Data Flow Analysis

Yegor Bugayenko

Lecture #7 out of 10
80 minutes

The slidedeck was presented by the author in this YouTube Video

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:

java Copy code

```java
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-a
nalyzer-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

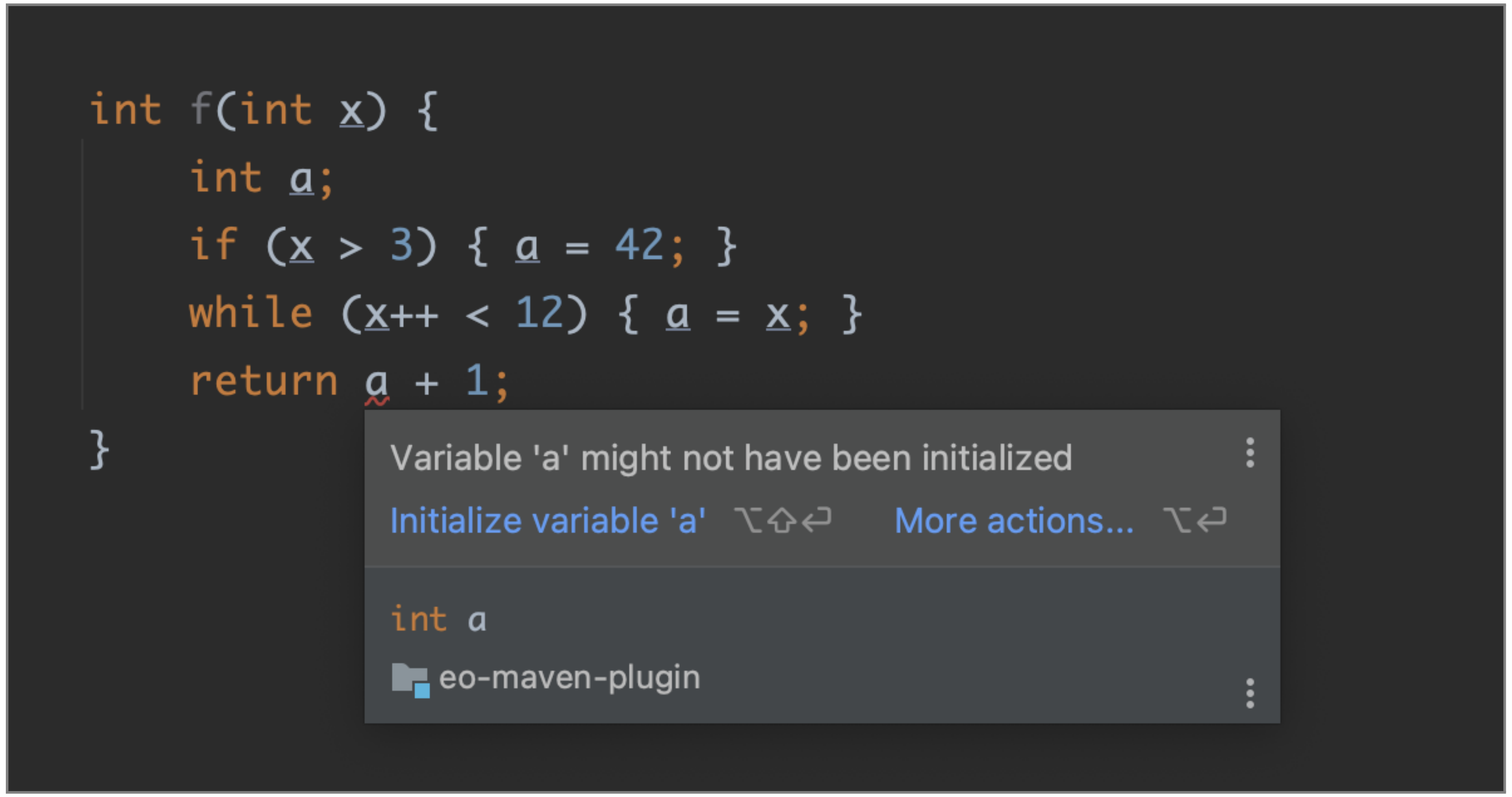
Clang UndefinedBehaviorSanitizer (the dynamic analyzer) can detect the issue in runtime:

```
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:



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
```

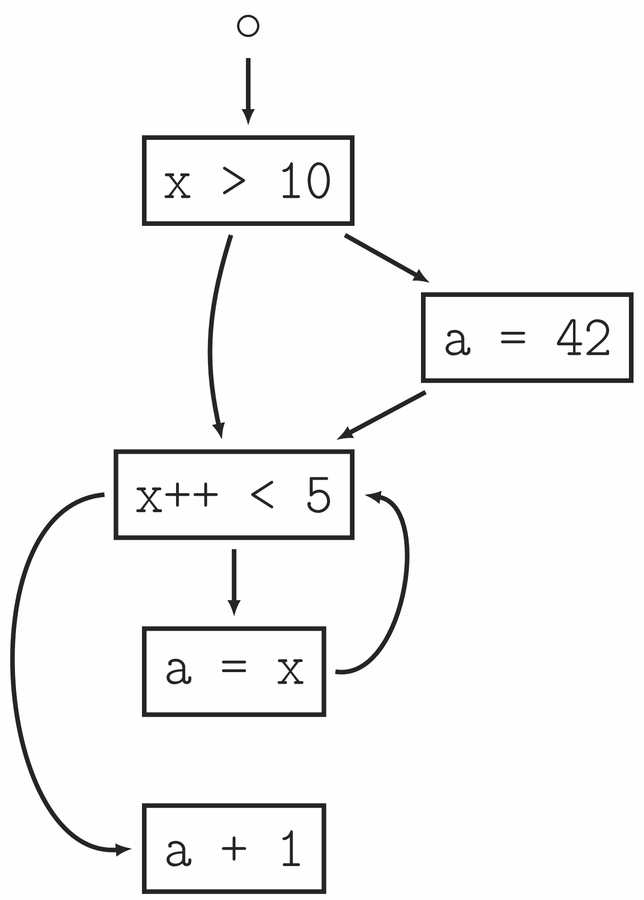
Chapter #2:

Method

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;
}
```

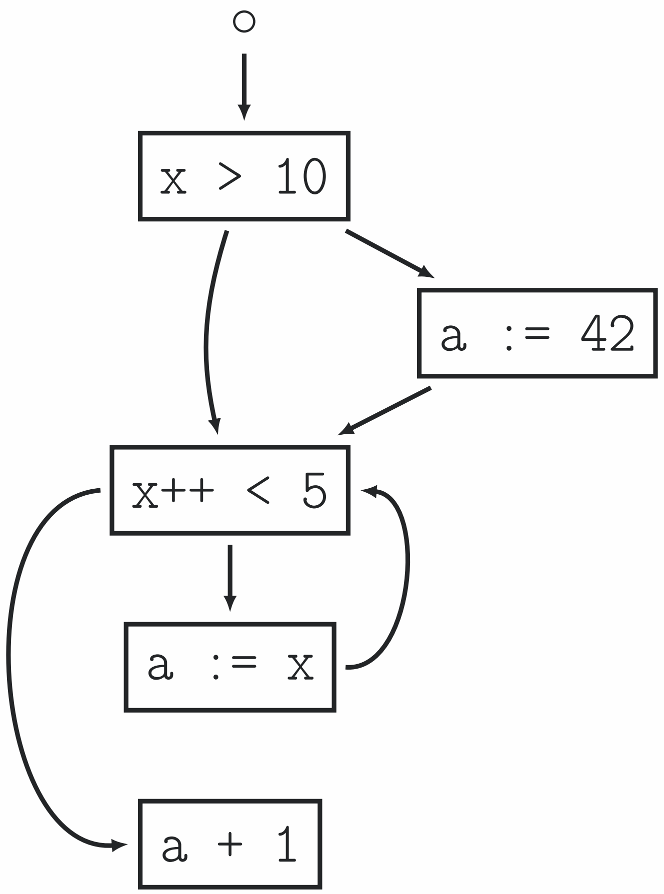


Six Properties of Data Flow Analysis

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)

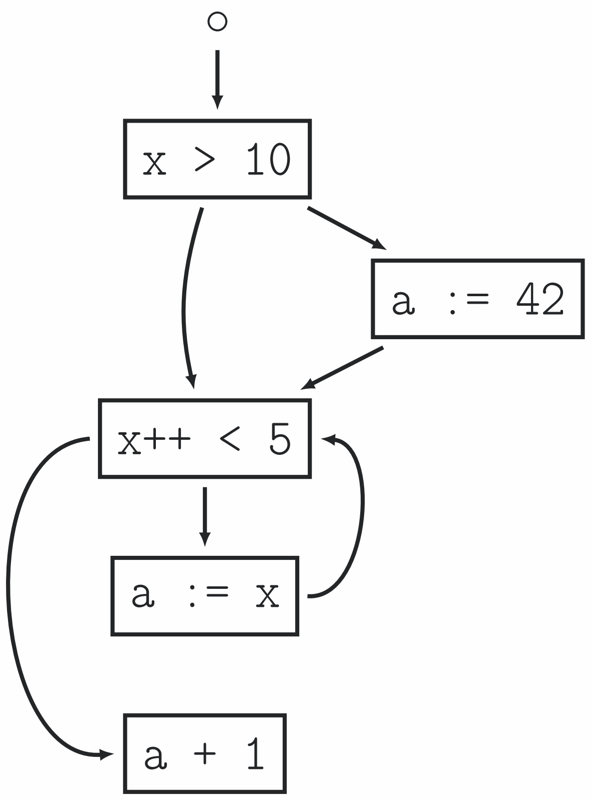
Over-approximation

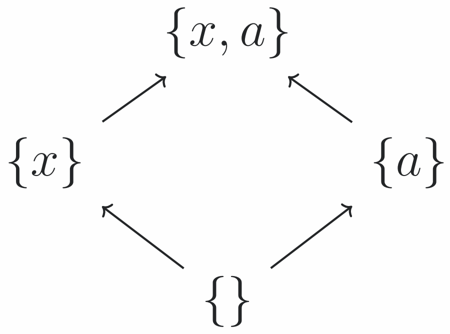


1. Domain:
   variable names
2. Direction:
   forward
3. Transfer Function:
   add on ":="
4. Confluence Operator:
   meet, intersection
5. Boundary Condition:
   $\{x\}$
6. Initial Values:
   empty sets

Meet Operator

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





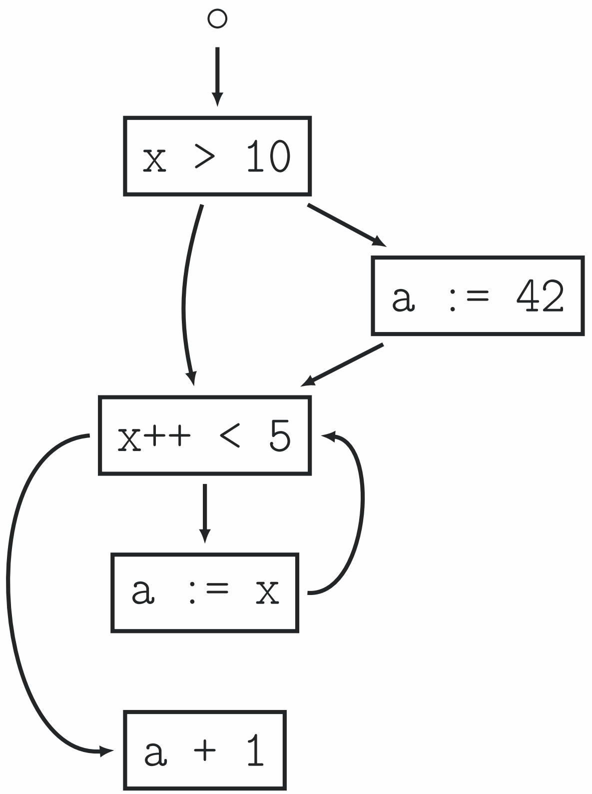
$$\{x, a\} \sqcap \{x\} \rightarrow \ldots$$

$$\{a\} \sqcap \{\} \rightarrow \ldots$$

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

GEN and KILL Functions

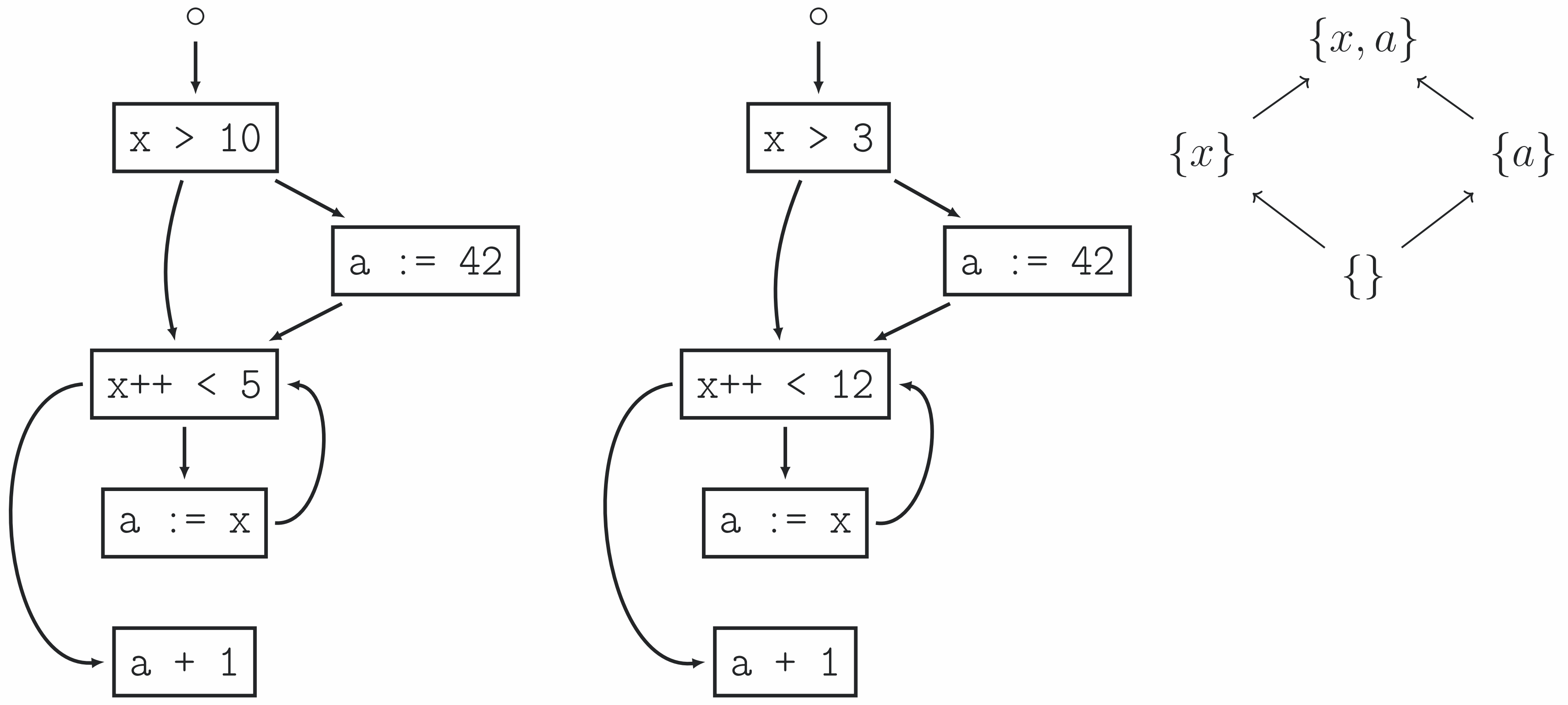
A transfer function may be defined by defining GEN and KILL functions:



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

Over-approximation = Low Precision

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

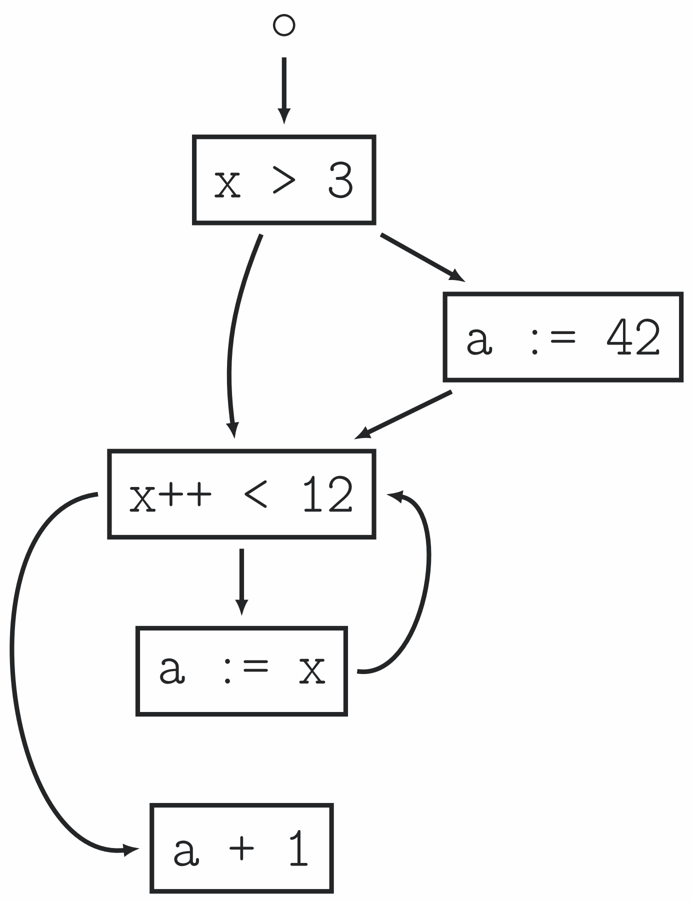


Chapter #3:

Sensitivities

Path-Sensitive Analysis

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


x > 3
a := 42
x++ < 12
a := x
a + 1

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:

```
a = 0;
a = 5;
a = a + 1;
// 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.

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

Chapter #4:

Most Common Types

Reaching Definitions Analysis

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

```
float price(int book) {
  float p = load_from_database();
  if (book < 100)
    p = 14.99;
  if (book > 50)
    p = 9.99;
  float discount = 0.90;
  return p * discount;
}
```

Do you see any problems with this code?

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).

```
int price(int book_id) {
  int p;
  int discount;
  if (book_id > 400)
    discount = 10;
  p = load_price_from_database(book_id);
  p = ( p * 95 ) / 100;
  return p;
}
```

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.

```
int salary(int user_id) {
  int s;
  if (user_id > 400) {
    s = get_salary_from_mysql(user_id);
  } else if (user_id < 400) {
    s = 0;
  }
  return s;
}
```

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.

```
int price(int book_id) {
  int p = 14;
  if (stock(book_id) < 100) {
    p = 19;
  } else if (stock(book_id) > 1000) {
    p = 9;
  }
  return p;
}
```

Shall we computer `stock(book_id)` twice?

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.

```
float discount(float price) {
  float d = 0.8;
  if (price < 14.99)
    d = 0.93;
  else
    d = d + 0.13;
  return price * d;
}
```

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