Program Analysis – Lecture 6

Data Flow Analysis (Part 3)

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Winter 2019/2020

Warm-up Quiz

What does the following code print?

```
function f(a,b) {
  var x;
  for (var i = 0; i < arguments.length; i++) {
    x += arguments[i];
  }
  console.log(x);
}
f(1,2,3);
```

3 **6** **NaN** **Nothing**

Warm-up Quiz

What does the following code print?

```
function f(a,b) {
  var x;
  for (var i = 0; i < arguments.length; i++) {
    x += arguments[i];
  }
  console.log(x);
}
f(1,2,3);
```

3 **6** **NaN** **Nothing**

Warm-up Quiz

What does the following code print?

```
function f(a,b) {
  var x;
  for (var i = 0; i < arguments.length; i++) {
    x += arguments[i];
  }
  console.log(x);
}
f(1,2,3);
```

Array-like object that contains all three arguments

3 **6** **NaN** **Nothing**

Warm-up Quiz

What does the following code print?

```
function f(a,b) {
  var x;
  for (var i = 0; i < arguments.length; i++) {
    x += arguments[i];
  }
  console.log(x);
}
f(1,2,3);
```

Initialized to `undefined`

`undefined` + some number yields `NaN`

3 **6** **NaN** **Nothing**

Outline

- First example: Available expressions
- Basic principles
- More examples ←
- Solving data flow problems
- Inter-procedural analysis
- Sensitivities

Live Variables Analysis

Goal: For each statement, find variables that are may be "live" at the exit from the statement

- "Live": The variable is used before being redefined
- Useful, e.g., for identifying dead code
  - Bug detection: Dead assignments are typically unexpected
  - Optimization: Remove dead code

Example

```
x = 2;
y = 4;
x = 1;
if (y > x) {
  z = y;
} else {
  z = y * y;
  x = z;
}
```

Example

```
x = 2;
y = 4;
x = 1;
if (y > x) {
  z = y;
} else {
  z = y * y;
  x = z;
}
```

`x` is not live after this statement

Example

```
x = 2;
y = 4;
x = 1;
if (y > x) {
  z = y;
} else {
  z = y * y;
  x = z;
}
```

Both `x` and `y` are live after this statement

Defining the Analysis

- **Domain: All variables occurring in the code**
- **Direction: Backward**
- **Meet operator: Union**
  - Because we care about whether a variable *may* be used

Transfer Function

Defined via gen and $kill$ functions (again)

- Returns set of variables
- Backward analysis, i.e., returned variables are live at entry of statement

Function $gen(s)$

- All variables v that are used in s

Function $kill(s)$

- If s assigns to x, then it kills x
- Otherwise: Empty set

Defining the Analysis (2)

- **Transfer function:**

  $$LV_{entry}(s) = (LV_{exit}(s) \setminus kill(s)) \cup gen(S)$$

- **Boundary condition: Final node starts with no live variables**
  - $LV_{exit}(finalNode) = \emptyset$
- **Initially, all nodes have no live variables**

Quiz: Live Variables

```
x = 2;
y = 4;
x = 1;
if (y > x) {
  z = y;
} else {
  z = y * y;
  x = z;
}
```

Compute the live variables before and after every statement. How many variables are in total in the sets you're computing?

Quiz: Live Variables

```
1  x = 2
2  y = 4
3  x = 1
4  if ( y > x ) {
5      z = y
   } else {
6      z = y * y
7      x = z
   }
```

s	$LV_{entry}(s)$	$LV_{exit}(s)$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{y\}$
3	$\{y\}$	$\{x, y\}$
4	$\{x, y\}$	$\{y\}$
5	$\{y\}$	$\emptyset$
6	$\{y\}$	$\{z\}$
7	$\{z\}$	$\emptyset$

$$\sum = 11$$

Outline

- First example: Available expressions
- Basic principles
- More examples
- Solving data flow problems ←
- Inter-procedural analysis
- Sensitivities

Data Flow Equations

- **Transfer functions yield data flow equations for each statement**
  - At entry, e.g., $AE_{entry}(2) = ...$
  - At exit, e.g., $AE_{exit}(3) = ...$
- **How to solve these equations?**
  - Goal: Fix point, i.e., nothing changes anymore

Data Flow Equations

- **Transfer functions yield data flow equations for each statement**
  - At entry, e.g., $AE_{entry}(2) = ...$
  - At exit, e.g., $AE_{exit}(3) = ...$

May depend on each other

- **How to solve these equations?**
  - Goal: Fix point, i.e., nothing changes anymore

Naive Algorithm

Round-robin, iterative algorithm

- For each statement s
  - Initialize entry and exit set of s
- While sets are still changing
  - For each statement s
    - Update entry set of s by applying meet operator to exit sets of incoming statements
    - Compute exit set of s based on its entry set

Naive Algorithm

Round-robin, iterative algorithm

- For each statement s
  - Initialize entry and exit set of s
- While sets are still changing
  - For each statement s
    - Update entry set of s by applying meet operator to exit sets of incoming statements
    - Compute exit set of s based on its entry set

Repeatedly computes each set, even if the input hasn't changed

Work List Algorithm

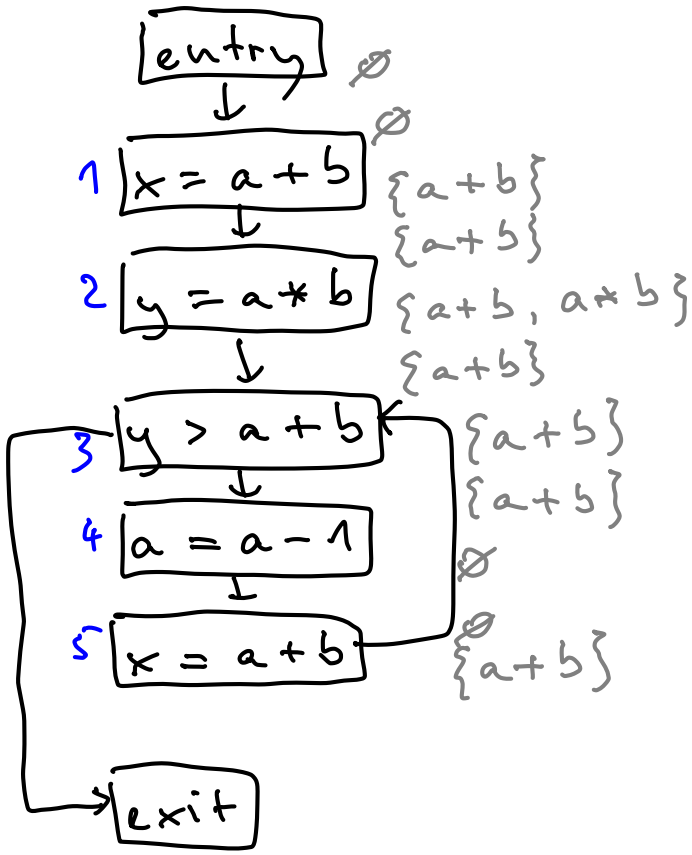
- For each statement s: Initialize entry and exit set
- Initialize W with initial/final node (for forward/backward analysis)
- While W not empty
  - Remove a statement s from W
  - Update entry set of s by applying meet operator to exit sets of incoming statements
  - Compute exit set of s based on its entry set
  - If exit set has changed (or statement visited for the first time): Add successors of s to W

Work List Algorithm

- For each statement s: Initialize entry and exit set
- Initialize $\boxed{W}$ with initial/final node (for forward/backward analysis)
- While $\boxed{W}$ not empty
  - Remove a statement s from W
  - Update entry set of s by applying meet operator to exit sets of incoming statements
  - Compute exit set of s based on its entry set
  - If exit set has changed (or statement visited for the first time): Add successors of s to W

Work list: Statements that need to be processed

Work List Algorithm: Example (Avail. Expr.)



Convergence

Will it always terminate?

- In principle, work list algorithms may run forever
- Impose constraints to ensure termination
  - Domain of analysis: Partial order with finite height
    - Not infinite ascending chains $X1 < X2 < ...$
  - Transfer function and meet operator:

    Monotonic w.r.t. partial order
    - Sets stay the same or grow larger

Convergence

Will it always terminate?

- In principle, work list algorithms may run forever
- Impose constraints to ensure termination
  - Domain of analysis: Partial order with finite height
    - Not infinite ascending chains $X1 < X2 < ...$
  - Transfer function and meet operator:

    Monotonic w.r.t. partial order
    - Sets stay the same or grow larger

Monotone framework

Outline

- First example: Available expressions
- Basic principles
- More examples
- Solving data flow problems
- Inter-procedural analysis ←
- Sensitivities

Intra- vs. Inter-procedural

- **Intra-procedural analysis**
  - Reason about a function in isolation
- **Inter-procedural analysis**
  - Reason about multiple functions
  - Calls and returns
- **Data flow analyses considered so far: Intra-procedural**

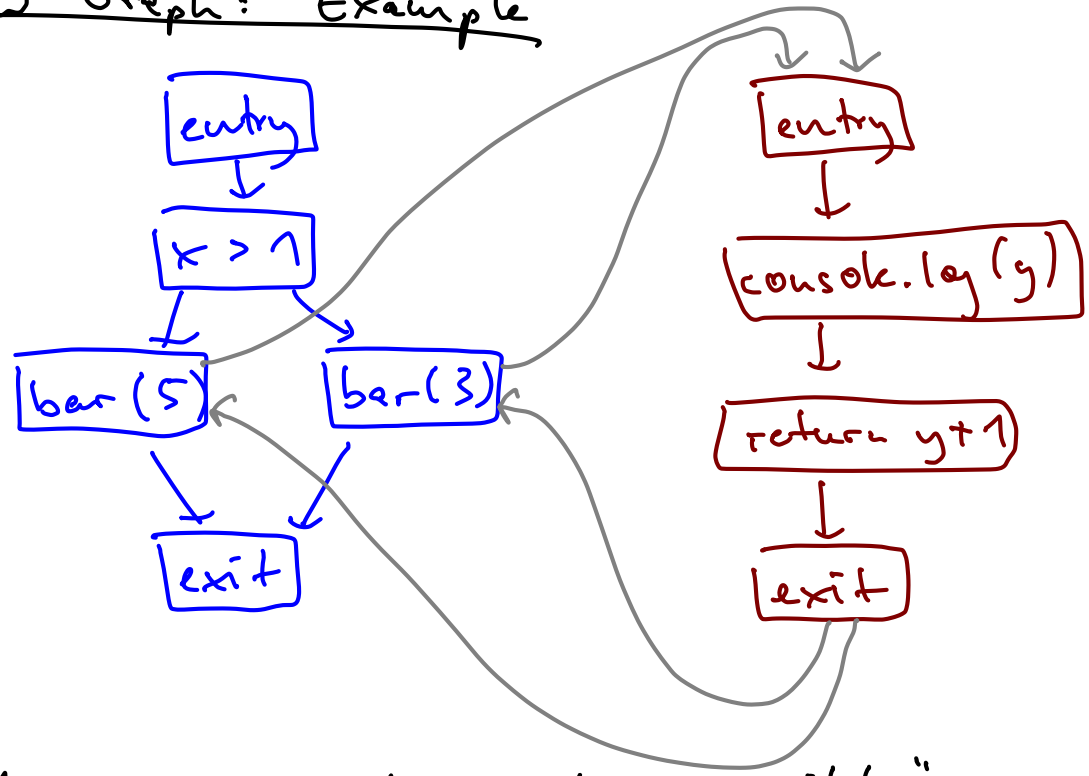
Inter-procedural Control Flow

- One control flow graph per function
- Connect call sites to entry node of callee
- Connect exit node back to call site

Inter-procedural Control Flow Graph: Example

```
function foo (x) {
    if (x > 1)
        z = bar (5)
    else
        z = bar (3)
}

function bar (y) {
    console.log (y)
    return y+1
}
```



Analysis considers only "possible" inter-proc. flows:

- After return, don't enter again
- When returning, go back to call site

Propagating Information

- **Arguments passed into call**
  - Propagate to formal parameters of callee
- **Return value**
  - Propagate back to caller
- **Local variables**
  - Do not propagate into callee
  - Instead, when call returned, continue with state just before call

Propagating Information

- **Arguments passed into call**
  - Propagate to formal parameters of callee
- **Return value**
  - Propagate back to caller
- **Local variables**
  - Do not propagate into callee
  - Instead, when call returned, continue with state just before call

For backward analysis: Everything in reverse

Outline

- First example: Available expressions
- Basic principles
- More examples
- Solving data flow problems
- Inter-procedural analysis
- Sensitivities ←

Sensitivities

Every static analysis: Sensitivities

- Flow-sensitive: Takes into account the order of statements
- Path-sensitive: Takes into account the predicates at conditional branches
- Context-senstive (inter-procedural analysis only): Takes into account the specific call site that leads into another function

Flow sensitivity: Example

```
if (..) {
        x = 3
        x = 5
}
```

← Value of x?

Flow-sensitive: 5

Flow-insensitive: 3 or 5

Path sensitivity: Example

```
x = 0
if (a > 0) {
    x = 1
} else {
    x = 2
}
if (a > 0) {
    x += 3
}
```

← Can x be 5?

Path-sensitive: no

Path-insensitive: yes

Context sensitivity: Example

```
function f(x) {
    if (x)
        g(3)
    else
        g(5)
}
function g(y) {

}
```

← What are possible values of y?

Context-insensitive: 3 or 5 (conflated into one piece of inform.)

Context-sensitive: 3 or 5 (analyzes two separate cases)

Quiz: Sensitivities

Consider an intra-procedural data flow analysis (specifically: live variables analysis).

What sensitivities does it have?

https://ilias3.uni-stuttgart.de/vote/KN2I

Quiz: Sensitivities

Consider an intra-procedural data flow analysis (specifically: live variables analysis).

What sensitivities does it have?

- Flow-sensitive: Yes (every data flow analysis)
- Path-sensitive: No (doesn't track predicates)
- Context-sensitive: Irrelevant (because intra-procedural)

https://ilias3.uni-stuttgart.de/vote/KN2I

Outline

- First example: Available expressions
- Basic principles
- More examples
- Solving data flow problems
- Inter-procedural analysis
- Sensitivities ✔