Program Analysis

Data Flow Analysis (Part 3)

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Winter 2023/2024

Warm-up Quiz

What does this JavaScript code print?

```
var a, b;
var x = {};
x[a] = 23;
console.log(x[b]);
```

Nothing **23** **undefined** **false**

Warm-up Quiz

What does this JavaScript code print?

```
var a, b;
var x = {};
x[a] = 23;
console.log(x[b]);
```

Nothing | **23** | undefined | false

Warm-up Quiz

What does this JavaScript code print?

```
var a, b;
var x = {};
x[a] = 23;
console.log(x[b]);
```

Have value `undefined`

Write and then read "undefined" property of `x`

Nothing | **23** | undefined | false

Outline

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

Very Busy Expression Analysis

Goal: For each program point, find expressions that must be very busy

- ”Very busy”: On all future paths, expression will be used before any of the variables in it are redefined
- Useful for program optimizations, e.g., hoisting
  - Hoisting an expression: Pre-compute it, e.g., before entering a block, for later use

Example

```
if (a > b) {
  x = b - a;
  y = a - b;
} else {
  y = b - a;
  x = a - b;
}
```

Example

```
if (a > b) {
  x = b - a;
  y = a - b;
} else {
  y = b - a;
  x = a - b;
}
```

`a - b` and `b - a` are very busy here

Defining the Analysis

- **Domain: All non-trivial expressions occurring in the code**
- **Direction: Backward**
- **Meet operator: Intersection**
  - Because we care about very busy expressions that *must* be used

Defining the Analysis (2)

Transfer function:

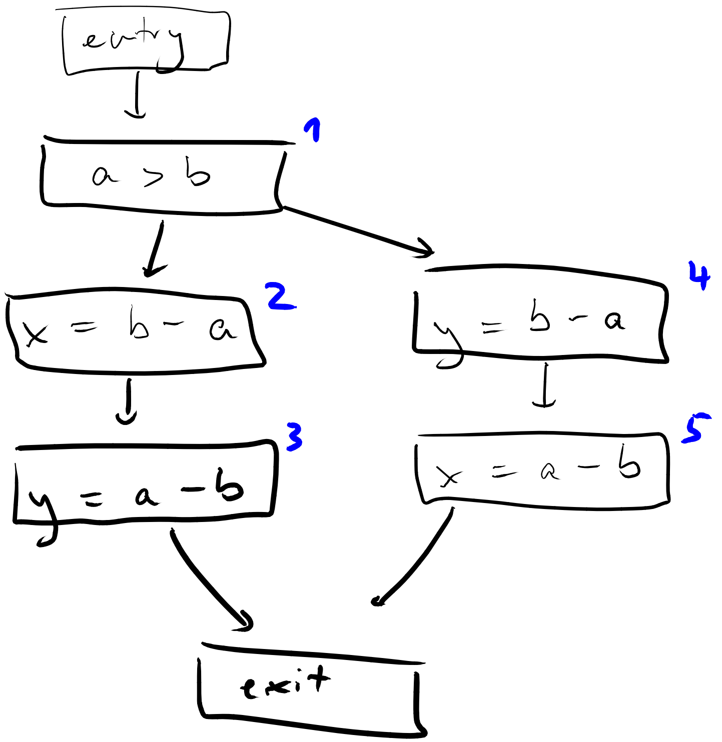
$$VB_{entry}(s) = (VB_{exit}(s) \setminus kill(s)) \cup gen(S)$$

- Backward analysis: Returns expressions that are very busy expressions at entry of statement
- Function $gen(s)$
  - All expressions e that appear in s
- Function $kill(s)$
  - If s assigns to x, all expressions in which x occurs
  - Otherwise: Empty set

Defining the Analysis (3)

- **Boundary condition: Final node starts with no very busy expressions**
  - $VB_{exit}(finalNode) = \emptyset$
- **Initially, all nodes have no very busy expressions**

Example: Very Busy Expressions Analysis



s	gen(s)	kill(s)
1	{a > b}	∅
2	{b - a}	∅
3	{a - b}	∅
4	{b - a}	∅
5	{a - b}	∅

s	$VB_{entry}(s)$	$VB_{exit}(s)$
1	{a-b, b-a, a > b}	{a-b, b-a}
2	{a - b, b - a}	{a - b}
3	{a - b}	∅
4	{a - b, b - a}	{a - b}
5	{a - b}	∅

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 unintended
  - 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

Defining the Analysis (2)

Transfer function:

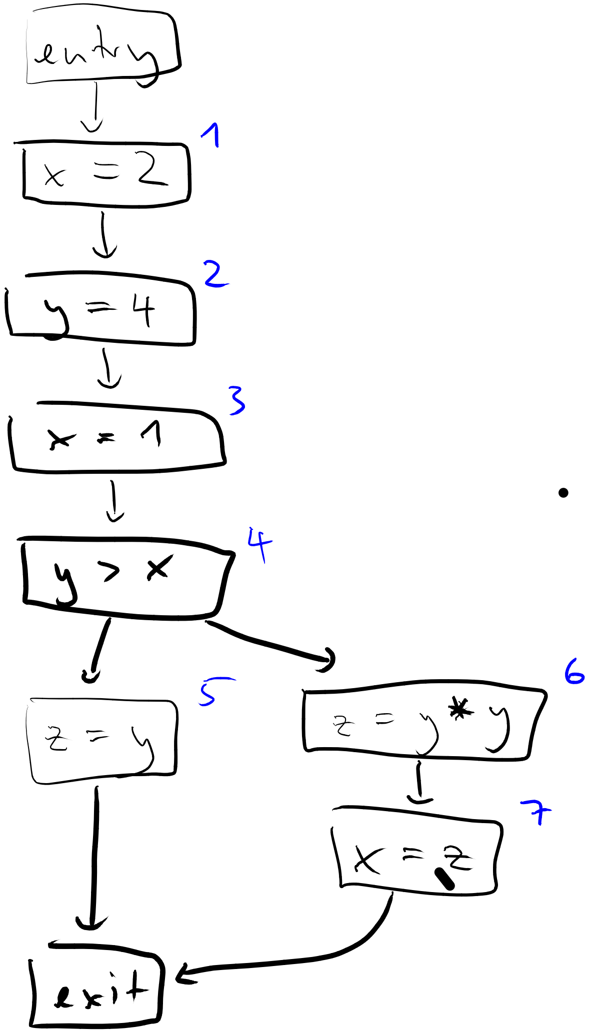
$$LV_{entry}(s) = (LV_{exit}(s) \setminus kill(s)) \cup gen(S)$$

- Backward analysis: Returns set of variables that 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 (3)

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

Example: Live Variables Analysis



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	$\{z\}$	$\emptyset$

$$LV_{exit}(4) = LV_{entry}(5) \cup LV_{entry}(6)$$

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., $AE_{exit}(3) = ...$

  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

Algorithms assume forward analysis (analogous for backward a.)

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

Algorithms assume forward analysis (analogous for backward a.)

Work List Algorithm

- For each statement s: Initialize entry and exit set
- Initialize W with initial node
- 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

Algorithms assume forward analysis (analogous for backward a.)

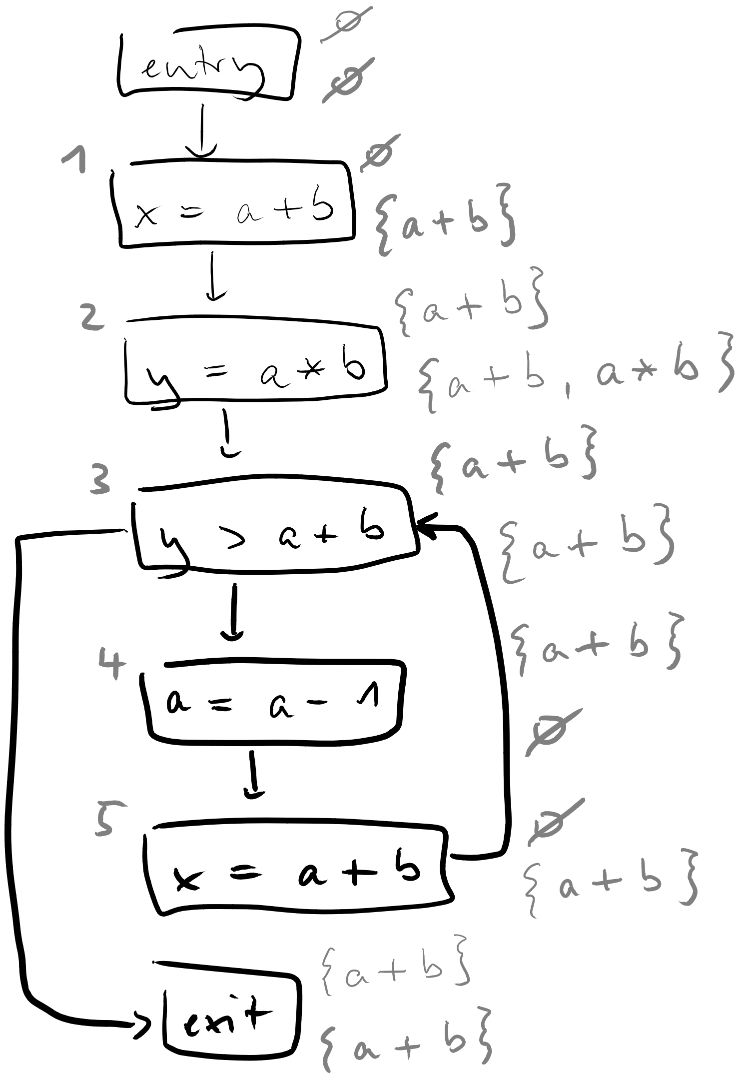
Work List Algorithm

- For each statement s: Initialize entry and exit set
- Initialize W with initial node
- 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: Statements that need to be processed

Algorithms assume forward analysis (analogous for backward a.)

Work List Algorithm: Example (Avail. Expr.)



W = ~~entry~~ ~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~exit~~ ~~5~~ ~~3~~

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
    - No 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
    - No 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

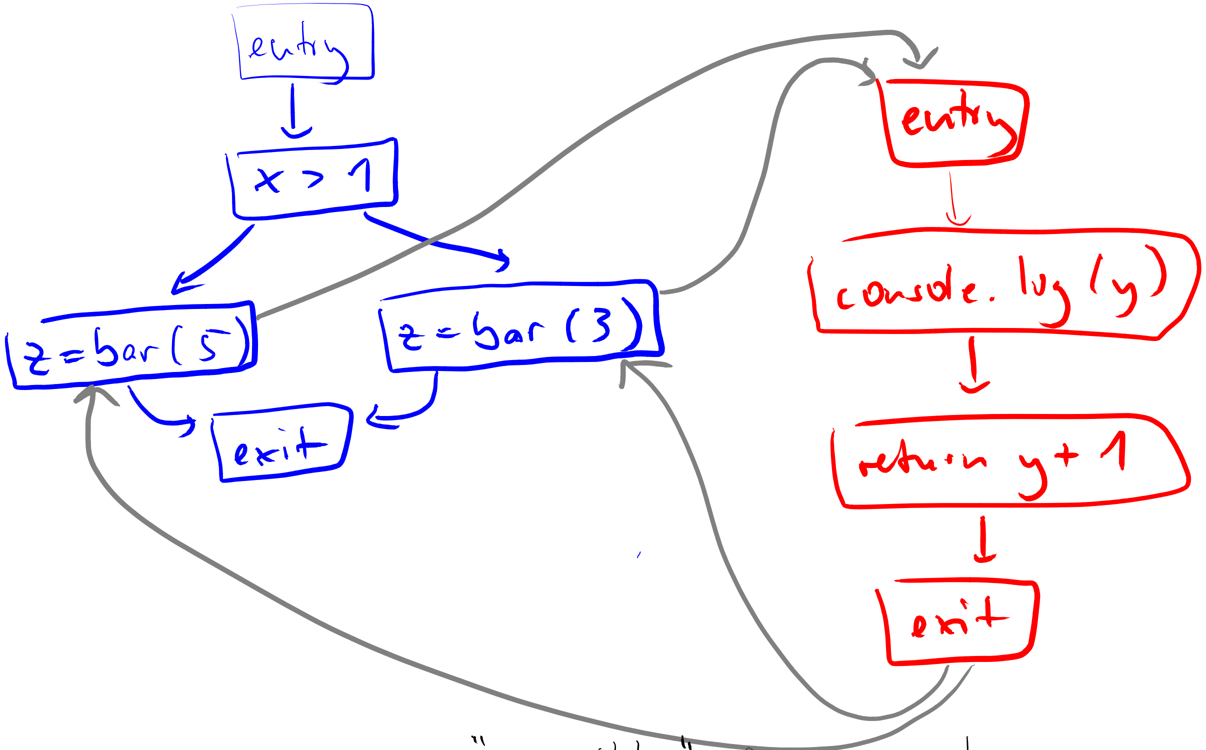
- **Intra-procedural analysis**
  - Reason about a function in isolation
- **Inter-procedural analysis**
  - Reason about multiple functions
  - Calls and returns
- **Data flow analyses considered so far: Intra-procedural**

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 "possible" flows only:

* 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 ←