Static Analysis
and Program Comprehension

Nuno Oliveira
Pedro Rangel Henriques

Agenda

- Control Flow Graphs
- Data Dependency Graphs
- System Dependency Graphs

- Abstract Interpretation

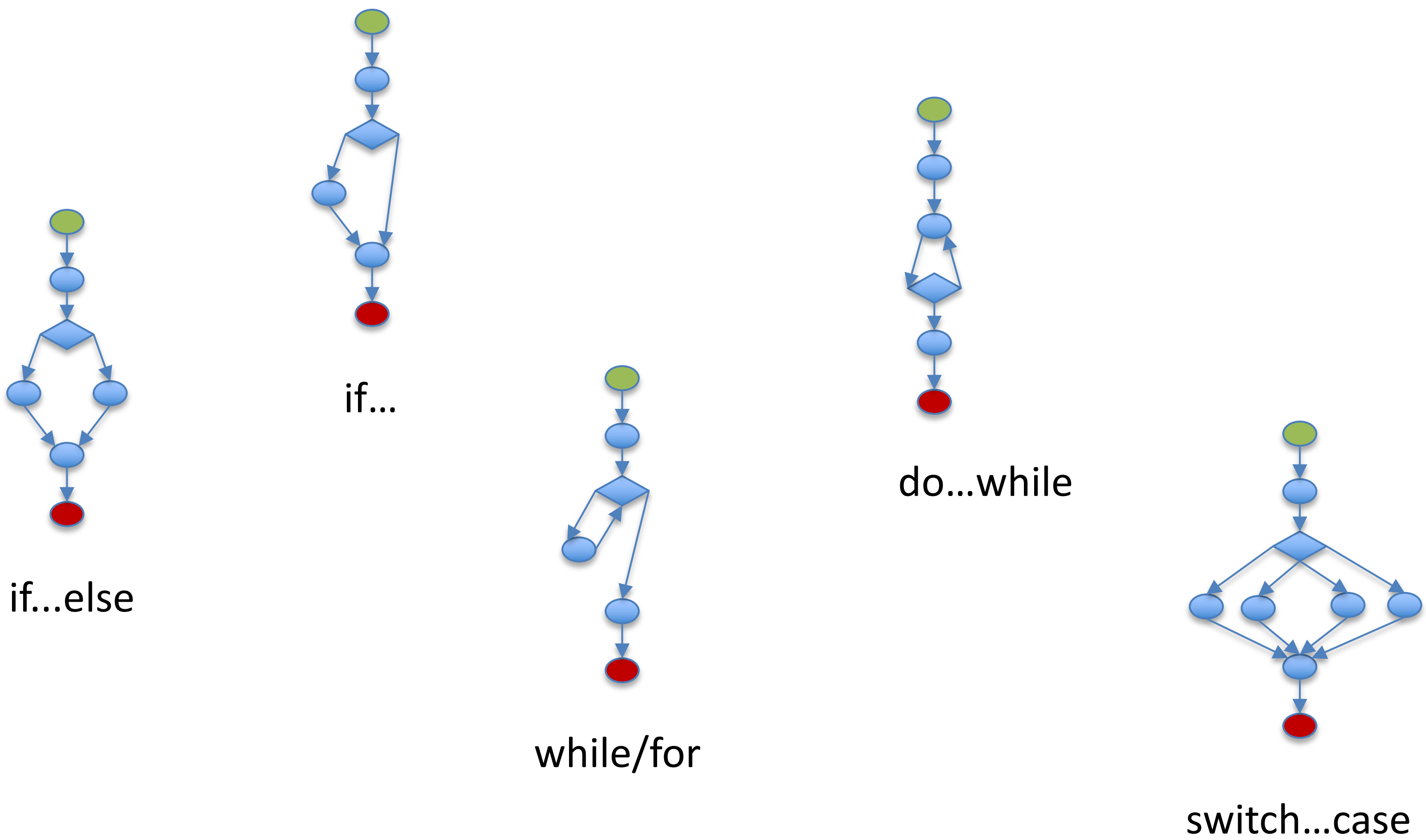
Control Flow Graphs

- A graph model of a function that represents control dependencies
  - Model all the possible paths that the execution of a function may take.

- CFG=(V,E)
  - V (vertices) store the basic blocks of functions
    - Simplification: each vertex represents a statement.
  - E (edges) define the connection between the vertices, hence the flow paths.

- Control statements have well-defined subgraph structures

Control Flow Graphs



Control Flow Graphs

- Allow the recognition of all executable paths
  - Determine the complexity of a function
    - McCabe's complexity = E – V + 2
  - Determine the minimum amount of unit tests for testing the code
- Enable early pinpoint of dead code
  - island graphs
- General understanding of the code by their flow of execution

Data Dependency Graphs

- A graph model of a function that represents the dependencies between data
- DFG=(V,E)
  - V (vertices) store data items (variables, constants, operators, etc)
  - E (edges) define the dependency between the data items
- Based on **Single Assignment** form of the code

Data Dependency Graphs

```
Original Code:

x = 5;
x = x - 3;
if(x<3) {
   y = x + 2;
   w = y;
} else {
   y = x – 3;
}
w = x - y;
z = x + y;
```

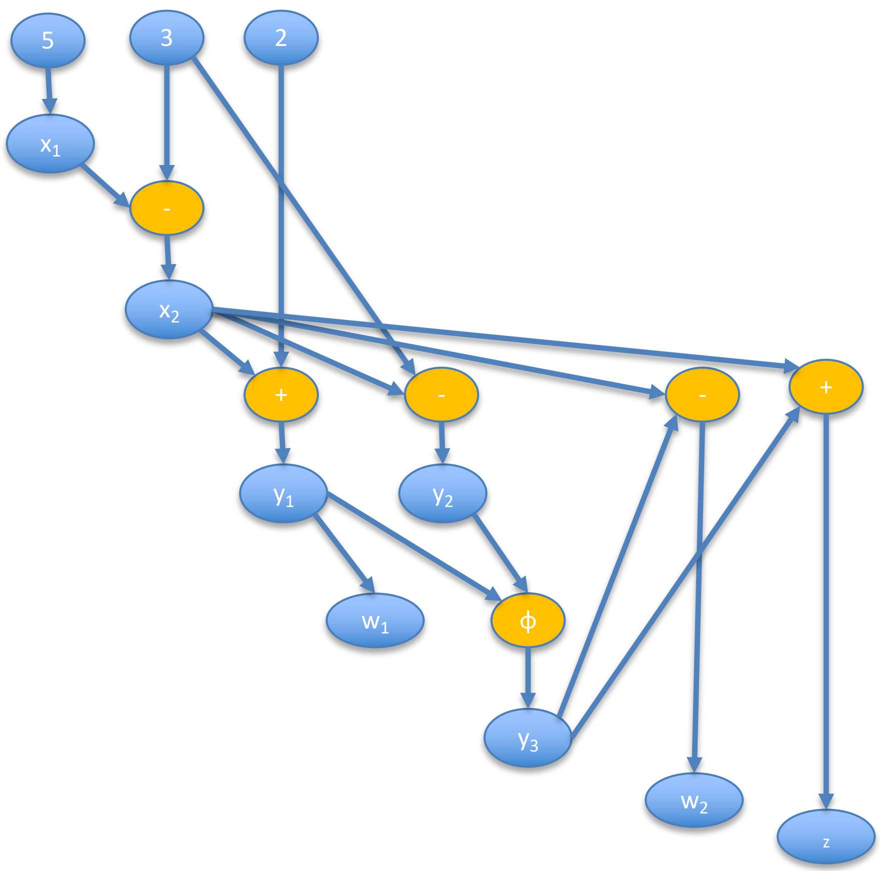
```
Single Assignment form:

x_1 = 5;
x_2 = x_1 - 3;
if ( x_2 < 3 ) {
   y_1 = x_2 + 2;
   w_1 = y_1;
} else {
   y_2 = x_2 – 3;
}
y_3 = φ(y_1,y_2);
w_2 = x_2 – y_3;
z = x_2 + y_3;
```

Data Dependency Graphs

Single Assignment form:

```
x₁ = 5;
x₂ = x₁ - 3;
if ( x₂ < 3 ) {
   y₁ = x₂ + 2;
   w₁ = y₁;
} else {
   y₂ = x₂ - 3;
}
y₃ = ϕ(y₁,y₂);
w₂ = x₂ - y₃;
z = x₂ + y₃;
```



Data Dependency Graphs

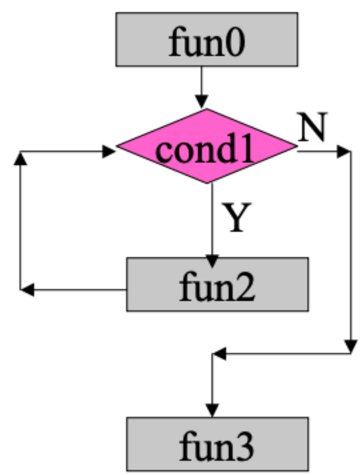
- Allow general identification of dependencies in the code
  - Variables usage and liveness
- Enable pinpointing of dead code
  - island subgraphs

- Important for compiler construction:
  - Memory/registry allocation and management
  - Scheduling of execution

Data Dependency Graphs

- Can be combined with CFG
  - Control/Data Flow Graphs (Program Dependency Graphs)
    - Clearly define the flow of data through the control struct

```
fun0();
while(cond1) {
  fun1();
}
fun2();
```



System Dependency Graphs

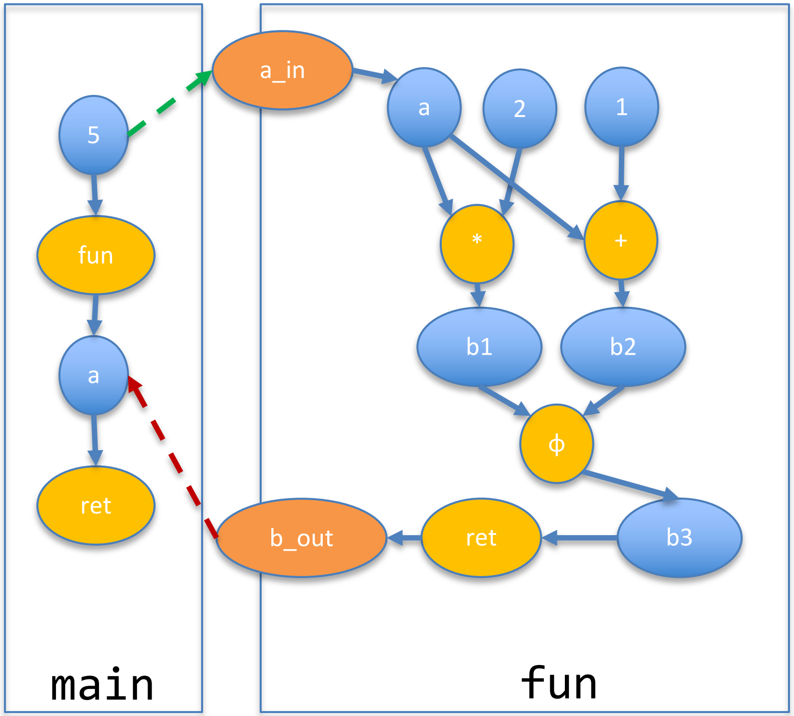
- Graph model of a system (set of functions) that represents both the flow and the dependencies between data and functions

- SDG = (PV,V,E,IPE)
  - PV (procedure vertices) represent the procedures/functions
  - V (vertices) represent the data items of each procedure
  - E (edges) represent the dependencies between the data items
  - IPE (inter-procedural edges) represent the passage of data to function vertices

System Dependency Graphs

```
main() {
 a = fun(5);
 return a;
}

fun(a) {
 b₁ = a * 2;
 if(b < 10) {
  b₂ = a + 1;
 }
 b₃ = ϕ(b₁,b₂);
 return b₃;
}
```



Notice that control flow is being skipped, but it may be considered as well!

System Dependency Graphs

- Provide a general overview of the system
  - Control flows
  - Data dependencies

- One graph allowing for the full analysis of a system
  - Security analysis (via taint analysis)
  - Pattern search (best coding practices/ linters)
  - ...

Abstract Interpretation (Basics)

- Is a formal method for the abstract interpretation of a piece of code.
  - Unlike concrete interpretation it assumes an abstract domain to approximate the set of concrete values in a program

- Works upon the CFG, and requires knowledge-base about the program variables
  - Such knowledge-base is called the environment, where all the abstract values of the variables are stored

Abstract Interpretation (Basics)

- A variable has, therefore an abstract value, represented as a set of possible values.
- Operation on the variables are actually abstract operations on the sets.
- For instance:
  - int x; -- will take values in the set of integers represented as intervals
    - Operation on x will change its value with usual numeric interval operations
  - String y; -- will take values in the set of char*

Abstract Interpretation (Basics)

`main() {`	
`a = fun(5);`	Method fun is invoked with concrete value 5. Abstract representation is [5,5].
`return a;`	
`}`	
`fun(a) {`	Parameter a takes the argument [5,5]
b_1 `= a * 2;`	[5,5] * [2,2] = [10,10]
`if(b < 10) {`	The test is done by checking whether b=[10,10] is contained in [-∞,9], which fails!
b_2 `= a + 1;`	Then this block is Dead Code – in the context of this system.
`}`	
$b_3 = \phi(b_1,b_2)$;	b_3 is, therefore [10,10],
`return` b_3;	which is returned to main and assigned to variable a.
`}`	
	Then the abstract value of this system is [10,10], which is a single value set {10}.

Abstract Interpretation (Basics)

- Is a complex subject, mainly when it comes to loops
  - It is necessary to find a fixed points but there's always the termination problem of a loop

Exercises

- Exercises:
  - Convert a TinyC program into its single assignment form, using visitors.
  - Define an abstract interpretation machine based on the TinyC grammar, using visitors and all the resources you think are reasonable to obtain the abstract value of a program (assuming it starts in method **main**).