

# Data Flow Analysis

YEGOR BUGAYENKO

Lecture #7 out of 10  
80 minutes

The slidedeck was presented by the author in this [YouTube Video](#)

All visual and text materials presented in this slidedeck are either originally made by the author or taken from public Internet sources, such as web sites. Copyright belongs to their respected authors.



Motivating Example

Method

Sensitivities

Most Common Types

Further Reading/Watching

Chapter #1:

# Motivating Example

## Unassigned Variable

Which code snippet has an error (“use of unassigned variable”)?

```
int f(int x) {  
    int a;  
    if (x > 10)  
        a = 42;  
    while (x++ < 5)  
        a = x;  
    return a + 1;  
}
```

```
int f(int x) {  
    int a;  
    if (x > 3)  
        a = 42;  
    while (x++ < 12)  
        a = x;  
    return a + 1;  
}
```

# ChatGPT vs. Clang Tidy

 Do you see any problems in this C++ code:

```
int f(int x) {
  int a;
  if (x > 10) { a = 42; }
  while (x++ < 5) { a = x; }
  return a + 1;
}
```

 Yes, there is a problem with this C++ code.

The variable `a` is not initialized if the condition `x > 10` is not met, which means that its value is undefined in the `while` loop. This can lead to undefined behavior in the program.

In C++, uninitialized variables can have unpredictable values, so it's important to always initialize them before they are used.

To fix this issue, we can initialize `a` to a default value before the `if` statement:

```
int f(int x) {
  int a = 0;
  if (x > 10) { a = 42; }
  while (x++ < 5) { a = x; }
  return a + 1;
}
```

With this change, the code should behave as expected without any undefined behavior.

```
/code/temp/cpp$ clang-tidy a.cpp --
1 warning generated.
/code/temp/cpp/a.cpp:5:12: warning: The left operand of '+' is a garbage value [clang-analyzer-core.UndefinedBinaryOperatorResult]
  return a + 1;
         ~ ^
/code/temp/cpp/a.cpp:2:3: note: 'a' declared without an initial value
  int a;
  ~~~~
/code/temp/cpp/a.cpp:3:7: note: Assuming 'x' is <= 10
  if (x > 10) { a = 42; }
  ~~~~~
/code/temp/cpp/a.cpp:3:3: note: Taking false branch
  if (x > 10) { a = 42; }
  ^
/code/temp/cpp/a.cpp:4:10: note: Assuming the condition is false
  while (x++ < 5) { a = x; }
  ~~~~~
/code/temp/cpp/a.cpp:4:3: note: Loop condition is false. Execution continues on line 5
  while (x++ < 5) { a = x; }
  ^
/code/temp/cpp/a.cpp:5:12: note: The left operand of '+' is a garbage value
  return a + 1;
         ~ ^
```

## UndefinedBehaviorSanitizer

Clang UndefinedBehaviorSanitizer (the dynamic analyzer) can detect the issue in runtime:

```
rultor@m2262:~$ cat > a.cpp
int f(int x) {
    int a;
    if (x > 10) { a = 42; }
    while (x++ < 5) { a = x; }
    return a + 1;
}
int main() {
    return f(7);
}
rultor@m2262:~$ clang -fsanitize=memory a.cpp -g
rultor@m2262:~$ ./a.out
==1494430==WARNING: MemorySanitizer: use-of-uninitialized-value
    #0 0x4950e4 in main /home/rultor/a.cpp:8:2
    #1 0x7fa900daf082 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x24082)
    #2 0x41c26d in _start (/home/rultor/a.out+0x41c26d)

SUMMARY: MemorySanitizer: use-of-uninitialized-value /home/rultor/a.cpp:8:2 in main
Exiting
```

## IntelliJ IDE

IntelliJ IDEA doesn't see the difference:

```
int f(int x) {  
    int a;  
    if (x > 3) { a = 42; }  
    while (x++ < 12) { a = x; }  
    return a + 1;  
}
```

Variable 'a' might not have been initialized  
Initialize variable 'a' More actions...

int a  
eo-maven-plugin

## Java Compiler

javac doesn't see the difference either:

```
/code/temp/java$ cat > A.java
class A {
    int f(int x) {
        int a;
        if (x > 3) { a = 42; }
        while (x++ < 12) { a = x; }
        return a + 1;
    }
}
/code/temp/java$ javac A.java
A.java:6: error: variable a might not have been initialized
        return a + 1;
               ^
1 error
```



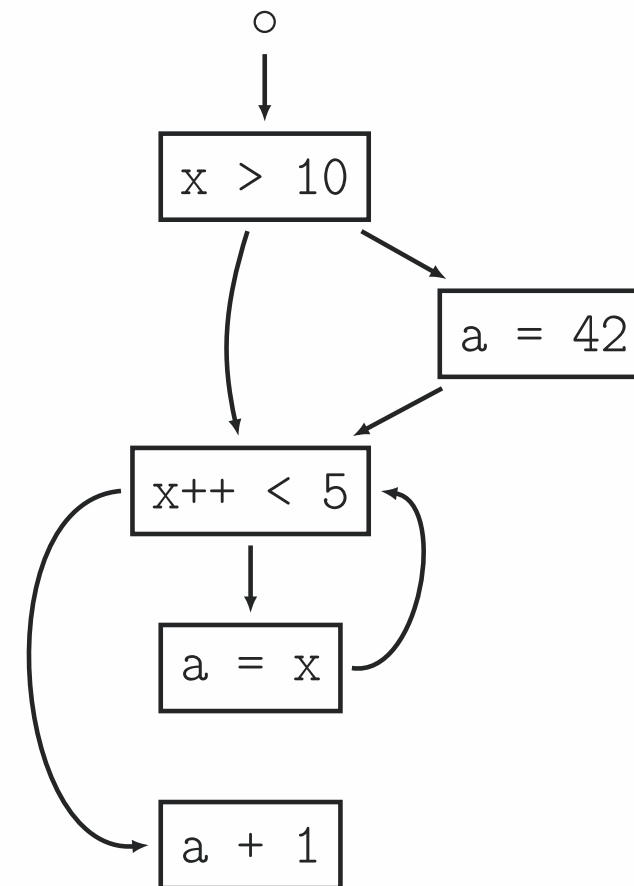
## Chapter #2: Method

[ [CFG](#) Properties Over-approximation Meet GEN/KILL Low ]

## Control Flow Graph

First, we represent the program as a Control Flow Graph (CFG):

```
int f(int x) {  
    int a;  
    if (x > 10)  
        a = 42;  
    while (x++ < 5)  
        a = x;  
    return a + 1;  
}
```



Example [Method](#) Sensitivities Types Literature

11/31

[ [CFG](#) Properties Over-approximation Meet GEN/KILL Low ]

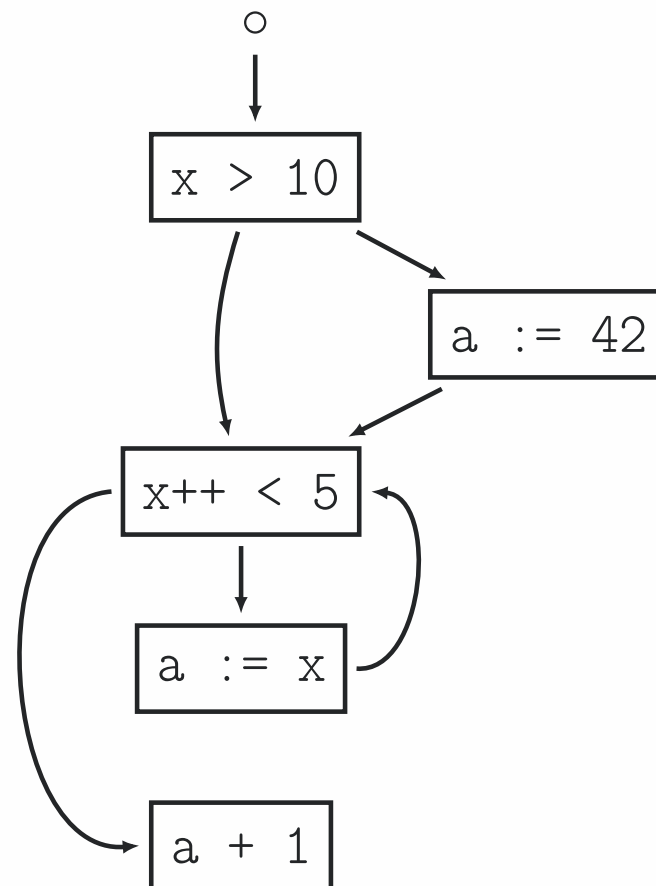
## Six Properties of Data Flow Analysis

Data flow analysis propagates information (data) along the control flow graph, with the following six properties in mind:

1. Domain (of data flow facts)
2. Direction (forward or backward)
3. Transfer Function (sometimes with GEN and KILL sets)
4. Confluence Operator ("meet" or "join")
5. Boundary Condition (start data for the entry node)
6. Initial Values (start data for each node)

[ CFG Properties [Over-approximation](#) Meet GEN/KILL Low ]

## Over-approximation

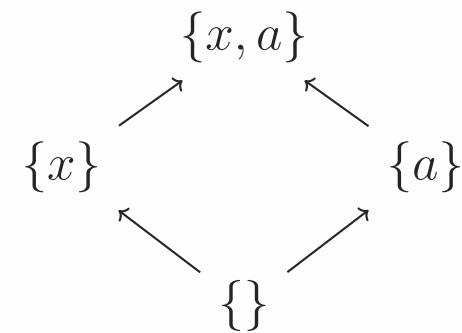
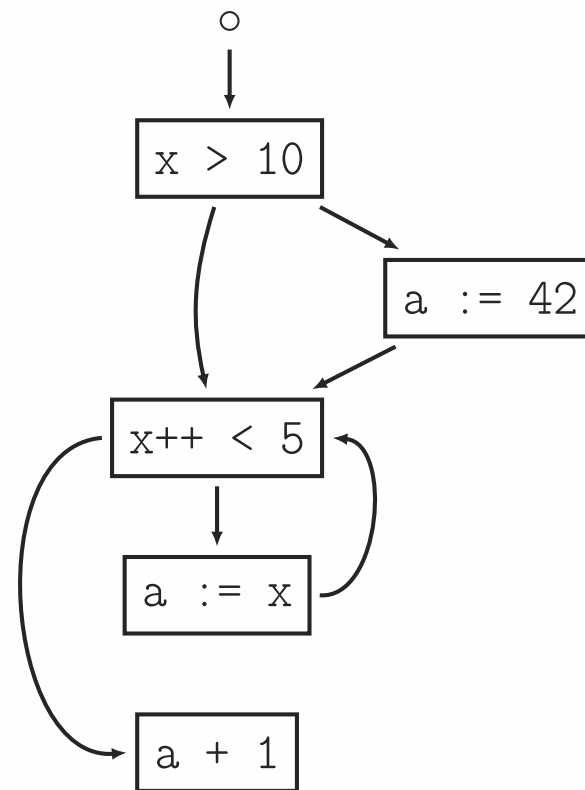


1. Domain:  
variable names
2. Direction:  
forward
3. Transfer Function:  
add on “:=”
4. Confluence Operator:  
meet, intersection
5. Boundary Condition:  
 $\{x\}$
6. Initial Values:  
empty sets

[ CFG Properties Over-approximation [Meet](#) GEN/KILL Low ]

## Meet Operator

The meet operator is coming from the lattice that abstracts the data that flows (remember abstract interpretation):



$$\{x, a\} \sqcap \{x\} \rightarrow \dots$$

$$\{a\} \sqcap \{\} \rightarrow \dots$$

$$\{a\} \sqcap \{x\} \rightarrow \dots$$

Example [Method](#) Sensitivities Types Literature

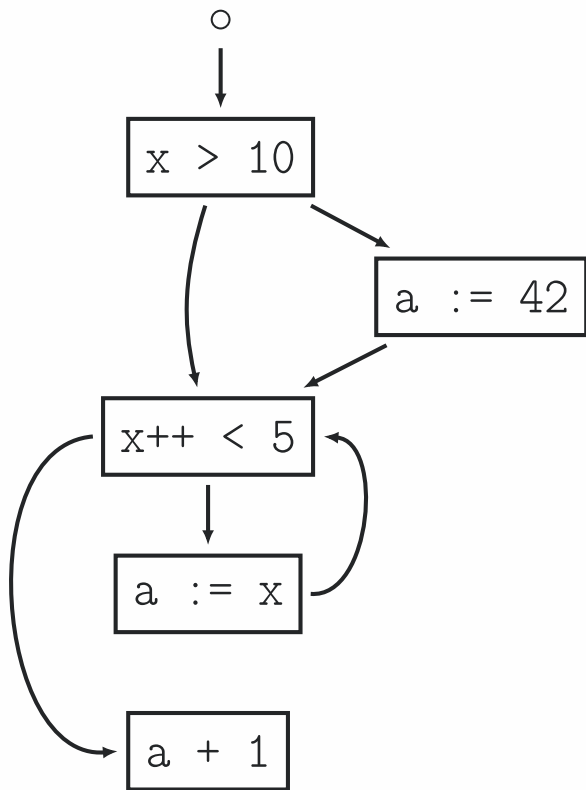
15/31

[ CFG Properties Over-approximation [Meet](#) GEN/KILL Low ]

[ CFG Properties Over-approximation Meet [GEN/KILL](#) Low ]

# GEN and KILL Functions

A transfer function may be defined by defining GEN and KILL functions:



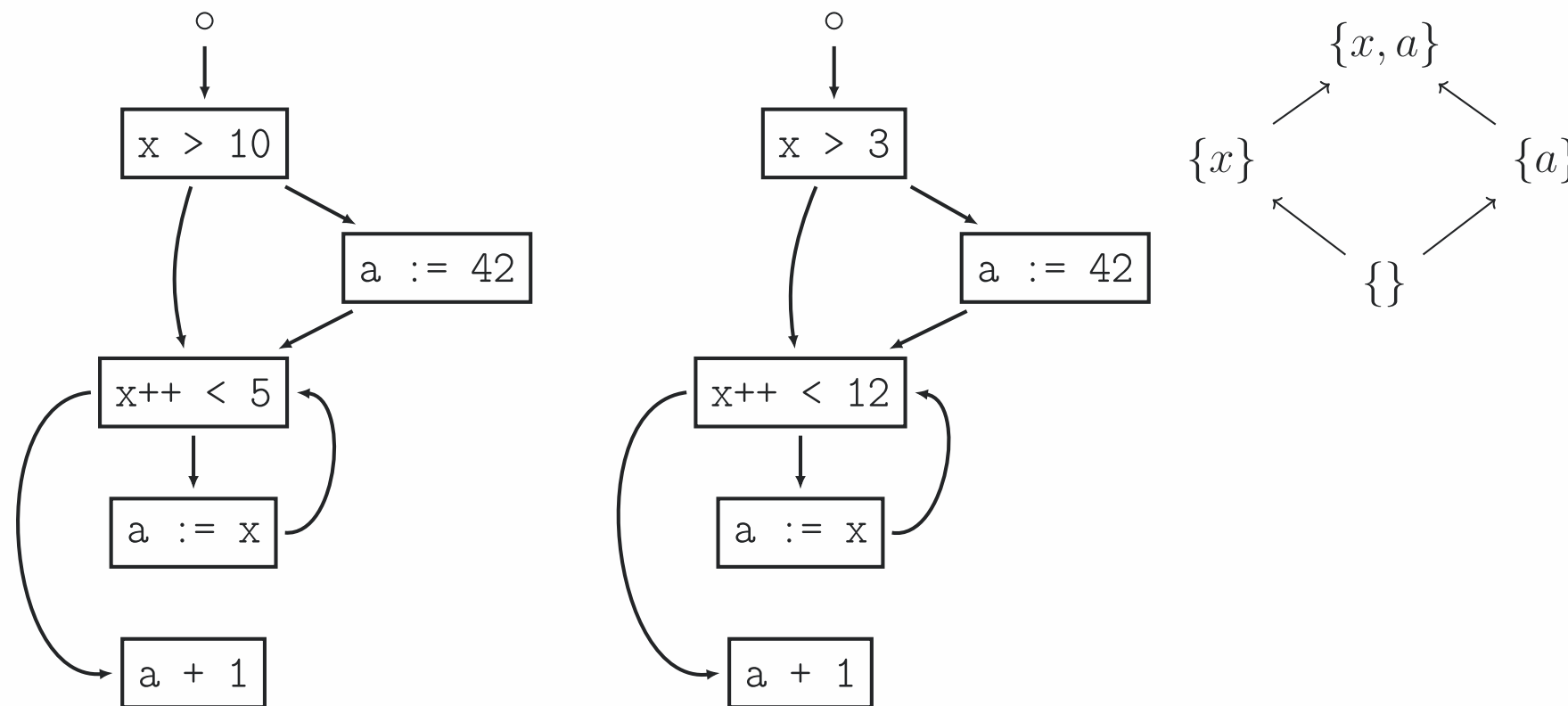
<i>s</i>	GEN( <i>s</i> )	KILL( <i>s</i> )
<i>x</i> > 10	{ }	{ }
<i>a</i> := 42	{ <i>a</i> }	{ }
<i>x</i> ++ < 5	{ }	{ }
<i>a</i> := <i>x</i>	{ <i>a</i> }	{ }
<i>a</i> + 1	{ }	{ }



[ CFG Properties Over-approximation Meet GEN/KILL [Low](#) ]

## Over-approximation = Low Precision

From the perspective of path insensitive data flow analysis, there are bugs in both CFGs, but it's wrong:



Example [Method](#) Sensitivities Types Literature

18/31

[ CFG Properties Over-approximation Meet GEN/KILL [Low](#) ]

Chapter #3:

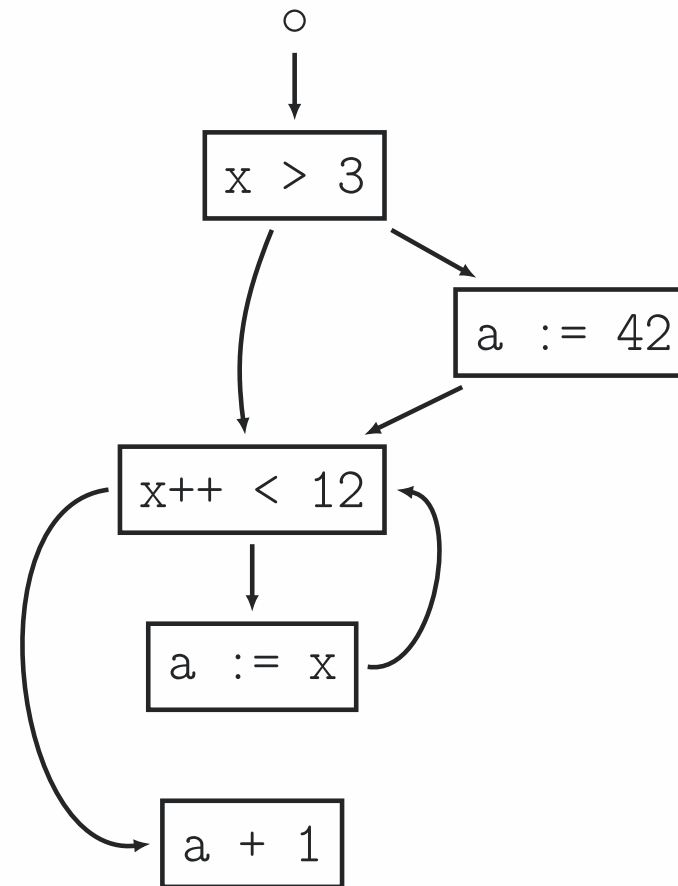
## Sensitivities

[ [Path-Sensitive Analysis](#) Flow-Sensitive Analysis Context-Sensitive Analysis ]

## Path-Sensitive Analysis

A path-sensitive analysis computes different pieces of analysis information dependent on the predicates at conditional branch instructions.

[ [Path-Sensitive Analysis](#) Flow-Sensitive Analysis Context-Sensitive Analysis ]



## Flow-Sensitive Analysis

A flow-sensitive analysis takes into account the order of statements in a program.

The analysis we did before was flow sensitive. Flow insensitive analysis example:

```
1 | a = 0;  
2 | a = 5;  
3 | a = a + 1;  
4 | // What is a possible value of 'a'?
```

## Context-Sensitive Analysis

A context-sensitive analysis is an interprocedural analysis that considers the calling context when analyzing the target of a function call.

```
1 | f(5, 6); // call-site #1
2 | f(6, 5); // call-site #2
3 | void f(x, y) {
4 |     // Is it possible to have x == y?
5 | }
```

Chapter #4:

## Most Common Types



[ [Reaching](#) Liveness Assignment Available Constants ]

## Reaching Definitions Analysis

Reaching definitions is a data-flow analysis which statically determines which definitions may reach a given point in the code.

```
1 float price(int book) {  
2     float p = load_from_database();  
3     if (book < 100)  
4         p = 14.99;  
5     if (book > 50)  
6         p = 9.99;  
7     float discount = 0.90;  
8     return p * discount;  
9 }
```

Do you see any problems with this code?

[ Reaching [Liveness](#) Assignment Available Constants ]

## Liveness Analysis

Live variable analysis calculates the variables that are live at each point in the program (they hold values that may be needed in the future).

```
1 int price(int book_id) {  
2     int p;  
3     int discount;  
4     if (book_id > 400)  
5         discount = 10;  
6     p = load_price_from_database(book_id);  
7     p = ( p * 95 ) / 100;  
8     return p;  
9 }
```

Do you see any problems in the code?

## Definite Assignment Analysis

Definite assignment analysis conservatively ensures that a variable or location is always assigned before it is used.

```
1 int salary(int user_id) {  
2     int s;  
3     if (user_id > 400) {  
4         s = get_salary_from_mysql(user_id);  
5     } else if (user_id < 400) {  
6         s = 0;  
7     }  
8     return s;  
9 }
```

Is there an error in this code?

## Available Expression Analysis

Available expression analysis determines for each point in the program the set of expressions that need not be recomputed.

```
1 int price(int book_id) {  
2     int p = 14;  
3     if (stock(book_id) < 100) {  
4         p = 19;  
5     } else if (stock(book_id) > 1000) {  
6         p = 9;  
7     }  
8     return p;  
9 }
```

Shall we compute `stock(book_id)` twice?

[ Reaching Liveness Assignment Available [Constants](#) ]

## Constant Propagation Analysis

Constant propagation analysis at every statement tells which variables is a constant: every execution that reaches that point, gives that variable the same value.

```
1 float discount(float price) {  
2     float d = 0.8;  
3     if (price < 14.99)  
4         d = 0.93;  
5     else  
6         d = d + 0.13;  
7     return price * d;  
8 }
```

Is there an error in this code?

Chapter #5:

**Further Reading/Watching**

Book and slides by Anders Møller et al.

Lectures of Michael Pradel on YouTube.

## References