软件分析与验证前沿

苏亭

软件科学与技术系

Outline

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

Example

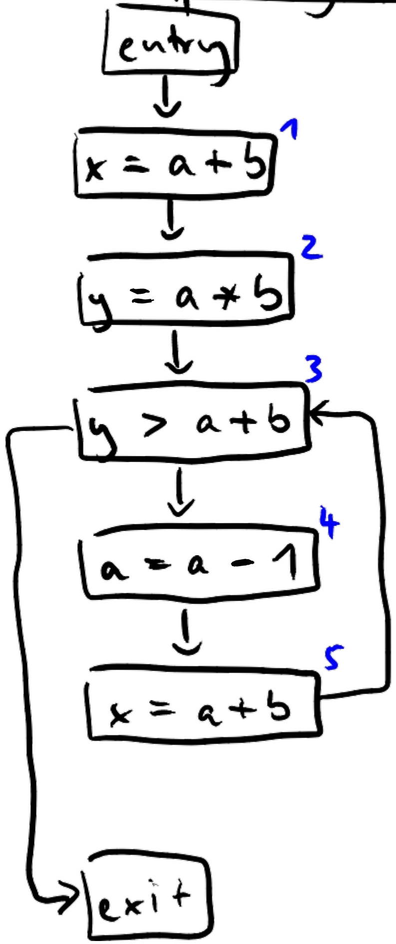
```
var x = a + b;
var y = a * b;
while(y > a + b) {
  a = a - 1;
  x = a + b;
}
```

Example

```
var x = a + b;
var y = a * b;
while(y > a + b) {
  a = a - 1;
  x = a + b;
}
```

Available every time execution reaches this point

Control flow graph



Non-trivial expressions

- $a + b$
- $a * b$
- $a - 1$

Transfer function for each statement:

Statement s	gen(s)	kill(s)
1	$\{a+b\}$	$\emptyset$
2	$\{a*b\}$	$\emptyset$
3	$\{a+b\}$	$\emptyset$
4	$\emptyset$	$\{a-1,\ a+b,\ a*b\}$
5	$\{a+b\}$	$\emptyset$

Data flow equations

$AE_{entry}(s)$ ... avail. express. at entry of s
$AE_{exit}(s)$ .. avail. express. at exit of s

$$AE_{entry}(1) = \emptyset$$
$$AE_{entry}(2) = AE_{exit}(1)$$
$$AE_{entry}(3) = AE_{exit}(2) \cap AE_{exit}(5)$$
$$AE_{entry}(4) = AE_{exit}(3)$$
$$AE_{entry}(5) = AE_{exit}(4)$$
$$AE_{exit}(1) = AE_{entry}(1) \cup \{a+b\}$$
$$AE_{exit}(2) = AE_{entry}(2) \cup \{a*b\}$$
$$AE_{exit}(3) = AE_{entry}(3) \cup \{a+b\}$$
$$AE_{exit}(4) = AE_{entry}(4) \setminus \{a+b, a*b, a-1\}$$
$$AE_{exit}(5) = AE_{entry}(5) \cup \{a+b\}$$

Solution of these equations:

s	$AE_{entry}(s)$	$AE_{exit}(s)$
1	$\emptyset$	$\{a+b\}$
2	$\{a+b\}$	$\{a+b, a*b\}$
3	$\{a+b\}$	$\{a+b\}$
4	$\{a+b\}$	$\emptyset$
5	$\emptyset$	$\{a+b\}$

Data Flow Equations

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

May depend on each other

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

Data flow equations

$AE_{entry}(s)$ ... avail. express. at entry of s
$AE_{exit}(s)$ .. avail. express. at exit of s

$$AE_{entry}(1) = \emptyset$$
$$AE_{entry}(2) = AE_{exit}(1)$$
$$AE_{entry}(3) = AE_{exit}(2) \cap AE_{exit}(5)$$
$$AE_{entry}(4) = AE_{exit}(3)$$
$$AE_{entry}(5) = AE_{exit}(4)$$
$$AE_{exit}(1) = AE_{entry}(1) \cup \{a+b\}$$
$$AE_{exit}(2) = AE_{entry}(2) \cup \{a*b\}$$
$$AE_{exit}(3) = AE_{entry}(3) \cup \{a+b\}$$
$$AE_{exit}(4) = AE_{entry}(4) \setminus \{a+b, a*b, a-1\}$$
$$AE_{exit}(5) = AE_{entry}(5) \cup \{a+b\}$$

Solution of these equations:

s	$AE_{entry}(s)$	$AE_{exit}(s)$
1	$\emptyset$	$\{a+b\}$
2	$\{a+b\}$	$\{a+b, a*b\}$
3	$\{a+b\}$	$\{a+b\}$
4	$\{a+b\}$	$\emptyset$
5	$\emptyset$	$\{a+b\}$

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.)

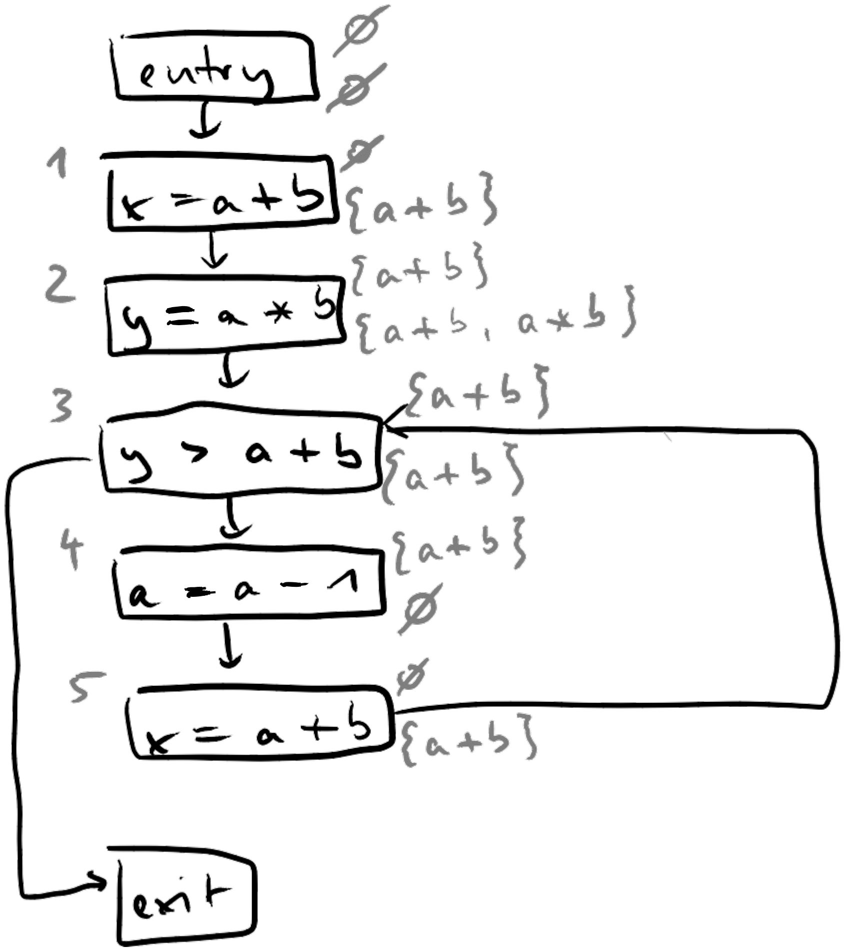
Work List Algorithm

- 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

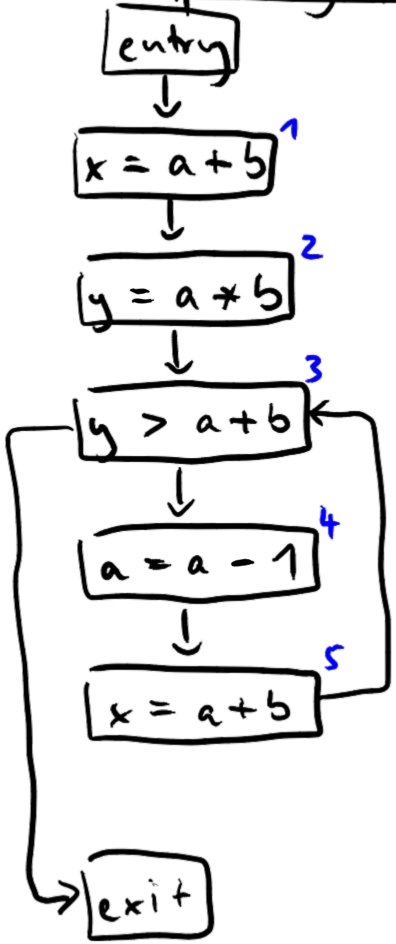
Algorithms assume forward analysis (analogous for backward a.)

Work List Algorithm: Example (Avail. Expr.)



W: ~~3~~ ~~4~~ ~~5~~ ~~3~~

Control flow graph



Non-trivial expressions

- $a + b$
- $a * b$
- $a - 1$

Transfer function for each statement:

Statement s	gen(s)	kill(s)
1	$\{a+b\}$	$\emptyset$
2	$\{a*b\}$	$\emptyset$
3	$\{a+b\}$	$\emptyset$
4	$\emptyset$	$\{a-1, a+b, a*b\}$
5	$\{a+b\}$	$\emptyset$

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 $X1 < X2 < \dots$
  - 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 $X1 < X2 < \ldots$
  - 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

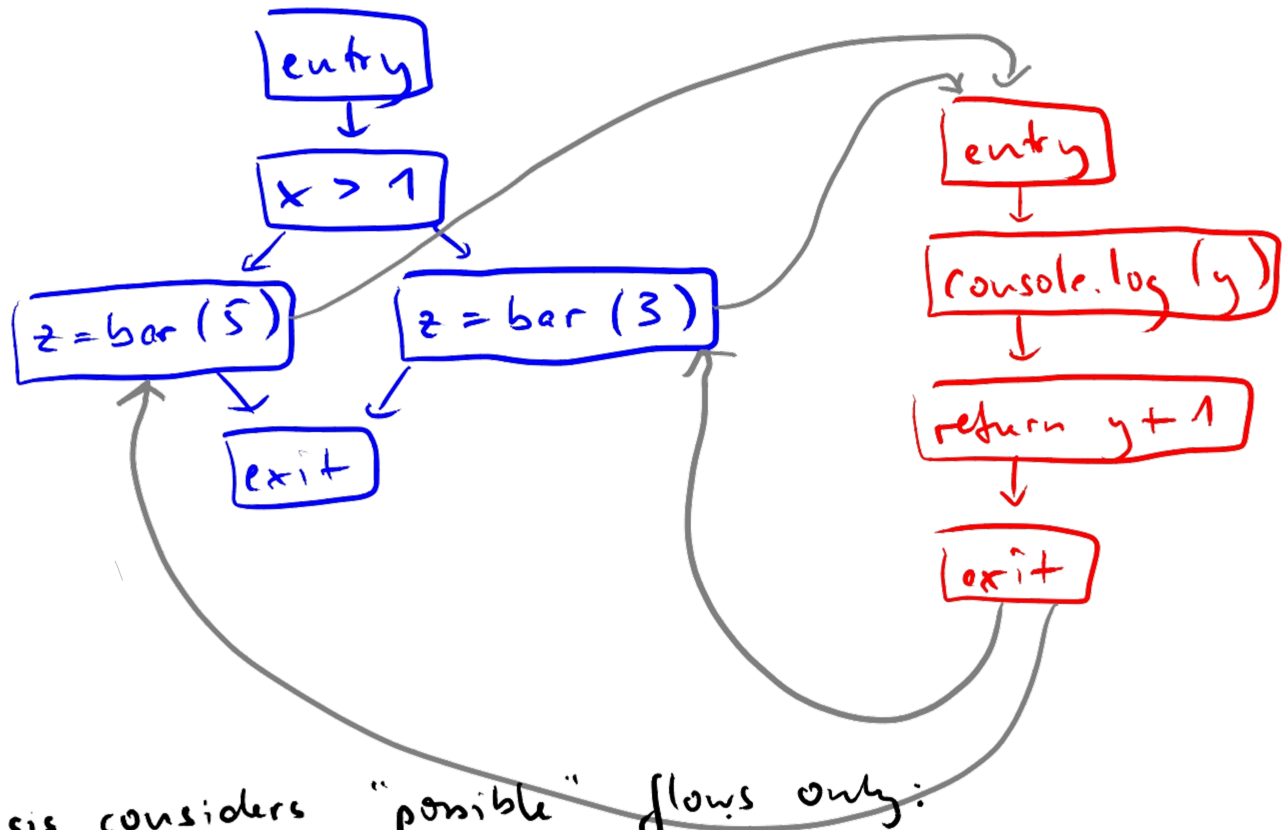
- One control flow graph per function
- Connect call sites to entry node of callee
- Connect exit node back to call site

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
}
```

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

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-sensitive (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 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 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

```
n = 1
function f(x) {
    if (x) {
        g(3)
    } else {
        n = 3
        g(5)
    }
}
function g(y) {

}
```

Can n be equal to y?

Context sensitivity: Example

```
n = 1
function f(x) {
    if (x) {
        g(3)
    } else {
        n = 3
        g(5)
    }
}
function g(y) {

}
```

Can n be equal to y?

context-sensitive: No

context-insensitive: Yes
(conflates all call sites of g)

Quiz: Sensitivities

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

What sensitivities does it have?

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)

Overall Pattern of Dataflow Analysis

$\boxed{\quad\quad}$ [n] = ($\boxed{\quad\quad}$ [n] - KILL[n]) ∪ GEN[n]

$\boxed{\quad\quad}$ [n] = $\boxed{\quad\quad}$ $\boxed{\quad\quad}$ [n']

n' ∈ $\boxed{\quad\quad}$ (n)

$\boxed{\quad\quad}$ $\boxed{\quad\quad}$ = IN or OUT

$\boxed{\quad\quad}$ = ∪ (may) or ∩ (must)

$\boxed{\quad\quad}$ = predecessors or successors

QUIZ: Available Expressions Analysis

$\boxed{\quad}$ [n] = ($\boxed{\quad}$ [n] - KILL[n]) ∪ GEN[n]

$\boxed{\quad}$ [n] = $\boxed{\quad}$ $\boxed{\quad}$ [n']

n' ∈ $\boxed{\quad}$ (n)

$\boxed{\quad}$ $\boxed{\quad}$ = IN or OUT

$\boxed{\quad}$ = ∪ (may) or ∩ (must)

$\boxed{\quad}$ = predecessors or successors

QUIZ: Available Expressions Analysis

$$\boxed{\text{OUT}}\,[n] = \left(\boxed{\text{IN}}\,[n] - \text{KILL}[n]\right) \cup \text{GEN}[n]$$

$$\boxed{\text{IN}}\,[n] = \underset{n' \in \boxed{\text{preds}}(n)}{\boxed{\cap}}\ \boxed{\text{OUT}}\,[n']$$

$\boxed{}$ $\boxed{}$ = IN or OUT

$\boxed{}$ = ∪ (may) or ∩ (must)

$\boxed{}$ = predecessors or successors

QUIZ: Live Variables Analysis

$\boxed{\quad}$ [n] = ($\boxed{\quad}$ [n] - KILL[n]) ∪ GEN[n]

$\boxed{\quad}$ [n] = $\boxed{\quad}$ $\boxed{\quad}$ [n’]

n’ ∈ $\boxed{\quad}$ (n)

$\boxed{\quad}$ $\boxed{\quad}$ = IN or OUT

$\boxed{\quad}$ = ∪ (may) or ∩ (must)

$\boxed{\quad}$ = predecessors or successors

QUIZ: Live Variables Analysis

$$\boxed{\text{IN}}[n] = (\boxed{\text{OUT}}[n] - \text{KILL}[n]) \cup \text{GEN}[n]$$

$$\boxed{\text{OUT}}[n] = \underset{n' \in \boxed{\text{succs}}(n)}{\boxed{\cup}} \boxed{\text{IN}}[n']$$

$\boxed{}$ $\boxed{}$ = IN or OUT

$\boxed{}$ = U (may) or ∩ (must)

$\boxed{}$ = predecessors or successors

Reaching Definitions Analysis

$$\boxed{\text{OUT}}[n] = (\boxed{\text{IN}}[n] - \text{KILL}[n]) \cup \text{GEN}[n]$$

$$\boxed{\text{IN}}[n] = \boxed{\bigcup}_{n' \in \boxed{\text{preds}}(n)} \boxed{\text{OUT}}[n']$$

$\boxed{\quad}$ $\boxed{\quad}$ = IN or OUT

$\boxed{\quad}$ = $\cup$ (may) or $\cap$ (must)

$\boxed{\quad}$ = predecessors or successors

Very Busy Expression Analysis

$\boxed{\text{IN}}$ [n] = ($\boxed{\text{OUT}}$ [n] - KILL[n]) ∪ GEN[n]

$\boxed{\text{OUT}}$ [n] = $\boxed{\cap}$ $\boxed{\text{IN}}$ [n']

n' ∈ $\boxed{\text{succs}}$ (n)

$\boxed{}$ $\boxed{}$ = IN or OUT

$\boxed{}$ = ∪ (may) or ∩ (must)

$\boxed{}$ = predecessors or successors

QUIZ: Classifying Dataflow Analyses

Match each analysis with its characteristics.

	May	Must
Forward		
Backward		

Very Busy Expressions

Reaching Definitions

Live Variables

Available Expressions

QUIZ: Classifying Dataflow Analyses

Match each analysis with its characteristics.

	May	Must
Forward	Reaching Definitions	Available Expressions
Backward	Live Variables	Very Busy Expressions

Assignment of this week

Exercise: Data Flow Analysis

1 Available Expressions [38 points]

Consider the following program in a toy language with syntax inspired by Python. Assume all variables are integers and operators have the obvious semantics.

```
x = a - 3
y = a + 3
if x > a + 3:
    a = a * 3
else:
    x = a + 3
end
y = a - 3
```

Your task is to perform the *Available Expressions* data flow analysis
Complete the following subtasks.

2 Live Variables [38 points]

Consider the following program in a toy language with syntax inspired by Python. Assume all variables are integers and operators have the obvious semantics.

```
x = 5
y = 0
while x > 0:
    x = x - 1
    while y < 10:
        y = x + y
end
y = 3
```

Your task is to perform the *Live Variables* data flow analysis, as presented in the lecture. Complete the following subtasks.

3 Constant Propagation [24 points]

3.1 Example [6 points]

The following is about a data flow analysis that was *not* discussed in the lecture: *Constant Propagation*. This analysis is commonly used by compilers to avoid unnecessary computations by recognizing and replacing expressions that depend only on constants, and hence, can be computed at compile time rather than at runtime.

First, write down an original (!) example of a simple program in pseudocode (similar to the programs above), without the optimization applied and which exposes an opportunity for a compiler to optimize via a *Constant Propagation* analysis: