# Static Analysis
# Dataflow Analysis

25 October 2017

# Static analysis: definition

Analysis of code (usually source) without executing the program, in order to determine some program *properties*

mainly correctness, but also performance, etc.

Complementary to *dynamic* analyses (that run the code)

Sample properties

- uninitialized variables
- null pointers
- unused assignments
- code vulnerabilities (overflows, index out of range, etc.)

Usually, static analyses are linked to program *semantics*
sometimes, limited to (syntactic) *structure* of program

History:

strongly linked to compilers (mainly optimization)

more recently: in language design; for error detection

# Dataflow analysis

Techniques originating in the compiler domain
used for *code generation* (e.g., register allocation)
and code *optimization* (constant propagation/folding, common subexpression elimination, detecting uninitialized variables, etc.)

The same techniques can be applied to *code analysis* – very general

Basic ideas
construct program *control flow graph*
analyze how *properties* of interest change throughout the program (while traversing CFG nodes / edges)

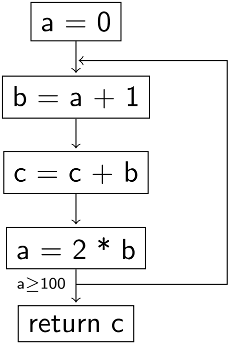
# Program control flow graph (CFG)

A program representation in which
  nodes are statements
  edges indicate sequencing/control flow (including jumps)
$\Rightarrow$ nodes may have:
    one successor (e.g., assignments)
    several successors (branch statements)
    several predecessors (e.g., join after an if)

Sometimes we also use the dual representation:
  nodes are program control points (program counter values)
  edges are statements with their effects

# Sample program and CFG

```
int a = 0, b, c = 0;
do {
  b = a + 1;
  c = c + b;
  a = 2 * b;
} while (a < 100);
return c;
```



# Notation

$G = (N, E)$ : control flow graph ($N$ : nodes; $E$ : edges)

$s$ : program statement (node in CFG)
$entry$, $exit$ : program entry and exit points
$in(s)$ : set of edges leading to $s$ (having $s$ as destination)
$out(s)$ : set of edges outgoing from $s$ (having $s$ as source)
$src(e)$, $dest(e)$ : source and destination statement of edge $e$
$pred(s)$ : set of predecessors of statement $s$
$succ(s)$ : set of successors of statement $s$

We will write *dataflow equations*:
describe how analyzed values (*dataflow facts*) change from one statement to another

We need the value (property) of interest:
at the entrypoint of $s$ (denote: $V_{in}$)
and the exit point (denote: $V_{out}$)

# Example: Reaching definitions

What are all *assignments* (definitions) *that may reach the current point* (without being overwritten by other assignments on the path)

Elements of interest: pairs (variable, source line for def).
For every statement (identified by its label $l$) we want the value before $RD_{in}(s)$ and after $RD_{out}(s)$

The entry point is not reached by any definition

$$RD_{out}(entry) = \{(v, ?) \mid v \in V\}$$

An *assignment* $l : v \leftarrow e$

removes all previous definitions for $v$ (unchanged for other vars)
and records current statement as definition

$$RD_{out}(l : v \leftarrow e) = (RD_{in}(s) \setminus \{(v, s')\}) \cup \{(v, l)\}$$

Def-values at *entry* of a statement are *union* of def-values at *exit* of predecessor statements:

$$RD_{in}(s) = \bigcup_{s' \in pred(s)} RD_{out}(s')$$

# Example: Live variables analysis

At every program point, which variables will have their values used on at least one path from that point?

useful in compilers for register allocation

Transfer function: $LV_{in}(s) = (LV_{out}(s) \setminus write(s)) \cup read(s)$

A variable is *live* before $s$

if it is read by $s$

or it is *live* after $s$ and not written by $s$

$\Rightarrow$ direction of analysis is *backwards*

Meet (combine) operation:

$$LV_{out}(s) = \begin{cases} \emptyset & \text{if } succ(s) = \emptyset \\ \bigcup_{s' \in succ(s)} LV_{in}(s') & \text{otherwise} \end{cases}$$

$\Rightarrow$ combination is union (*may*, at least one path)

Computation: *worklist* algorithm that makes changes from initial values until there are no more changes $\Rightarrow$ *fixpoint* is reached

# Example: Available expressions

At every program point, what are the expressions whose value is available (previously computed) without having changed on any path to that point?

if value is stored in a temp / register, need not recompute

Transfer function: $AE_{out}(s) = (AE_{in}(s) \setminus \{e \mid V(e) \cap write(s) \neq \emptyset\})$
$\cup \{e \in Subexp(s) \mid V(e) \cap write(s) = \emptyset\}$
(expressions at entry of $s$ that have not been changed by $s$,
and any expressions computed in $s$ without change to their variables)

Meet (combine) operation:

$$AE_{in}(s) = \begin{cases} \emptyset & \text{if } pred(s) = \emptyset \\ \bigcap_{s' \in pred(s)} AE_{out}(s') & \text{otherwise} \end{cases}$$

$\Rightarrow$ combination done by intersection (*must*, on all paths);
analysis direction is *forward*

# Example: Very busy expressions

What expressions *must* be evaluated on any path from the current point before any of their variables is modified ?
$\Rightarrow$ evaluation can be hoisted up to the current point, before any branches
– a backwards and *must* (universal) analysis

$$VBE_{in}(s) = (VBE_{out}(s) \setminus \{e \mid V(e) \cap write(s) \neq \emptyset\}) \cup Subexp(s)$$

$$VBE_{out}(s) = \begin{cases} \emptyset & \text{if } succ(s) = \emptyset \\ \bigcap_{s' \in succ(s)} VBE_{in}(s') & \text{otherwise} \end{cases}$$

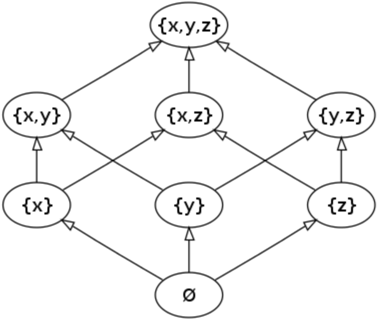
# Analyzed properties (dataflow facts)

*Concretely*, for each problem: we analyze some property, e.g.
– value of a variable at a program point
– or *interval* of values for a variable
– or sets of variables (live), expressions (available, very busy),
– possible definitions for a value (reaching definitions), etc.

*Abstract* view: a set $D$ of values for a property (*dataflow facts*)
Restriction: $D$ is a *finite* set

# Lattices

A *lattice* is a *partially ordered* set, in which every pair of elements has a least upper bound and a greatest lower bound.
(an element "larger", resp. "smaller" than either of them)
Ex: powerset of a set (intersection, union)
Ex: set of divisors of a number (gcd, least common multiple)



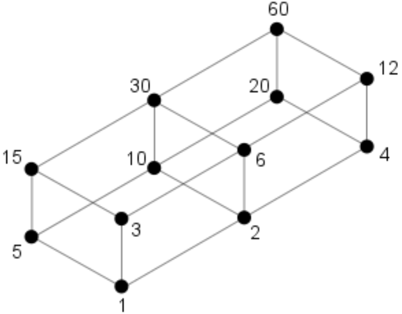




Image: `http://en.wikipedia.org/wiki/File:Hasse_diagram_of_powerset_of_3.svg`

`http://en.wikipedia.org/wiki/File:Lattice_of_the_divisibility_of_60.svg`

# Transfer functions

*Concrete* domain: program statements change program state.
e.g., value of variable after a statement is a function of its value before the statement.

*Abstract* domain: Each statement $s$ has an associated *transfer function* $F(s) : D \to D$ that determines how the value of a property at the start of a statement is changed by that statement:
$Val_{out}(s) = F(s)(Val_{in}(s))$ (for analysis going forward),
or conversely (for backwards analyses)

Restriction: analysis is easier for *monotone* transfer functions:

$$x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$$

(intuition: if the argument is more precise, so is the result)

Special case: *bitvector frameworks*: the lattice is a powerset, $\mathcal{P}(D)$, transfer functions are monotone, of the form:

$$F(s)(v) = (v \setminus kill(s)) \sqcup gen(s)$$

($v$ = dataflow fact, $gen/kill(s)$ = information generated/deleted by $s$)

# Dataflow equations

Example for forward analyses:

$$Val_{out}(s) = F(s)(Val_{in}(s))$$
$$Val_{in}(s) = \prod_{s' \in pred(s)} Val_{out}(s')$$

where $\prod$ is *meet* (combining effects) over several paths (could be $\cap$ or $\cup$)

Intially, we know value of $Val_{out}(entry)$.

For backwards analyses, we initially know $Val_{in}(exit)$ and the roles of *in* and *out* are switched.

# Solution: *worklist* algorithm

To compute a solution to this equation system: an iterative algorithm that propagates changes in the direction of the analysis.

**foreach** $s \in N$ **do** $Val_{in}(s) = \top$ // no info
$Val_{in}(entry) = init$ // depending on analysis
$W = \{entry\}$
**while** $W \neq \emptyset$
    **choose** $s \in W$
    old_out $= Val_{out}(s)$
    $W = W \setminus \{s\}$
    $Val_{in}(s) = \prod_{s' \in pred(s)} Val_{out}(s')$
    $Val_{out}(s) = F(s)(Val_{in}(s))$
    **if** $Val_{out}(s) \neq$ old_out **then**
        **forall** $s' \in succ(s)$ **do** $W = W \cup \{s'\}$

# Termination: fixpoint condition

Termination of analysis is guaranteed if the transfer function is monotone: $x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$, which implies that the computed values

Def: A *fixpoint* of a function $f$ is a value $x$ so that $f(x) = x$
*Kanster-Tarski* theorem guarantees that a monotone function over a complete lattice has a least and a greatest fixpoint.

The worklist algorithm computes the least fixpoint solution for the equation system given by the transfer functions.

# Meet over all paths

We wish to compute the combined effect of the program statements:
For a path (statement sequence) $p = s_1 s_2 \ldots s_n$ we define

$$F(p) = F(s_n) \circ \ldots \circ F(s_2) \circ F(s_1)$$

and we wish to compute:

$$\prod_{p \in Path(Prog)} F_p(entry)$$

The iterative algorithm *combines* effects at each join point before continuing computation. Since functions are monotone, we have:

$$f(x \sqcup y) \sqsupseteq f(x) \sqcup f(y)$$

so analysis *loses precision*

*Distributive* transfer functions satisfy: $f(x) \cup f(y) = f(x \cup y)$

In this case, the iterative fixpoint algorithm is equivalent with *meet over all paths*.

$\Rightarrow$ combining info on execution paths does not lose precision

All 4 classical examples (live variables, etc.) are distributive.

# Classification of analyses

– forward or backwards
– must or may
– flow-sensitive or insensitive (flow = control flow)
  e.g., does the statement order in the program matter ?
    – no: for variable used/changed, called functions, etc.
    – yes: for properties linked to actual values computed by program
– context-sensitive or context-insensitive ?
  is the analysis of a function/procedure specialized depending on the call site or not ? (generic function summary)
– *path*-sensitive or path-insensitive
  does it account for correlation between execution paths ?