]
\[ \text{in}(5) = \text{out}(4) \]

Kill/Gen

Statement \[ \text{out}(\ell) \]
\[ x := a \]
\[ \text{in}(\ell) \setminus \{ (x, i) \mid i \in \text{Lab} \} \cup \{ (x, \ell) \} \]
\[ \text{skip} \]
\[ \emptyset \]
\[ \text{cond} \]
\[ \text{in}(\ell) \]

Statement \[ \text{in}(\ell) \]
\[ x := a \]
\[ \{ (x, \ell) \} \]
\[ \emptyset \]
\[ \text{cond} \]
\[ \emptyset \]

\[ \text{out}(\ell) = \text{in}(\ell) \setminus \text{kill}(B) \cup \text{gen}(B') \]
\[ B' = \text{statement (or block) at label } \ell \]

Reaching Definitions Revisited

Statement \[ \text{out}(\ell) \]
\[ x := a \]
\[ \{ (x, \ell) \} \setminus \{ (x, i) \mid i \in \text{Lab} \} \cup \{ (x, \ell) \} \]
\[ \text{skip} \]
\[ \emptyset \]
\[ \text{cond} \]
\[ \emptyset \]

Statement \[ \text{in}(\ell) \]
\[ x := a \]
\[ \{ (x, \ell) \} \]
\[ \emptyset \]
\[ \text{cond} \]
\[ \emptyset \]

Live Variables

1. \( x := 2 \)
2. \( y := 4 \)
3. \( x := 1 \)
4. if \( y > x \)
5. then \( z := y \)
6. else \( z := y + y \)
7. \( x := z \)

For each program point, which assignments \textbf{may} have been made, and not overwritten, when program execution reaches that point along \textit{some path}.

For each program point, which variables \textbf{may} be live (i.e., has some future use before re-definition, along \textit{some path}) at the exit from that point.
Live Variables
1: x := 2
2: y := 4
3: x := 1
4: if y > x
5: then z := y
6: else z := y * y;
7: x := z

Block kill gen
x := a 
{ x } 
FV(a)

Block kill gen
x := a 
{ x } 
FV(a)

Live Variables — Solution
1: x := 2
2: y := 4
3: x := 1
4: if y > x
5: then z := y
6: else z := y * y;
7: x := z

out(1) = ∅
in(2) = in(1) = ∅
in(3) = { y }
in(4) = { y }
in(5) = { z }
in(6) = { y }
in(7) = ∅

Monotone Framework
\[
\text{in}(\ell) = \begin{cases} \text{initial} \quad \text{when } \ell \text{ is an initial state} \\
\bigcup \{ \text{out}(\ell') \mid (\ell,\ell') \in \text{CFG edges} \} \quad \text{otherwise} 
\end{cases}
\]
\[
\text{out}(\ell) = f_{\ell}(\text{in}(\ell))
\]

- CFG edges can be traversed either forward or backwards
- Entry labels are initial program labels or final program labels (when going backwards)
- Initial is an initial state (or final when going backwards)
- \( f \) is the transfer function associated with the block \( B \)

Forward vs. Backward Analyses
Example: Reaching Definition

- \( L = \mathcal{P}(\text{Var} \times \text{Lab}) \), partially ordered by \( \subseteq \)
- \( \sqcup \) is \( \cup \)
- \( L \) has no infinite chains because \( \text{Var} \times \text{Lab} \) is finite (for a given program)

Example: Available Expressions

- \( L = \mathcal{P}(\text{AExp}) \), partially ordered by \( \sqsubseteq = \supseteq \)
- \( \sqcup \) is \( \cap \)
- \( L \) has no infinite chains because \( \text{AExp} \) is finite (for a given program)

Analyses Summary

<table>
<thead>
<tr>
<th>Analyses</th>
<th>Reaching Definitions</th>
<th>Available Expressions</th>
<th>Live Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>( \mathcal{P}(\text{Var} \times \text{Lab}) )</td>
<td>( \mathcal{P}(\text{AExp}) )</td>
<td>( \mathcal{P}(\text{Var}) )</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>( \sqcup )</td>
<td>( \sqcup )</td>
<td>( \sqcup )</td>
<td>( \sqcup )</td>
</tr>
<tr>
<td>( \sqsubseteq )</td>
<td>( \sqsubseteq )</td>
<td>( \sqsubseteq )</td>
<td>( \sqsubseteq )</td>
</tr>
<tr>
<td>Initial</td>
<td>( {x, ? \mid x \in \text{Globals}} )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>Entry labels</td>
<td>{init}</td>
<td>{init}</td>
<td>final</td>
</tr>
<tr>
<td>Direction</td>
<td>forward</td>
<td>forward</td>
<td>backward</td>
</tr>
<tr>
<td>( f_r )</td>
<td>( {\text{val} } = (\text{val}, \text{kill}) )</td>
<td>( \cup )</td>
<td>( \cup )</td>
</tr>
</tbody>
</table>

Summary

- Static Analysis
  - Prove properties of a program at compile time
  - Over-approximate possible program behaviors
- Dataflow Analysis
  - Build control-flow graph
  - Assign transfer functions
  - Compute fixed point
- Monotone Framework
  - Can be used to express many useful analyses

Coming Up