DEBUGGING: STATIC ANALYSIS

WS 2017/2018

Martina Seidl
Institute for Formal Models and Verification

JOHANNES KEPLER
UNIVERSITY LINZ

Deduction Techniques (1/2)

basic idea: reasoning from abstract program to concrete program runs (program is not executed)

```
10 x = read ();
...
20 y = 0;
...
30 x = y;
..
40 print ("x = " + x);
```

question: what is the value of variable x in line 40 and why?

Deduction Techniques (2/2)

approach:

- identification of statements that could have caused the failure
  $\Rightarrow$ focus on relevant statements
- identification of statements that could not have caused the failure
  $\Rightarrow$ ignore irrelevant statements

$\Rightarrow$ identification of possible origins of the failure

$\Rightarrow$ narrow down search space

$\Rightarrow$ more effective debugging

Interplay of Statements

- effects of statements: contribution to information flow
  - write: change a program state
  - control: determine next executed statement
- affected statements: involvement in information flow
  - read: continue with changed program state
  - execution: effect only manifests on execution

dependencies between statements:

- control dependence
- data dependence

Data Dependence / Control Dependence

data dependence: Statement B is data dependent on statement A if

- A write some variable x that is read by B
- there is at least one path in the control flow graph from A to B in which x is not overwritten by some other statement

control dependence: Statement B is control dependent on statement A if

- B's execution is potentially controlled by A

$\Rightarrow$ visualization in **program-dependence graph**

$\Rightarrow$ analysis which statements influence which statements

Control Flow Graph

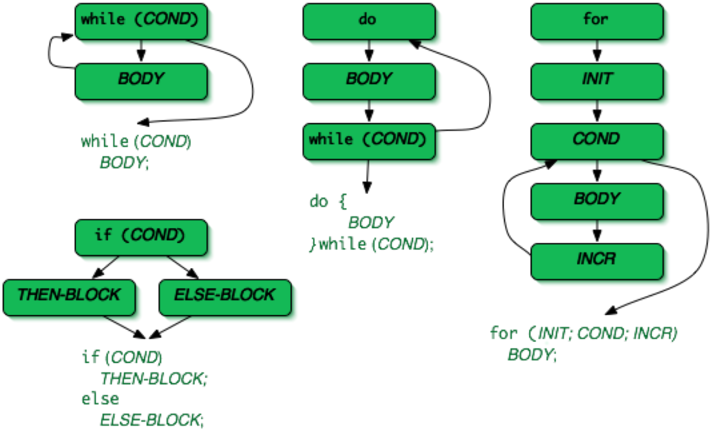
A control flow graph is a representation of all paths that might be traversed through a program during its execution.

elements of a control flow graph

- node: program statement
- edge: control flow
- special node: entry/exit node
- basic blocks: nodes that follow each other

Control Flow Patterns

patterns for control structures: composing structure of a program



Complications When Reasoning about Programs

- **jumps and gotos**
  unconditional transfer of control
- **indirect jumps**
  jump address is stored in a variable
- **dynamic dispatch**
  method overwriting in object-oriented languages
- **exceptions**
  transfer of control to calling function

Example: Fibonacci Numbers
Implementation with Defect

```
int fib (int n) {
  int f;
  int f0 = 1;
  int f1 = 1;

  while (n > 1) {
    n = n - 1;
    f = f0 + f1;
    f0 = f1;
    f1 = f;
  }
  return f;
}
```

```
int main () {
  int n = 9;

  while (n > 0) {
    printf("fib(%d)=%d\n",
            n, fib(n));
    n = n - 1;
  }
  return 0;
}
```

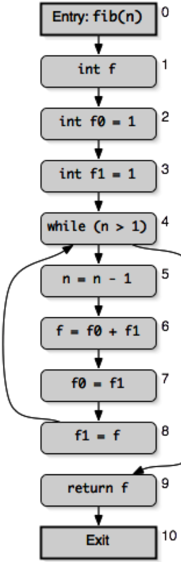
problem: `fib (1)`

Example: Effects

	Statement	Reads	Writes	Controls
0	`fib(n)`		`n`	1-10
1	`int f`		`f`	
2	`f0 = 1`		`f0`	
3	`f1 = 1`		`f1`	
4	`while (n > 1)`	`n`		5-8
5	`n = n - 1`	`n`	`n`	
6	`f = f0 + f1`	`f0, f1`	`f`	
7	`f0 = f1`	`f1`	`f0`	
8	`f1 = f`	`f`	`f1`	
9	`return f`	`f`	`<ret>`	

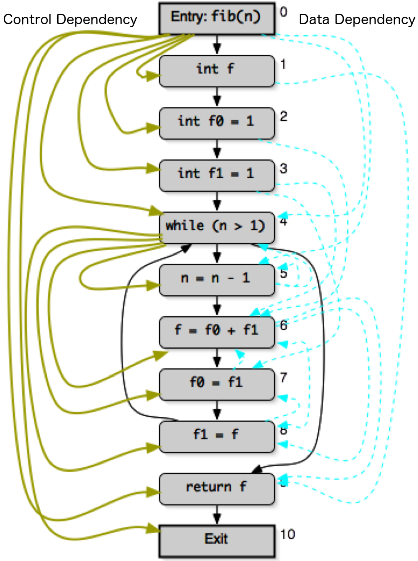
Example: Control Flow Graph

```
0 int fib (int n) {
1   int f;
2   int f0 = 1;
3   int f1 = 1;

4   while (n > 1) {
5     n = n - 1;
6     f = f0 + f1;
7     f0 = f1;
8     f1 = f;
    }
9   return f;
  }
```



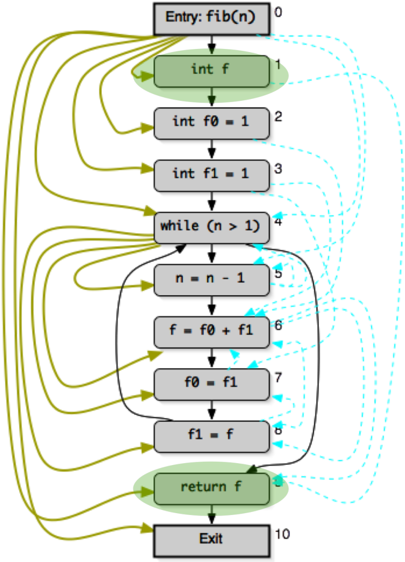
Example: Control Flow Graph

```
0 int fib (int n) {
1   int f;
2   int f0 = 1;
3   int f1 = 1;

4   while (n > 1) {
5     n = n - 1;
6     f = f0 + f1;
7     f0 = f1;
8     f1 = f;
    }
9   return f;
  }
```



Example: Control Flow Graph

```
0 int fib (int n) {
1   int f;
2   int f0 = 1;
3   int f1 = 1;

4   while (n > 1) {
5     n = n - 1;
6     f = f0 + f1;
7     f0 = f1;
8     f1 = f;
    }
9   return f;
  }
```



Program Slicing

problem: program computes wrong value for variable z at line 1024, but the statement at line 1024 is correct. Why?

$\Rightarrow$ automaticly find defect with program slicing

> A **program slice** is a reduced program that preserves the original program's behavior for a given set of variables at a chosen point in a program.

basic idea:

- focus on relevant statements and filter irrelevant ones
- narrow down infection sites

Static Slicing

example:

original program

```
1 x = 2;
2 y = x + 2;
3 z = x + 1 ;
4
```

slice w.r.t. (4, {z})

```
1 x = 2;
2
3 z = x + 1;
4
```

what happened?

- deletion of statements
- projection of program semantics was preserved

Static Slicing: (Informal) Definition

A **static slicing criterion** of a program P is a pair (s, V), where s is a statement in P and V is a subset of the variables in P.

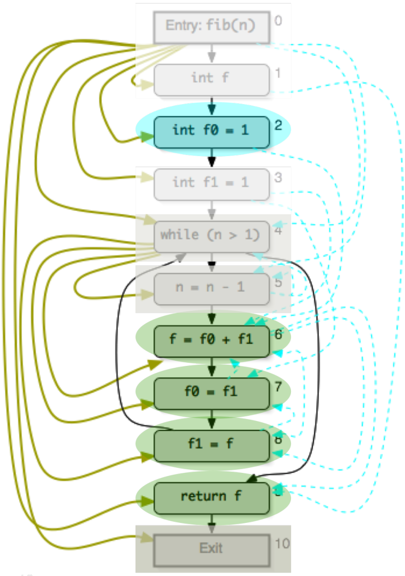
A **slice** S of a program P on a slicing criterion (s, V) is a program such that

- S is obtained by deleting statements from P
- P's behavior on variables V is preserved in s

note: no algorithm to find state-minimal slices
(finding minimal slices is equivalent to solve the Halting problem!)

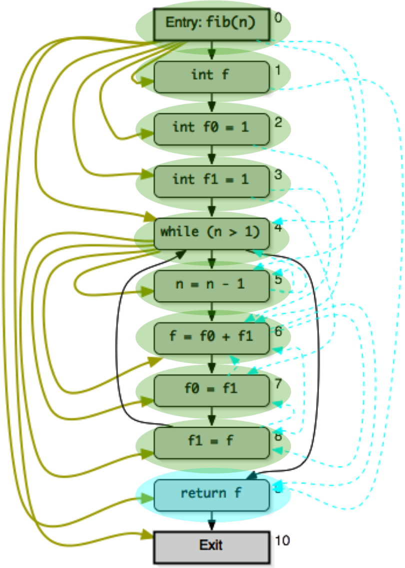
Forward Slicing

- given a statement A, the forward slice contains all statements whose read variables or execution could be influenced by A
- $S^F(A) = \{B \mid A \xrightarrow{+} B\}$
- not included statements cannot be affected by A



Backward Slicing

- Given a statement B, the backward slice contains all statements that could influence the read variables or execution of B
- $S^B(B) = \{B \mid A \xrightarrow{+} B\}$
- often all statements between A and B are included



Multiple Slices

- example: two slices (addition, multiplication)
- backward slice of addition
- backward slice of multiplication
- backward slice of addition and multiplication

```
int main () {
   int a, b, sum, mul;
   sum = 0;
   mul = 1;
   a = read ();
   b = read ();
   while (a <= b) {
      sum = sum + a;
      mul = mul * a;
      a = a + 1;
   }
   write (sum);
   write (mul);
}
```

Backbone

- statements that occur in both slices
- useful for focusing on common behavior

```
a = read ();
b = read ();
while (a <= b) {
```

```
a = a + 1;
```

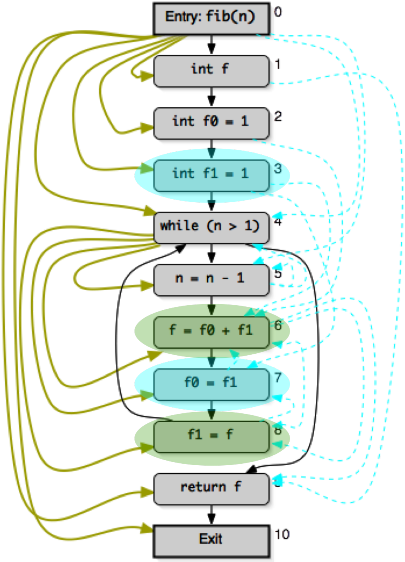
Dice

- only the difference between two slices
- useful for focusing on differing behavior

```
    sum = 0;

        sum = sum + a;

    write (sum);
```

Chop

- intersection between a forward and a backward slice
- useful for determining how statement A (originating the forward slice) influences statement B (originating the backward slice)



Code Smells

A code smell is a surface indication that usually corresponds to a deeper problem in the system (M. Fowler)

examples:

- use of uninitialized variables
- unused values
- unreachable code
- memory leaks
- interface misuse
- null pointers

Example: Uninitialized Variables

example 1:

```
$ gcc -Wall -O -o fibo fibo.c
fibo.c: In function ‘fib’:
fibo.c:7: warning: ‘f’ might be used uninitialized in this function
```

example 2 (false positive):

```
int go;
switch (color) {
    case RED:
    case AMBER:
        go = 0; break;
    case GREEN:
        go = 1; break;
}
if (go) { ... }
```

Unused Variable / Unreachable Code

unused variable: variable that is never read

in the dependency graph, no other statement is data dependent on the write of such a variable

unreachable code: code that is never executed

example:

```
if (w >= 0)
    printf ("w is non-negative\n");
else if (w > 0)
    printf ("w is positive\n");
```

Memory Leaks / Null Pointers

```
1  int *readbuf (int size) {
2    int *p = malloc (size * sizeof(int));
3    for (int i = 0; i < size; i++) {
4     p[i] = readint ();
5     if (p[i] == 0)
6         return 0;  // end-of-file
7    }
8    return p;
9 }
```

problems:

- line 2: return value of `malloc` is NULL
  $\Rightarrow$ no memory is allocated
- lines 5 and 6: function is left without reference to `p`
  $\Rightarrow$ `p` cannot be released

Interface Missuse

- memory is not the only resource that must be explicitly deallocated when no longer in use, e.g., streams, sockets, locks, devices, ...
- indication in control flow graph: path from stream opening to statement where stream reference is lost without closing stream

example:

```
void readfile() {
    int fp = open(file);
int size = readint(file);
if (size <= 0)
    return;
...
close(fp);
}
```

Defect Patterns

- class implements `Cloneable` but does not define or use clone method
- method might ignore exception
- null pointer dereference in method
- class defines `equal()`; should it be `equals()`?
- method may fail to close database resource
- method may fail to close stream
- method ignores return value
- unread field
- unused field
- unwritten field

Limits of Static Analysis

- many false positives
- many questions are undecidable (Halting problem)
- many imprecisions
  - indirect access, e.g., `a [i]` depends on `i`
  - pointers
  - functions
  - object orientation, concurrency
- further risks
  - code mismatch
  - abstracting away the execution environment
  - imprecision