You are here

Up Until Now

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”

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

Static Analysis

- Central idea: use approximation

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

- Conservative static analysis: assertion may be violated

Over-Approximation

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

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

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

Precision
Static Analysis

- Formalize software behavior in 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 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:
  \[ x = y + z \]
  \( x \) is a definition of \( x \)
  \( y \) is a definition of \( y \)
  \( z \) is a definition of \( z \)
- A definition reaches a use if
  \( y \) value written by definition...
  \( z \) may be read by use

Example: Reaching Definitions

```
1 y := x
2 z := 1
3 while (y > 0) {
4   y := y - 1
5 } 
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. \( z := z \times y \)
   5. \( y := y - 1 \)
4. \( y := 0 \)
5. Return \( y + z \)

(adapted from Nielson, Nielson & Hankin)
Time for Some Math

Partial Orders

- Set $P$
- Binary relation $\leq$ such that $\forall x, y, z \in P$:
  - $x \leq x$ (reflexive)
  - $x \leq y$ and $y \leq x$ implies $x = y$ (antisymmetric)
  - $x \leq y$ and $y \leq z$ implies $x \leq z$ (transitive)
- Can use partial order to define:
  - Upper and lower bounds
  - Least upper bound
  - Greatest lower bound

Upper Bounds

- For $S \subseteq P$:
  - $x \in P$ is an upper bound of $S$ if $\forall y \in S. y \leq x$
  - $x \in P$ is the least upper bound of $S$ if
    - $x$ is an upper bound of $S$, and
    - $x \leq y$ for all upper bounds $y$ of $S$
  - $\lor$ - join, least upper bound, lub, supremum, sup
    - $\lor$ is the least upper bound of $S$
    - $x \lor y$ is the least upper bound of $\{x, y\}$
    - Often written as $\sqcup$ as well

Lower Bounds

- For $S \subseteq P$:
  - $x \in P$ is a lower bound of $S$ if $\forall y \in S. x \leq y$
  - $x \in P$ is the greatest lower bound of $S$ if
    - $x$ is a greatest lower bound of $S$, and
    - $y \leq x$ for all greatest lower bounds $y$ of $S$
  - $\land$ - meet, greatest lower bound, glb, infimum, inf
    - $\land$ is the greatest lower bound of $S$
    - $x \land y$ is the greatest lower bound of $\{x, y\}$
    - Often written as $\sqcap$ as well

Covering

- $x < y$ if $x \leq y$ and $x \neq y$
- $x$ is covered by $y$ (y covers x) if
  - $x < y$, and
  - $x \leq z \implies x = z$
- Conceptually,
  - $y$ covers $x$ if there are no elements between $x$ and $y$
Lattices

- If \( x \lor y \) and \( x \land y \) exist for all \( x, y \in P \) then \( P \) is a lattice
- If \( \lor S \) and \( \land S \) 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 \( \mathbb{Z} \)
  - \( \lor \) = max, \( \land \) = min
  - But \( \lor \mathbb{Z} \) and \( \land \mathbb{Z} \) do not exist \( \Rightarrow \) not complete
  - Conversely, \( \mathbb{Z} \cup \{\infty, -\infty\} \) is a complete lattice

Example

- \( P = \{000, 001, 010, 011, 100, 101, 110, 111\} \) (standard boolean lattice, also called hypercube)
- \( x \leq y \) iff \( (x \& y) = x \) where \( \& \) is bitwise ‘and’

Top and Bottom

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

Product Lattices

- Given two lattices \( L \) and \( Q \), the product can easily be made a lattice:
  \[ (l_0, q_0) \subseteq (l_1, q_1) \Rightarrow l_1 \subseteq l_0 \text{ and } q_1 \subseteq q_0. \]
- For vectors of \( L \), defining a lattice is also easy
  \[ (l_0, l_1, \ldots, l_n) \subseteq (l_0, l_1, \ldots, l_1) \Rightarrow \forall i \in [0, L], l_i \subseteq l_i \]

Lattices of Program Properties

- Properties of interest can often be arranged into a lattice
- Example: Lattices of values
  - odd even
  - 1 2 3
  - When the value of each variable is a lattice, the state of the program is a product lattice of the states of all variables.
Example

\[ y := 0; \]
while (x < 10) {
    \[ x := x + 1; \]
    \[ y := y + 2; \]
}

Could be odd or even

\[ x = \begin{cases} 
\text{odd} & \text{if } x \\
\text{even} & \text{otherwise}
\end{cases} \]

Is \( y \) guaranteed to be even?

A lattice of predicates
- \( \langle x = (\bot, \text{even}, \text{odd}, \top), y = (\bot, \text{even}, \text{odd}, \top) \rangle \)
- \( x: x \in \text{even}, y: y \in \text{odd}, z: z \in \text{even} \)
- Product lattice from 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 monotonic (see later why)

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

```
return y
```

From CFG to Equations
- In the case of Reaching Definitions:
  - \( \text{out}_i = \text{in}_i \setminus (\{x^*, i\} \cup \{x, i\}) \)
  - where
    - \( x \) is the variable assigned to in \( i \)
    - \( \{x, i\} = \{x, i \mid i \in \text{Lab}\} \)

```
return y
```
Input/output Sets

1: \textit{y} := x
2: \textit{z} := 1
3: while (\textit{y} > 0) {
   4:   \textit{z} := \textit{z} \ast \textit{y}
   5:   \textit{y} := \textit{y} – 1
5:   return \textit{y} + \textit{z}

Transfer Functions

1: \textit{y} := x
2: \textit{z} := 1
3: while (\textit{y} > 0) {
   4:   \textit{z} := \textit{z} \ast \textit{y}
   5:   \textit{y} := \textit{y} – 1
5:   return \textit{y} + \textit{z}

System of Equations

1: \textit{y} := x
2: \textit{z} := 1
3: while (\textit{y} > 0) {
   4:   \textit{z} := \textit{z} \ast \textit{y}
   5:   \textit{y} := \textit{y} – 1
5:   return \textit{y} + \textit{z}

Solving the Equations

\textbullet\ Fixed Point Problem
  \textbullet\ Given a function \( F : L \rightarrow L \), find \( x \in L \)
  such that \( F(x) = x \)

\textbullet\ With transfer functions, you will commonly find that \( \top \) is one such solution...
  \textbullet\ We would like the most precise solution

Knaster-Tarski Theorem

\textbullet\ Definition. the least fixed point \( x_L \) is a fixed point \( F(x_L) = x_L \)
  such that for any \( x \), if \( F(x) = x \), then \( x_L \leq x \)
Kleene Fixed-point Theorem

Order Preserving (Monotonic) Function:
\[ x \subseteq y = f(x) \subseteq f(y) \]

Now, let \( s_0 \) be the least fixed point of \( f: L \to L \)
- \( \mathbb{1} \cup s_0 \)

Now, let \( s_0 \) be a and \( s_1 = f(s_0) \)
- By induction, \( x, L, S \)
- Also, the chain \( s_1 \) is a ascending chain
- If \( L \) has no infinite ascending chains, sooner or later \( s_1 = s_2 = s_3 \)

Same trick works for greatest fixed point!
- But then you have to start with \( s_0 = 1 \)

Chains

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

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

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 \( \text{out}(u) \) from \( \text{in}(u) \)
  - If \( \text{out}(u) \) has changed, recompute \( \text{in}(v) \) for all successors \( v \) of \( u \) and add \( v \) to the work list

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 definition
    - useful for debugging
    - very simple constant folding
For each program point, find which expressions must have already been computed, and not later modified, on all paths leading to that program point.

```plaintext
1: x := a + b
2: y := a * b
3: while (y > a + b) {
   4: a := a + 1
   5: x := a + b
}
```

Some Required Notation

- Classes of expressions:
  - \( \text{AExp} \) – arithmetic expressions
  - \( \text{BExp} \) – boolean expressions
- \( \text{FV} \): \((\text{BExp} \cup \text{AExp}) \rightarrow \text{Var}\)
  - Variables used in an expression
- \( \text{AExp}(a) = \) all (non-atomic) arithmetic sub-expressions of an arithmetic expression \( a \)
- \( \text{AExp}(b) \) for a boolean expression \( b \)

Available Expressions Analysis

- Property space
  - \( \text{in}, \text{out}: \text{Lab} \rightarrow (\exists \text{AExp}) \)
    Map a label to set of arithmetic expressions available at (before, after) that label
- Dataflow equations
  - Flow equations – how to join incoming dataflow facts
  - Effect equations – given an input set of expressions \( S \), what is the effect of a statement

- \( \text{in}(\ell) = \emptyset \) when \( \ell \) is the initial label
- \( \cap \{ \text{out}('\ell') \mid '\ell' \in \text{pred}(\ell) \} \) otherwise

Available Expressions Analysis

- Statement
  - \( \text{state} = \{ \begin{array}{l}
  x := a \\
  \text{skip} \\
  \text{cond}
  \end{array} \}
  \text{out} = \{ \begin{array}{l}
  x \in \text{AExp} | x \in \text{FV}(a') | x \in \text{AExp}(a') | x \in \text{FV}(a') \\
  x \\
  x \neq \text{FV}(a')
  \end{array} \}

Transfer Functions

- \( \text{in}[1] = \emptyset \)
- \( \text{in}[2] = \text{out}[2] \cup \text{out}[5] \)
- \( \text{in}[3] = \text{out}[3] \)
- \( \text{in}[4] = \text{out}[4] \)
- \( \text{in}[5] = \text{out}[5] \)

- \( \text{in}[1] = \emptyset \)
- \( \text{in}[2] = \text{out}[2] \cup \text{out}[5] \)
- \( \text{in}[3] = \text{out}[3] \)
- \( \text{in}[4] = \text{out}[4] \)
- \( \text{in}[5] = \text{out}[5] \)

- \( \text{out}[1] = \text{in}[1] \cup \{ a + b \} \)
- \( \text{out}[2] = \text{in}[2] \cup \{ a * b \} \)
- \( \text{out}[3] = \text{in}[3] \cup \{ a + 1 \} \)
- \( \text{out}[4] = \text{in}[4] \cup \{ a + b \} \)
- \( \text{out}[5] = \text{in}[5] \cup \{ a + b \} \)
Solution

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

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

Kill/Gen

\( \text{Statement} \quad \text{in}(\ell) \quad \text{out}(\ell) \quad \text{cond} \)

\( \text{Statement} \quad \text{in}(\ell) \quad \text{out}(\ell) \quad \text{cond} \)

Kill/Gen

\( \text{Statement} \quad \text{in}(\ell) \quad \text{out}(\ell) \quad \text{cond} \)

\( \text{Statement} \quad \text{in}(\ell) \quad \text{out}(\ell) \quad \text{cond} \)

Reaching Definitions Revisited

\( \text{Statement} \quad \text{in}(\ell) \quad \text{out}(\ell) \quad \text{cond} \)

\( \text{Statement} \quad \text{in}(\ell) \quad \text{out}(\ell) \quad \text{cond} \)

Live Variables

\( [x := 2]; \)
\( [y := 4]; \)
\( [x := 1]; \)
\( \text{if } [y > x] \text{ then } [z := y]; \)
\( \text{else } [z := y \cdot y]; \)
\( [x := z]; \)

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

Live Variables

\( [x := 2]; \)
\( [y := 4]; \)
\( [x := 1]; \)
\( \text{if } [y > x] \text{ then } [z := y]; \)
\( \text{else } [z := y \cdot y]; \)
\( [x := z]; \)

For each program point, which variables \textbf{may} be live at the exit from the point.
### Live Variables

\[ \begin{align*}
1: & \quad x := 2 \\
2: & \quad y := 4 \\
4: & \quad \text{if } y > x \text{ then } z := y \text{ else } z := y^2 \\
5: & \quad x := z
\end{align*} \]

### Live Variables: solution

\[ \begin{align*}
1: & \quad x := 2 \\
2: & \quad y := 4 \\
4: & \quad \text{if } y > x \text{ then } z := y \text{ else } z := y^2 \\
5: & \quad x := z
\end{align*} \]
Example: Available Expressions

- $L = \mathcal{P}(A\text{Exp})$ is partially ordered by $\sqsubseteq$
- $\sqcup$ is $\cap$
- $L$ satisfies the Ascending Chain Condition because $A\text{Exp}$ is finite (for a given program)

Analyses Summary

<table>
<thead>
<tr>
<th>Refining Definitions</th>
<th>Available Expressions</th>
<th>Live Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sqsubseteq$</td>
<td>$\mathcal{P}(\text{Var} \times \text{Lab})$</td>
<td>$\mathcal{P}(\text{Var})$</td>
</tr>
<tr>
<td>$\sqcup$</td>
<td>$\sqsubseteq$</td>
<td>$\sqsubseteq$</td>
</tr>
<tr>
<td>Initial</td>
<td>${ (x,?)</td>
<td>x \in \text{Var} }$</td>
</tr>
<tr>
<td>Entry labels</td>
<td>(init)</td>
<td>(init) Final</td>
</tr>
<tr>
<td>Direction</td>
<td>Forward</td>
<td>Forward Backward</td>
</tr>
<tr>
<td>$f_{\text{lab}}$</td>
<td>$f_{\text{lab}}(\text{val}) = (\text{val} \cup \text{kill}) \cup \text{gen}$</td>
<td></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...