CS510 Software Engineering

Static Program Analysis

Asst. Prof. Mathias Payer

Department of Computer Science
Purdue University

TA: Scott A. Carr
Slides inspired by Xiangyu Zhang

http://nebelwelt.net/teaching/15-CS510-SE

Spring 2015

Table of Contents

1 Static Analysis

2 Data-Flow Analysis
- Motivating Example: Reaching Definitions
- Common Analysis Framework

Static Analysis

- Static analysis analyzes a program without executing it.
- Static analysis is widely used in bug finding, vulnerability detection, or property checking.
- "*Easier*" to apply compared to dynamic analysis (as long as you have code): analysis can be transparent to the user.
- Better scalability compared to some dynamic analysis (e.g., tracing).
- Large success in recent years: findbug, coverity[1], codesurfer.

[1] Reading material: Al Bessey et al., A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World, CACM'10.

Static Analaysis: Syntax/Structure

- Focus on syntax and structure, not semantics.
- Look at CFG, dominator, post-dominator, loop detection
- Application: detect code copies (comparison based on text, AST, CFG)
- Application: Malware analysis
- Recover information about the program, serve as basis for further advanced static/dynamic analysis.
- Limitation: cannot reason about program semantics or state.

Static Analysis: Semantics

- Focus on program semantics.
- Reason about program meaning/logic.
- Evaluate the meaning of syntactically legal strings defined by a specific programming language, reason about involved computation.
- (Illegal strings – according to the language definition – result in non-computation).
- We'll focus on semantic-based static analysis.

Simple Static Analysis (1)

What are possible definitions of each use?

```
z = val1;
x = val2;
if (p1)
  x = val3;
else
  s1;
z = val4;
if (p2)
  y = x;
else
  y = z;
```

$y = \{?\}$
$z = \{val4\}$ $x = \{val2, val2\}$
$y = \{val2, val3, val4\}$

Simple Static Analysis (2)

What are possible call targets?

```
p = F1;
q = F2;
if (p1)
  q = F3;
else
  p = F4;
if (p2)
  p = F5;
else
  p = q;
(*p)()
```

$q = \{F2, F3\}$
$p = \{F5, F2, F3\}$

Simple Static Analysis (3)

What are possible ranges of a variable?

```
1  x = 10;
2  y = input();
3  i = x+y;
4  if (i>20)
5     i = 20;
6  else
7     z = input();
8     if (3<z<5)
9        i=i-z;
10 print i;
```

$val1 <= i <= val2?$

$i_5 = 10..20?$ $i_5 = 10 or 20!$

$i_7 = 10..20$

$i_9 = 6..20$

Static Analysis: Requirements

- Abstract domain: the results we want to compute by static analysis.
- Transfer function: how the abstract values are computed/updated at each relevant instruction.
- (We must consider the instruction semantics for the transfer function!)

Simple Static Analysis (4)

What are possible call targets?

```
x = F1; y = F2; q = &x;
if (p1)
  x = F3;
else
  p = &x;
if (p2)
  p = q;
else
  p = &y;
*(*p)();
```

$p = \{\&y \text{ OR } q\}$ $q = \{\&x\}$
$x = \{F1, F3\}$, $y = \{F2\}$
Possible call targets: $\{F1, F2, F3\}$(Note double indirection!)

Static Analysis: Loops

- When shall we terminate a loop path?
- How many iterations should we consider?
- Is the loop bound? How to infer possible values?
- Observation: we are interested in the aggregation of abstract values along paths.
- If the aggregation stabilizes, we can terminate.
- Assumption: monotonic growth.
- Assumption: abstract domain is finite.

Static Analysis: Use-cases

- Optimization: Global Common Subexpression
- Optimization: Copy Propagation
- Optimization: Dead-Code Elimination
- Optimization: Code Motion
- Optimization: Strength Reduction
- All these optimizations depend on data-flow analysis!

Table of Contents

1 Static Analysis

2 Data-Flow Analysis
- Motivating Example: Reaching Definitions
- Common Analysis Framework

Data-Flow Analysis

Data-Flow Analysis

Data-Flow Analysis refers to a body of techniques that derive information about the flow of data along program execution paths.

For example, to implement *global subexpression elimination* the compiler uses data-flow analysis to prove that along any execution path two textually identical expressions evaluate to the same value. Another example is *dead store elimination* where the compiler proves that a value will not be read along any path after the assignment, allowing the removal of the assignment.

Reaching Definitions

Reaching Definitions

The definitions d that *may* reach a program point p along *some* path are known as *reaching definitions*.
A definition d of a variable x reaches a point p if there is a path from d to p along which x is not redefined.

Aliasing makes it hard to determine if an assignment redefines (kills) a particular variable. Program analysis is conservative: if we do not know that an assignment does not define a variable we assume it *may*.

Reaching definitions are, e.g., used to find possible uses of uninitialized variables. At variable declaration, add a dummy definition to the data-flow graph. If the dummy definition reaches any statement that uses the variable then we flag a use-before-def.

Iterative Algorithm

$$OUT[ENTRY] = \emptyset$$

$$\forall_{B \neq ENTRY} OUT[B] = \emptyset$$

while (changes):

$$OUT[B] = gen_B \cup (IN[B] - kill_B)$$

$$IN[B] = \cup_{\text{P a predecessor of B}} OUT[P]$$

Example

```
1  i = m-1
2  j = n
3  a = u1
4  do {
5     i = i + 1
6     j = j - 1
7     if (p2)
8        a = u2
9     i = u3
10 } while (p1)
```

$$gen_{B_1} = \{d_1, d_2, d_3\}$$
$$kill_{B_1} = \{d_5, d_6, d_8, d_9\}$$
$$gen_{B_2} = \{d_5, d_6\}$$
$$kill_{B_2} = \{d_1, d_2, d_9\}$$
$$gen_{B_3} = \{d_8\}$$
$$kill_{B_3} = \{d_3\}$$
$$gen_{B_4} = \{d_9\}$$
$$kill_{B_4} = \{d_1, d_5\}$$

Iterative Algorithm: Computation

Block B	$OUT[B]^0$	$IN[B]^1$	$OUT[B]^1$	$IN[B]^2$	$OUT[B]^2$
B_1	000 0000	000 0000	111 0000	000 0000	111 0000
B_2	000 0000	111 0000	001 1100	111 0111	001 1110
B_3	000 0000	001 1100	000 1110	001 1110	001 0111
B_4	000 0000	001 1110	001 0111	001 1110	001 0111
EXIT	000 0000	001 0111	001 0111	001 0111	001 0111

Data-Flow Analysis Framework

Data-Flow Analysis Framework

A *data-flow analysis framework* $(D, V, \wedge, F)$ consists of:

1. A direction of the data flow D, which is either Forwards or Backwards.
2. A semilattice, which includes a *domain* of values V and a *meet operator* $\wedge$.
3. A family F of transfer functions from V to V. Note that F must include constant transfer functions for the special nodes ENTRY and EXIT in the flow graph.

Semilattice

Semilattice

A meet semilattice is an algebraic structure $\langle S, \wedge \rangle$ consisting of a set S of values ("a domain of values") and a meet operator $\wedge$ such that:

$$\forall a, b, c \in S : a \wedge a = a; a \wedge b = b \wedge a; a \wedge (a \wedge c) = (a \wedge b) \wedge c$$

(idempotent, commutative, and associative)

$$\forall a, b, c \in S : a \geq b \iff a \wedge b = b;$$
$$a > b \iff a \geq b \text{ and } a \neq b; a \geq b \text{ and } b \geq c \iff a \geq c$$

($\wedge$ imposes partial ordering on S)

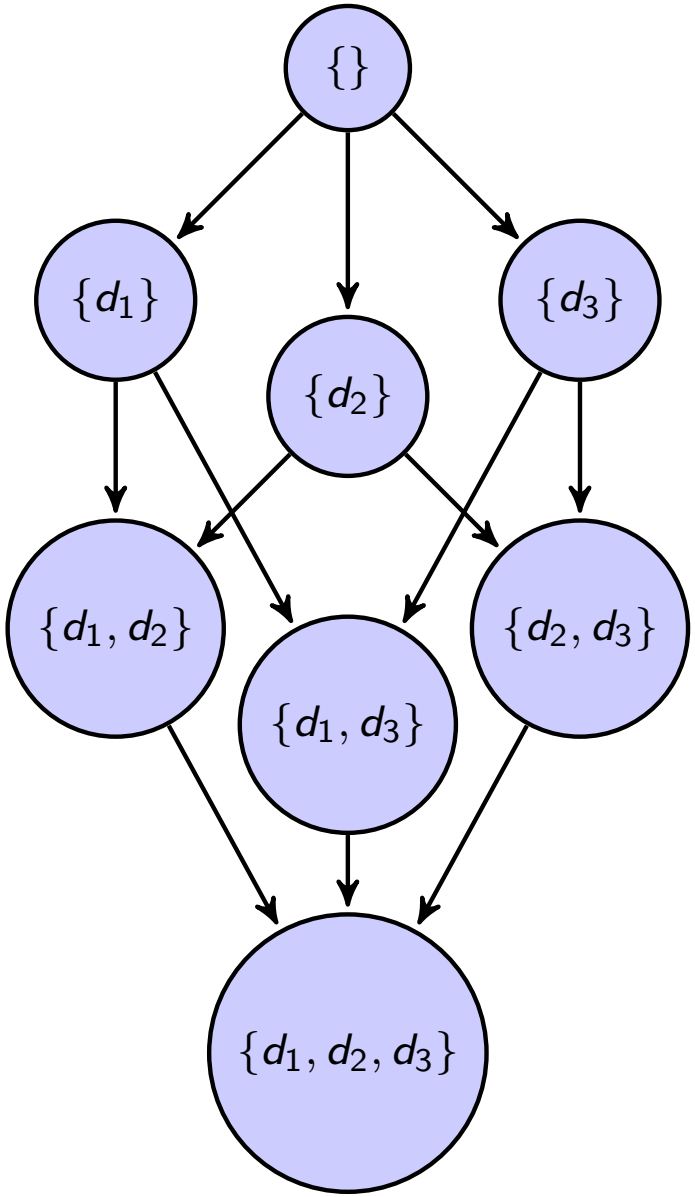
$$\exists T : \forall a \in S : a \leq T; T \wedge a = a$$

(there exists a top element T)

Semilattice Diagrams

Drawing the domain V helps understanding semilattice data-flow analyses.

The analysis starts at the top (knowing nothing) and tries to push down towards bottom (e.g., determining the reaching definitions).



Scaling Data-Flow

- Lattice, monotonicity, finite height leads to termination.
- Are we there yet?
- Unfortunately not due to path explosion (2^N transitions for N definitions).
- Analyze multiple paths at a time and compute aggregate information directly.

Summary

- Static analysis is over an *abstraction domain*.
- Static analysis uses *transfer functions* to transition between states.
- Static analysis must terminate.

Questions?

?