Research Topics in Software Quality

Detecting Mechanical Bugs using Static Analysis
Acknowledgements

These notes combine slides from:

- Jonathan Aldrich
  “Analysis of Software Artefacts - Introduction to Static Analysis”

- Galeotii, Gorla, Rau
  “Automated Testing and Verification - Intraprocedural Dataflow Analysis”

- Stephen Chong
  “Advanced Topics in Programming Languages - Dataflow Analysis”

- Jeff Foster
  “Program Analysis and Understanding - Data Flow Analysis”

- Nielson, Nielson, Hankin
  “Principles of Program Analysis, Section 2.1”
Find the Bug!

Source: Engler et al., *Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI ’00.

```c
/* From Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct stripe_head *sh,
    int b_size) {
    struct buffer_head *bh;
    unsigned long flags;

    save_flags(flags);
    cli();
    if ((bh = sh->buffer_pool) == NULL)
        return NULL;
    sh->buffer_pool = bh->b_next;
    bh->b_size = b_size;
    restore_flags(flags);
    return bh;
}
```
/* From Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct stripe_head *sh,
    int b_size) {
    struct buffer_head *bh;
    unsigned long flags;

    save_flags(flags);
    cli();
    if ((bh = sh->buffer_pool) == NULL)
        return NULL;
    sh->buffer_pool = bh->b_next;
    bh->b_size = b_size;
    restore_flags(flags);
    return bh;
}
Metal Interrupt Analysis

```c
#include "linux-includes.h"

sm check_interrupts {
    // Variables
    // used in patterns
dcl { unsigned } flags;

    // Patterns
    // to specify enable/disable functions.
pat enable = { sti(); }
        | { restore_flags(flags); };
pat disable = { cli(); };

    // States
    // The first state is the initial state.
is_enabled: disable ==> is_disabled
        | enable ==> { err("double enable"); };
is_disabled: enable ==> is_enabled
        | disable ==> { err("double disable"); }
        | $end_of_path$ ==> 
        | { err("exiting w/intr disabled!"); };
    end path ==> err(end path with/intr disabled)
}
```

Source: Engler et al., *Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI ’00.
Static Analysis Finds “Mechanical” Errors

- Defects that result from inconsistently following simple, mechanical design rules

- Security vulnerabilities
  - Buffer overruns, unvalidated input…

- Memory errors
  - Null dereference, uninitialized data…

- Resource leaks
  - Memory, OS resources…

- Violations of API or framework rules
  - e.g. Windows device drivers; real time libraries; GUI frameworks

- Exceptions
  - Arithmetic/library/user-defined

- Encapsulation violations
  - Accessing internal data, calling private functions…

- Race conditions
  - Two threads access the same data without synchronization
Data Flow Analysis

- One of the most popular static analysis techniques.

**Purpose:** Infer automatically interesting properties of a program
  - Specifically, to a given program point

**Principle:** Model execution of a program as the solution of a set of equations describing the flow of values through the program instructions.
Data Flow Analysis: Intuition

- "execute" the program using **abstract values**.

- **Collect in each program point** all the information flowing to that point
  - It can give us information for each program point.
    - Which are the possible values of variable Y after executing instruction #5?
    - Can the "null" value flow towards x in any instruction?
  - It can distinguish instruction order
    - Was a file read **after** it was closed?

- **Flow sensitive**
  - Needs a control-flow graph
Intermezzo: Control Flow Graphs

Control Flow Graph (CFG)

- Shows order of statement execution
  - Determines where data flows
- Decomposes expressions into primitive operations
  - Typically one CFG node per “useful” AST node
    - constants, variables, binary operations, assignments, if, while…
  - Loops are written out
    - Form a loop in the CFG
- Benefit: analysis is defined one operation at a time
Intuition for Building a CFG

- Connect nodes in order of operation
  - Defined by language
- Java order of operation
  - Expressions, assignment, sequence
    - Evaluate subexpressions left to right
    - Evaluate node after children (postfix)
  - While, If
    - Evaluate condition first, then if/while
    - if branches to else and then
    - while branches to loop body and exit
Intermezzo: Control Flow Graphs

Control Flow Graph Example

while i*2 < 10 do
  if x < i+2
    then x := x + 5
  else i := i + 1
END

BEGIN

note: simpler on intermediate code (e.g., a graph of which the nodes are Jimple instructions), but then have to map intermediate code to original AST in error reporting
Control Flow Graph Example

while \( i \times 2 < 10 \) do
  if \( x < i + 2 \)
    then \( x := x + 5 \)
  else \( i := i + 1 \)

BEGIN

\( \times \) \( 10 \)

\( < \)

\( := \) \( := \)

\( + \) \( + \) \( + \)

\( i \) \( 2 \) \( x \) \( x \) \( i \) \( i \)

\( < \) \( + \) \( + \) \( 5 \) \( 1 \)

END
Application: Available Expressions

- An expression e is **available** at program point p if
  - e is computed on every path to p, and
  - the value of e has not changed since the last time e was computed on the paths to p
- Available expressions can be used to optimize code
  - If an expression is available, don’t need to recompute it (provided it is stored in a register somewhere)

```c
...  
{   }  
int b = a + 2;  
{ a + 2}  
int c = b*b;  
{ a + 2, b*b }  
int d = c + 1;  
{ a + 2, b*b, c+1}  
c = 4;  
{ a + 2, b*b }  
if(b < c) b = 2;  
else c = a+1;  
{ a + 2 }  
return d;
```
Application: Live Variables

• A variable $v$ is **live** at program point $p$ if
  • $v$ will be used on some execution path originating from $p$ before $v$ is overwritten

• **Optimization**
  • If a variable is not live, no need to keep it in a register
  • If variable is dead at assignment, can eliminate assignment

```c
... 
{ a } 
int b = a + 2; 
{ b } 
int c = b*b; 
{ c } 
int d = c + 1; 
{ d } 
c = 4; 
{ d, c } 
return d+c; 
{ } 
```
Application: Bug Detection

possible division by zero:

\[
\begin{align*}
x &:= 8; \\
y &:= x; \\
z &:= 0; \\
\text{while } y > -1 \text{ do} \\
\quad x &:= x / y; \\
\quad y &:= y - 2; \\
\quad z &:= 5;
\end{align*}
\]

possible null dereference:

\[
\begin{align*}
x &= \text{null}; \\
y &= x; \\
y.m();
\end{align*}
\]
Data Flow Analysis: Ingredients

- **Control-flow graph**: A representation of the flow of control in the program

- **Abstract values**: represent information flowing through the program

- **Transfer function**: what is the effect over the program state for every program instruction

- **Dataflow Equations**: how abstract values flow according to the control flow of the program
Ingredient: Abstract Values

- Choose an abstraction according to the interesting property
  - X can be equal to Zero?
  - Was expression a+b computed previously?
  - Do we need variable x at this point of the program?
  - Where does the value being assigned to x came from?
  - Is this file open?
  - Is variable x equal to null when it is de-referenced?
  - Do x and y represent the same object?

- **Key:** The abstract state must be **tractable**
  - Abstract values must belong to a **lattice**.
  - Typically **finite lattice** (at least lattice height)
Intermezzo: Lattices and Data Flow Facts

- Typically, data flow facts form lattices
- E.g., available expressions

\[
\begin{align*}
\top & \quad \emptyset \\
a+b, a*b, a+1 & \quad a+b, a*b \\
a*b, a+1 & \quad a+b, a+1 \\
a*b & \quad a+1 \\
a+1 & \quad a+b \\
\emptyset & \quad \bot
\end{align*}
\]
Intermezzo: Lattices and Data Flow Facts

• **A partial order** is a pair \((P, \leq)\) such that
  - \(\leq\) is a relation over \(P\) \((\leq \subseteq P \times P)\)
  - \(\leq\) is reflexive, anti-symmetric, and transitive

• **A partial order is a lattice** if every two elements of \(P\) have a unique least upper bound and greatest lower bound.
  - \(\sqcap\) is the meet operator: \(x \sqcap y\) is the greatest lower bound of \(x\) and \(y\)
    - \(x \sqcap y \leq x\) and \(x \sqcap y \leq y\)
    - if \(z \leq x\) and \(z \leq y\) then \(z \leq x \sqcap y\)
  - \(\sqcup\) is the join operator: \(x \sqcup y\) is the least upper bound of \(x\) and \(y\)
    - \(x \leq x \sqcup y\) and \(y \leq x \sqcup y\)
    - if \(x \leq z\) and \(y \leq z\) then \(x \sqcup y \leq z\)

• A join semi-lattice (meet semi-lattice) has only the join (meet) operator defined
Intermezzo: Lattices and Data Flow Facts

- A partially ordered set is a **complete lattice** if meet and join are defined for all subsets (i.e., not just for all pairs)
- A complete lattice always has a bottom element and a top element
- A finite lattice always has a bottom element and a top element
intermezzo: lattices and data flow facts

- $(2^S, \subseteq)$ forms a lattice for any set $S$
  - $2^S$ is powerset of $S$, the set of all subsets of $S$
- If $(S, \leq)$ is a lattice, so is $(S, \geq)$
  - i.e., can “flip” the lattice

- Lattice for constant propagation
Ingredient: Transfer Function

- Indicates the effect of each instruction on the abstract state
- Given a node (instruction) and a abstract state it creates a new abstract state
  - \( F_{\text{stmt}}(\sigma) = \sigma' \)
- Example:
  - \( F_{x:=x+y}([y \rightarrow \text{NZ}, x \rightarrow \text{Z}]) = [y \rightarrow \text{NZ}, x \rightarrow \text{NZ}] \)
- Some properties:
  - It has to be monotonous: \( x \leq y \rightarrow f(x) \leq f(y) \)
  - Closed under composition (\( f(f(x)) \) is always defined)
Ingredient: Data Flow Equations

- They provide how a node’s output is computed given its inputs
- In which order data flow and how it is combined
  - **Forward**: From the program entry towards its exit
    - Zero analysis, available expressions, etc
  - **Backward**: From the program exit to its entry
    - Live variables analysis
- How to interpret the collected data
  - What to do if there are data flowing from to different nodes:
    - Apply the “upper bound”/ **MAY Analysis**
    - Apply the “lower bound”/ **MUST Analysis**
*Ingredient: Data Flow Equations*

- **Live variables analysis**
  - \( \text{in}[n], \text{out}[n] = \text{set of variables} \)
  - \( \text{transfer}[n](X) = \text{gen}(n) \cup (X \setminus \text{kill}(n)) \)
    - \( \text{gen}(n) = \text{read accesses to variables in node } n \)
    - \( \text{kill}(n) = \text{write accesses to variables in } n \)
  - \( \oplus = \bigcup \) (of sets)
  - \( \text{out}[n] := \bigcup \{ \text{in}[m] \mid m \in \text{succ}(n) \} \)
  - \( \text{in}[n] := \text{transfer}[n](\text{out}[n]) \)

- **Available expressions**
  - \( \text{in}[n], \text{out}[n] = \text{set of expressions} \)
  - \( \text{transfer}[n](X) = \text{gen}(n) \cup (X \cap \text{kill}(n)) \)
    - \( \text{gen}(n) = \text{new expressions created} \)
    - \( \text{kill}(n) = \text{exprs containing variables written by } n \)
  - \( \oplus = \bigcap \) (of sets)
  - \( \text{in}[n] := \bigcap \{ \text{out}[m] \mid m \in \text{pred}(n) \} \)
  - \( \text{out}[n] := \text{transfer}[n](\text{in}[n]) \)
A Framework for Data Flow Analysis

For each node $n$:

- $\text{in}[n]$: *abstract values* before program point $n$
- $\text{out}[n]$: *abstract values* after program point $n$
- $\text{transfer}[n]$: *operation to apply on the values flowing through node $n$*

For each analysis:

- $\oplus$: join operator (for joining several input/output values)

<table>
<thead>
<tr>
<th>Direction</th>
<th>$\cup$ (MAY )</th>
<th>$\cap$ (MUST)</th>
</tr>
</thead>
</table>
| **Forward** | Given $\text{in}[n]$, compute $\text{out}[n]$  
Apply $\text{transfer}[n]$ to $\text{predecessors}[n]$  
Property holds in some path  
(reaching defs, zero analysis) | Given $\text{in}[n]$, compute $\text{out}[n]$  
Apply $\text{transfer}[n]$ to $\text{predecessors}[n]$  
Property holds in all paths  
(available expressions) |
| **Backward** | Given $\text{out}[n]$, compute $\text{in}[n]$  
Apply $\text{transfer}[n]$ to $\text{successors}[n]$  
Property holds in some path  
(live variable analysis) | Given $\text{out}[n]$, compute $\text{in}[n]$  
Apply $\text{transfer}[n]$ to $\text{successors}[n]$  
Property holds in all paths  
(very busy expressions) |
Solving Data Flow Equations

iterative algorithm

**Compute out[n] for each n ∈ N:**
out[n] := ⊥ (or TOP if MUST analysis)

Repeat
  For each n
    in[n] := ⊕ \{ out[m] | m ∈ pred(n) \}
    out[n] := transfer[n](in[n])

Until no further changes to in/out
Example: Zero Analysis

**lattice**: \((\text{var} \rightarrow \{\text{bot}, \text{Z}, \text{NZ}, \text{MZ}\})\)
- \(\text{bot} = []\), \(\text{top} = \{x \mapsto \text{MZ}, y \mapsto \text{MZ}, z \mapsto \text{MZ}\}\)
- \(\cup, \subseteq\)

**transfer**:
- \(F_{x:=e}(\sigma) = \sigma[x \leftarrow \sigma(e)]\)
- \(\sigma(\text{Z+NZ}) = \text{NZ}\)
- \(\sigma(\text{Z+Z}) = \text{Z}\)
- \(\sigma(\text{Z-Z}) = \text{Z}\)
- \(\sigma(\text{NZ-NZ}) = \text{MZ}\)
- ...

\(\text{in}[n] := \cup \{ \text{out}[m] \mid m \in \text{pred}(n) \}\)
\(\text{out}[n] := \text{transfer}[n](\text{in}[n])\)
Example: Zero Analysis

1: $x := 8$

2: $y := x$

3: $z := 0$

$y > -1$

4: $x := x/y$

5: $y := y - 2$

6: $z := 5$

$\sigma = []$

$[x \rightarrow \text{NZ}]$

$[x \rightarrow \text{NZ}, y \rightarrow \text{NZ}]$
Example: Zero Analysis

$\sigma = []$

$[x \rightarrow \text{NZ}]$

$[x \rightarrow \text{NZ, } y \rightarrow \text{NZ}]$

$[x \rightarrow \text{NZ, } y \rightarrow \text{NZ, } z \rightarrow \text{Z}]$

$\alpha(0) = \text{Z}$

1: $x := 8$

2: $y := x$

3: $z := 0$

4: $x := x/y$

5: $y := y - 2$

6: $z := 5$

in[y<-1] depends on out[3] and out[6].. latter has still its initial bottom value => join (=union) of out[3] and bottom = out[3], so can be ignored in first iteration
Example: Zero Analysis

\[ \sigma = [] \]

\[ [x \rightarrow NZ] \]

\[ [x \rightarrow NZ, y \rightarrow NZ] \]

\[ [x \rightarrow NZ, y \rightarrow NZ, z \rightarrow Z] \]

\[ [x \rightarrow NZ, y \rightarrow NZ, z \rightarrow Z] \]

\[ [x \rightarrow NZ, y \rightarrow NZ, z \rightarrow Z] \]

NZ \_a Y = NZ
Example: Zero Analysis

\[ \sigma = [] \]

1. \( x := 8 \)

2. \( y := x \)

3. \( z := 0 \)

4. \( x := x/y \)

5. \( y := y - 2 \)

6. \( z := 5 \)

\( y > -1 \)

\( x \rightarrow \text{NZ} \)

\( x \rightarrow \text{NZ, } y \rightarrow \text{NZ} \)

\( x \rightarrow \text{NZ, } y \rightarrow \text{NZ, } z \rightarrow Z \)

\( x \rightarrow \text{NZ, } y \rightarrow \text{MZ, } z \rightarrow Z \)

\( \text{NZ} \rightarrow \alpha(\alpha(2) = \text{NZ} \rightarrow \text{MZ} \)
Intermezzo: Imprecision in Static Analyses

- Abstraction => do not handle the concrete state
  - We do not handle actual information

- Example: Natural numbers
  - $3 - 3 = 0$
  - \text{Abs}(3) = \text{NZ}
  - How much is \text{NZ} – \text{NZ}?
    - \text{Abs}(3) = \text{NZ}
    - \text{Abs}(3-3)=\text{NZ}-\text{NZ} => \text{MZ}
Example: Zero Analysis

\[\sigma = []\]

\[[x \rightarrow \text{NZ}]\]

\[[x \rightarrow \text{NZ}, y \rightarrow \text{NZ}]\]

\[[x \rightarrow \text{NZ}, y \rightarrow \text{NZ}, z \rightarrow \text{Z}]\]

\[[x \rightarrow \text{NZ}, y \rightarrow \text{NZ}, z \rightarrow \text{Z}]\]

\[[x \rightarrow \text{NZ}, y \rightarrow \text{MZ}, z \rightarrow \text{Z}]\]

\[[x \rightarrow \text{NZ}, y \rightarrow \text{MZ}, z \rightarrow \text{NZ}]\]
Example: Zero Analysis

\[\sigma = []\]

[\[x \rightarrow \text{NZ}\]]

[\[x \rightarrow \text{NZ}, y \rightarrow \text{NZ}\]]

[\[x \rightarrow \text{NZ}, y \rightarrow \text{NZ}, z \rightarrow \text{Z}\]]

[\[x \rightarrow \text{NZ}, y \rightarrow \text{MZ}, z \rightarrow \text{Z}\]]

[\[x \rightarrow \text{NZ}, y \rightarrow \text{MZ}, z \rightarrow \text{NZ}\]]

\[y \rightarrow \text{NZ} \oplus \text{MZ}\]

\[z \rightarrow \text{Z} \oplus \text{NZ}\]
Example: Zero Analysis

\[ \sigma = [] \]

\[ [x \mapsto \text{NZ}] \]

\[ [x \mapsto \text{NZ}, y \mapsto \text{NZ}] \]

\[ [x \mapsto \text{NZ}, y \mapsto \text{MZ}, z \mapsto \text{MZ}] \]

1: \( x := 8 \)

2: \( y := x \)

3: \( z := 0 \)

4: \( x := x/y \)

5: \( y := y - 2 \)

6: \( z := 5 \)

\[ [x \mapsto \text{NZ}, y \mapsto \text{MZ}, z \mapsto \text{MZ}] \]

\[ y > -1 \]

\[ [x \mapsto \text{NZ}, y \mapsto \text{NZ}, z \mapsto \text{Z}] \]

\[ [x \mapsto \text{NZ}, y \mapsto \text{MZ}, z \mapsto \text{Z}] \]

\[ [x \mapsto \text{NZ}, y \mapsto \text{MZ}, z \mapsto \text{NZ}] \]
Example: Zero Analysis

\[
\sigma = [] \\
[x \mapsto \text{NZ}] \\
[x \mapsto \text{NZ}, y \mapsto \text{NZ}] \\
[x \mapsto \text{NZ}, y \mapsto \text{NZ}, z \mapsto \text{Z}] \\
[x \mapsto \text{NZ}, y \mapsto \text{MZ} \mapsto \text{MZ}] \\
[x \mapsto \text{NZ}, y \mapsto \text{MZ}, z \mapsto \text{Z}] \\
[x \mapsto \text{NZ}, y \mapsto \text{MZ}, z \mapsto \text{Z}] \\
[x \mapsto \text{NZ}, y \mapsto \text{MZ}, z \mapsto \text{NZ}]
\]
Example: Zero Analysis

1: \( x := 8 \)

2: \( y := x \)

3: \( z := 0 \)

4: \( x := x / y \)

5: \( y := y - 2 \)

6: \( z := 5 \)

\( \sigma = [] \)

\([x \rightarrow \text{NZ}]\)

\([x \rightarrow \text{NZ}, y \rightarrow \text{NZ}]\)

\([x \rightarrow \text{NZ}, y \rightarrow \text{NZ, } z \rightarrow \text{Z}]\)

\([x \rightarrow \text{NZ, } y \rightarrow \text{MZ, } z \rightarrow \text{MZ}]\)

\([x \rightarrow \text{NZ, } y \rightarrow \text{MZ, } z \rightarrow \text{MZ}]\)

\([x \rightarrow \text{NZ, } y \rightarrow \text{MZ, } z \rightarrow \text{MZ}]\)

\([x \rightarrow \text{NZ, } y \rightarrow \text{MZ, } z \rightarrow \text{NZ}]\)

\(\text{MZ-}_\alpha \alpha(2) = \text{MZ-}_\alpha \text{NZ} = \text{MZ}\)
Example: Zero Analysis

\[ \sigma = [] \]

1: \( x := 8 \)

2: \( y := x \)

3: \( z := 0 \)

4: \( x := x/y \)

5: \( y := y - 2 \)

6: \( z := 5 \)

\[ [x \mapsto \text{NZ}] \]

\[ [x \mapsto \text{NZ}, y \mapsto \text{NZ}] \]

\[ [x \mapsto \text{NZ}, y \mapsto \text{MZ}, z \mapsto \text{MZ}] \]

\[ [x \mapsto \text{MZ}, y \mapsto \text{MZ}, z \mapsto \text{MZ}] \]

\[ [x \mapsto \text{MZ}, y \mapsto \text{MZ}, z \mapsto \text{MZ}] \]

\[ [x \mapsto \text{NZ}, y \mapsto \text{MZ}, z \mapsto \text{NZ}] \]
Example: Zero Analysis

Convergence

Iteration #2

\[ \sigma = [] \]

\[ x \rightarrow NZ \]

\[ x \rightarrow NZ, y \rightarrow NZ \]

\[ x \rightarrow NZ, y \rightarrow NZ, z \rightarrow Z \]

\[ x \rightarrow NZ, y \rightarrow MZ, z \rightarrow MZ \]

\[ x \rightarrow NZ, y \rightarrow MZ, z \rightarrow MZ \]

\[ x \rightarrow NZ, y \rightarrow MZ, z \rightarrow N \]

Iteration #3

\[ \sigma = [] \]

\[ x \rightarrow NZ \]

\[ x \rightarrow NZ, y \rightarrow NZ \]

\[ x \rightarrow NZ, y \rightarrow NZ, z \rightarrow Z \]

\[ x \rightarrow NZ, y \rightarrow MZ, z \rightarrow MZ \]

\[ x \rightarrow NZ, y \rightarrow MZ, z \rightarrow MZ \]

\[ x \rightarrow NZ, y \rightarrow MZ, z \rightarrow N \]

\[ x \rightarrow NZ, y \rightarrow MZ, z \rightarrow NZ \]
Example: Zero Analysis
Interpreting Results

σ = []
x := 8;
σ = [x → NZ]
y := x;
σ = [x → NZ, y → NZ]
z := 0;
σ = [x → NZ, y → NZ, z → Z]
while y > -1 do
    σ = [x → NZ, y → NZ, z → Z]
x := x / y;
σ = [x → NZ, y → NZ, z → Z]
y := y - 2;
σ = [x → NZ, y → NZ, z → Z]
z := 5;
σ = [x → NZ, y → NZ, z → NZ]

Warning: This program *might* produce a division by zero
Example: Zero Analysis
Interpreting Results

Interpreting the result:
- Visit each expression of the form X/Y in the program
- Look at the result from the zero analysis for that program point for the variable Y
  - IF Y = NZ, OK!

Static analysis is conservative:
- if there is a possibility of Y being zero
  => the analysis results are guaranteed to contain Y->Z or Y->MZ
  so only warn the user about these cases

However, in case Y->MZ
  => there is a possibility we are warning about a false positive
  e.g., will warn about possible division by zero:

```
y = 3;
y := y-1;
x = 6/y
```

False positives can be reduced by improving the precision of the analysis
  e.g., by improving transfer functions
More Efficient Iteration: Worklist

add statements to worklist if their dependencies might change
fixpoint reached when worklist is empty

Compute $\text{out}[n] \text{ for each } n \in N$:

$\text{out}[n] := \bot$

$\text{work}.\text{add} = \{\text{entry}\}$

WHILE $\text{work}$ is not empty:

$n := \text{work}.\text{pop}();$

$\text{in'}[n] := \bigoplus \{ \text{out}[m] \mid m \in \text{pred}(n) \}$

$\text{out'}[n] := \text{transfer}[n](\text{in'}[n])$

IF $!(\text{out'}[n] \subseteq \text{out}[n])$

for each $m \in \text{succ}(n)$ work.add($m$);

$\text{out}[n] := \text{out'}[n]; \text{ in}[n] := \text{in'}[n];$
Termination: Monotonicity of Transfer Functions

- A function $f$ on a partial order is **monotonic** if
  - if $x \leq y$ then $f(x) \leq f(y)$

- Functions for computing $\text{In}(s)$ and $\text{Out}(s)$ are monotonic
  - $\text{In}(s) := \bigcap_{s' \in \text{pred}(s)} \text{Out}(s')$
  - $\text{temp} := \text{Gen}(s) \cup (\text{In}(s) - \text{Kill}(s))$  
    
- Putting them together: $\text{temp} := f_s(\bigcap_{s' \in \text{pred}(s)} \text{Out}(s'))$
Termination: Lattice Structure

- A **descending chain** in a lattice is a sequence \( x_0 < x_1 < \ldots \)
- The **height of a lattice** is the length of the longest descending chain in the lattice
- Then, dataflow must terminate in \( O(nk) \) time
  - \( n = \# \) of statements in program
  - \( k = \) height of lattice
  - assumes meet operation and transfer function takes \( O(1) \) time
Termination: Fixpoint Theorem

in each iteration of the algorithm
either the worklist gets smaller
or \( \text{out}[n] \) decreases in precision (i.e., ascends the lattice) for some \( n \)
because transfer function is monotonic
with lattice of finite height / complete lattice with ACC
\( \text{out}[n] \) can decrease in precision only finitely often

Theorem (Fixpoint Theorem by Tarski and Knaster)

Let \( (D, \sqsubseteq) \) be a complete lattice satisfying ACC and \( \Phi : D \rightarrow D \)
monotonic. Then

\[
\text{fix}(\Phi) := \bigsqcup \{ \Phi^k(\bot) \mid k \in \mathbb{N} \}
\]

is the least fixpoint of \( \Phi \) where

\[
\Phi^0(d) := d \text{ and } \Phi^{k+1}(d) := \Phi(\Phi^k(d)).
\]

Requirements for dataflow analysis

The domain must be a complete lattice satisfying ACC, and all transfer functions must be a monotonic.

For details, see:
Example: Live Variable Analysis

Lattice: \( P(\{x, yz\}) \)
- bot = \{\}, top={x,y,z}, \cup, \subseteq

- Transfer:
  - \( F(\sigma, e) = \sigma \cup \text{vars}(e) \)
  - \( F(\sigma, x:=e) = (\sigma - \{ x \}) \cup \text{vars}(e) \)
  - \( F(\sigma, \text{any other}) = \sigma \)

\[ \text{out}[n] := \cup \{ \text{in}[m] \mid m \in \text{succ}(n) \} \]
\[ \text{in}[n] := \text{transfer}[n](\text{out}[n]) \]
Example: Live Variable Analysis

Lattice: $P\{x, y, z\}$

Transfer function:
- $F(\sigma, e) = \sigma \cup \text{vars}(e)$
- $F(\sigma, x:=e) = (\sigma - \{ x \}) \cup \text{vars}(e)$
- $F(\sigma, \text{other case}) = \sigma$

\[ \text{out}[n] := \sigma \cup \{ \text{in}[m] | m \in \text{succ}(n) \} \]
\[ \text{in}[n] := F(\text{out}[n], \text{stm}[n]) \]

Compute $\text{in}[n]$ for each $n \in N$:
\[ \text{out}[n] := T \]
\[ \text{work}.add = \{ \text{exit} \} \]
WILE work is not empty:
\[ \text{n} := \text{work}.pop(); \]
\[ \text{out}'[n] := \sigma \cup \{ \text{in}[m] | m \in \text{succ}(n) \} \]
\[ \text{in}'[n] := \text{transfer}[n](\text{out}'[n]) \]
IF $\neg (\text{in}'[n] \subseteq \text{in}[n])$
\[ \text{for all } m \in \text{pred}(n) : \text{work}.add(m); \]
\[ \text{out}[n] := \text{out}'[n]; \]
\[ \text{in}[n] := \text{in}'[n]; \]
Example: Live Variable Analysis

Lattice: $P\{x, y, z\}$
Transfer function:
- $F(\sigma, e) = \sigma \cup \text{vars}(e)$
- $F(\sigma, x:=e) = (\sigma - \{ x \}) \cup \text{vars}(e)$
- $F(\sigma, \text{other case}) = \sigma$

$\text{out}[n] := \bigcup \{ \text{in}[m] \mid m \in \text{succ}(n) \}$
$\text{in}[n] := F(\text{out}[n], \text{stm}[n])$

Compute $\text{in}[n]$ for each $n \in N$:
$\text{out}[n] := T$
$\text{work}.add = \{ \text{exit} \}$

While $\text{work}$ is not empty:
- $n := \text{work}.pop();$
- $\text{out}'[n] := \bigcup \{ \text{in}[m] \mid m \in \text{succ}(n) \}$
- $\text{in}'[n] := \text{transfer}[n](\text{out}'[n])$
- If $(\text{in}'[n] \subseteq \text{in}[n])$
  - for all $m \in \text{pred}(n)$: $\text{work}.add(m);$
- $\text{out}[n] := \text{out}'[n];$
- $\text{in}[n] := \text{in}'[n];$

work = $\{ \text{exit} \}$
$n = \text{exit}$
out’[7] = $
in’ [7] = \{z\}$
work’ = $\{6\}$

7: ret $z$
Example: Live Variable Analysis

Lattice: \( P\{x, y, z\} \)
Transfer function:
- \( F(\sigma, e) = \sigma \cup \text{vars}(e) \)
- \( F(\sigma, x:=e) = (\sigma - \{x\}) \cup \text{vars}(e) \)
- \( F(\sigma, \text{other case}) = \sigma \)

\[
\begin{align*}
\text{Compute } \text{in}[n] \text{ for each } n \in N: & \\
\text{out}[n] := T & \\
\text{work.add} = \{\text{exit}\} & \\
\text{WILE work is not empty:} & \\
& \text{work} = \{6\} \\
& n = 6 \\
& \text{out'}[6] = \{z\} \\
& \text{in'}[6] = \{z\} \\
& \text{work'} = \{3\} \\
\end{align*}
\]

\[
\begin{align*}
1: y & := x \\
2: z & := 1 \\
3: y & > 1 \\
4: z & := z \times y \\
5: y & := y - 1 \\
6: y & := 0 \\
7: \text{return } z
\end{align*}
\]
**Example: Live Variable Analysis**

**Lattice:** \( P\{x, y, z\} \)

**Transfer function:**
- \( F(\sigma, e) = \sigma \cup \text{vars}(e) \)
- \( F(\sigma, x:=e) = (\sigma - \{x\}) \cup \text{vars}(e) \)
- \( F(\sigma, \text{other case}) = \sigma \)

\[
\begin{align*}
\text{out}[n] &:= \sigma \cup \{ \text{in}[m] \mid m \in \text{succ}(n) \} \\
\text{in}[n] &:= F(\text{out}[n], \text{stm}[n])
\end{align*}
\]

**Compute \( \text{in}[n] \) for each \( n \in N \):**

\[
\begin{align*}
\text{out}[n] &:= T \\
\text{work.add} &:= \{ \text{exit} \}
\end{align*}
\]

**WILE** work is not empty:

\[
\begin{align*}
\text{n} &:= \text{work.pop}(); \\
\text{out}'[n] &:= \sigma \cup \{ \text{in}[m] \mid m \in \text{succ}(n) \} \\
\text{in}'[n] &:= \text{transfer}[n](\text{out}'[n]) \\
\text{IF ! (in}'[n] \subseteq \text{in}[n]) \\
\text{for all } m \in \text{pred}(n) &:\ \text{work.add}(m); \\
\text{out}[n] &:= \text{out}'[n]; \\
\text{in}[n] &:= \text{in}'[n];
\end{align*}
\]

\[
\begin{align*}
\text{work} &:= \{3\} \\
\text{n} &:= 3 \\
\text{out}'[3] &:= \{z\} \text{ (de 6 y 4)} \\
\text{in}'[3] &:= \{z,y\} \\
\text{work'} &:= \{2,5\}
\end{align*}
\]
Example: Live Variable Analysis

Lattice: $P\{x, y, z\}$
Transfer function:
- $F(\sigma, e) = \sigma \cup \text{vars}(e)$
- $F(\sigma, x:=e) = (\sigma - \{x\}) \cup \text{vars}(e)$
- $F(\sigma, \text{other case}) = \sigma$

$out[n] := \bigcup \{in[m] \mid m \in \text{succ}(n)\}$
$in[n] := F(out[n], \text{stm}[n])$

Compute $in[n]$ for each $n \in N$:
$out[n] := T$
work.add = \{exit\}
WILE work is not empty:
  n := work.pop();
  $out'[n] := \bigcup \{in[m] \mid m \in \text{succ}(n)\}$
  $in'[n] := \text{transfer}[n](out'[n])$
  IF !(in'[n] $\subseteq$ in[n])
    for all $m \in \text{pred}(n)$: work.add(m);
  $out[n] := out'[n];$
  $in[n] := in'[n];$

work = \{2, 5\}
n = 5
$out'[5] = \{z, y\}$
in'[5] = \{z, y\}
work' = \{2, 4\}
Example: Live Variable Analysis

Lattice: \( P\{x, y, z\} \)
Transfer function:
- \( F(\sigma, e) = \sigma \cup \text{vars}(e) \)
- \( F(\sigma, x:=e) = (\sigma - \{x\}) \cup \text{vars}(e) \)
- \( F(\sigma, \text{other case}) = \sigma \)

\[ \begin{align*}
\text{out}[n] & := \bigcup \{ \text{in}[m] \mid m \in \text{succ}(n) \} \\
\text{in}[n] & := F(\text{out}[n], \text{stm}[n])
\end{align*} \]

Compute \( \text{in}[n] \) for each \( n \in N \):
- \( \text{out}[n] := T \)
- work.add = \{exit\}
- WILE work is not empty:
  - \( n := \text{work}.\text{pop}() \)
  - \( \text{out}'[n] := \bigcup \{ \text{in}[m] \mid m \in \text{succ}(n) \} \)
  - \( \text{in}'[n] := \text{transfer}[n](\text{out}'[n]) \)
  - IF \( !\left( \text{in}'[n] \subseteq \text{in}[n] \right) \)
    - for all \( m \in \text{pred}(n) \): work.add(m);
  - \( \text{out}[n] := \text{out}'[n] \)
  - \( \text{in}[n] := \text{in}'[n] \)

work = \{2, 4\}
n = 4
out'[4] = \{z, y\}
in' [4] = \{z, y\}
work' = \{2, 3\}
Example: Live Variable Analysis

Lattice: \( P(\{x, y, z\}) \)

Transfer function:
- \( F(\sigma, e) = \sigma \cup \text{vars}(e) \)
- \( F(\sigma, x:=e) = (\sigma - \{x\}) \cup \text{vars}(e) \)
- \( F(\sigma, \text{other case}) = \sigma \)

\[
\begin{align*}
\text{out}[n] & := \bigcup \{ \text{in}[m] \mid m \in \text{succ}(n) \} \\
\text{in}[n] & := F(\text{out}[n], \text{stm}[n])
\end{align*}
\]

Compute \( \text{in}[n] \) for each \( n \in N \):
\[
\begin{align*}
\text{out}[n] & := T \\
\text{work}.add & = \{\text{exit}\}
\end{align*}
\]

WILE \( \text{work} \) is not empty:
\[
\begin{align*}
\text{n} & := \text{work}.pop(); \\
\text{out}'[n] & := \bigcup \{ \text{in}[m] \mid m \in \text{succ}(n) \} \\
\text{in}'[n] & := \text{transfer}[n](\text{out}'[n]) \\
\text{if} \! (\text{in}'[n] \subseteq \text{in}[n]) \\
& \quad \text{for all} \ m \in \text{pred}(n) : \text{work}.add(m); \\
\text{out}[n] & := \text{out}'[n]; \\
\text{in}[n] & := \text{in}'[n];
\end{align*}
\]

work = \( \{2, 3\} \)

\( n = 3 \)

out'[3] = \( \{z, y\} \) (of 6 and 4)

\( \text{in}'[3] = \{z, y\} \)

work' = \( \{2\} \)
Example: Live Variable Analysis

Lattice: \( P\{x, y, z\} \)

Transfer function:
- \( F(\sigma, e) = \sigma \cup \text{vars}(e) \)
- \( F(\sigma, x:=e) = (\sigma - \{ x \}) \cup \text{vars}(e) \)
- \( F(\sigma, \text{other case}) = \sigma \)

\( \text{out}[n] := \cup \{ \text{in}[m] \mid m \in \text{succ}(n) \} \)

\( \text{in}[n] := F(\text{out}[n], \text{stm}[n]) \)

Compute \( \text{in}[n] \) for each \( n \in N \):

\( \text{out}[n] := T \)

work.add = \{exit\}

WILE work is not empty:

\( n:= \text{work.pop}(); \)

\( \text{out}'[n] := \cup \{ \text{in}[m] \mid m \in \text{succ}(n) \} \)

\( \text{in}'[n] := \text{transfer}[n](\text{out}'[n]) \)

IF \( ! (\text{in}'[n] \subseteq \text{in}[n]) \)

for all \( m \in \text{pred}(n) \): work.add(m);

\( \text{out}[n] := \text{out}'[n]; \)

\( \text{in}[n] := \text{in}'[n]; \)

work = \{2\}

\( n = 2 \)

out'[2] = \{z, y\}

in' [2] = \{y\}

work' = \{1\}
Example: Live Variable Analysis

Lattice: \( P(\{x, y, z\}) \)

Transfer function:
- \( F(\sigma, e) = \sigma \cup \text{vars}(e) \)
- \( F(\sigma, x:=e) = (\sigma - \{x\}) \cup \text{vars}(e) \)
- \( F(\sigma, \text{other case}) = \sigma \)

\[
\begin{align*}
\text{out}[n] &:= \cup \{ \text{in}[m] \mid m \in \text{succ}(n) \} \\
\text{in}[n] &:= F(\text{out}[n], \text{stm}[n])
\end{align*}
\]

Compute \( \text{in}[n] \) for each \( n \in N \):
\[
\begin{align*}
\text{out}[n] &:= T \\
\text{work.add} &= \{\text{exit}\} \\
\text{WILE} \text{ work is not empty:} & \\
& \quad n := \text{work.pop}(); \\
& \quad \text{out}'[n] := \cup \{ \text{in}[m] \mid m \in \text{succ}(n) \} \\
& \quad \text{in}'[n] := \text{transfer}[n](\text{out}'[n]) \\
& \quad \text{IF } (\text{in}'[n] \subseteq \text{in}[n]) \\
& \quad \quad \text{for all } m \in \text{pred}(n): \text{work.add}(m); \\
& \quad \text{out}[n] := \text{out}'[n]; \\
& \quad \text{in}[n] := \text{in}'[n]; \\
\end{align*}
\]

work = \{1\}
\[
\begin{align*}
n &= 1 \\
\text{out}'[1] &= \{y\} \\
\text{in}'[1] &= \{x\} \\
\text{work'} &= \{\text{entry}\}
\end{align*}
\]
Terminology: Flow Sensitivity

• Data flow analysis is *flow-sensitive*
  - The order of statements is taken into account
  - I.e., we keep track of facts per program point

• Alternative: *Flow-insensitive* analysis
  - Analysis the same regardless of statement order
  - Standard example: types
    - /* x : int */ x := ... /* x : int */
  - Variables updated by a procedure

    - $M(x := e) = \{ x \}$
    - $M(S_1; S_2) = M(S_1) \cup M(S_2)$

Observe that $M(S_1; S_2) = M(S_2; S_1)$
Terminology: Path Sensitivity

• Consider the following program:

```c
FILE *pFile = NULL;
if (debug) {
    pFile = fopen("debuglog.txt", "a")
}
...
if (debug) {
    fputs("foo", pFile);
}
```

• Can `pFile` be NULL when used for `fputs`?

• What dataflow analysis could we use to determine if it is?
Terminology: Path Sensitivity

- `pFile`: Not a standard one ... most dataflow analyses are path insensitive.
Terminology: Inter vs Intra-procedural

- An analysis that models only a single function at a time is *intraprocedural*
- An analysis that takes multiple functions into account is *interprocedural*

Function calls kill dataflow facts (have to assume worst (e.g., top element is returned))

-one of the next lectures!