University of BRISTOL

Program Analysis for Security

Sanjay Rawat

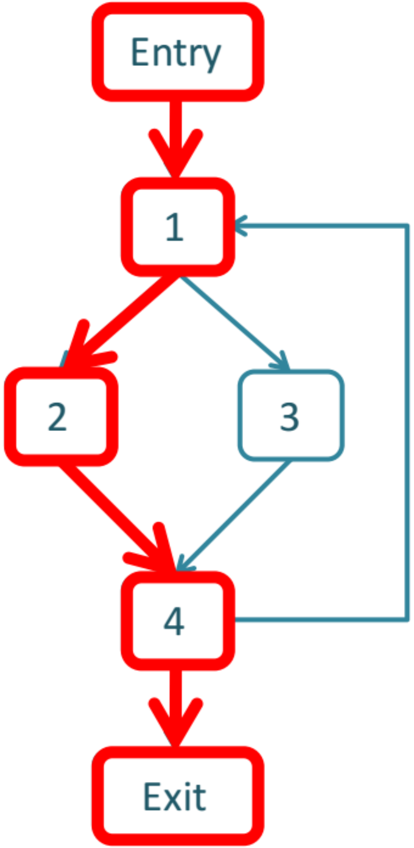
Adopted from courses from L. Mounier, Y. Lakhnech (& S. Rawat) of Verimag & J. Michell of Stanford.

Lecture Agenda

- Motivation for Program analysis
- Types of program analysis
  - Static
  - Dynamic
- A quick tour for static analysis techniques

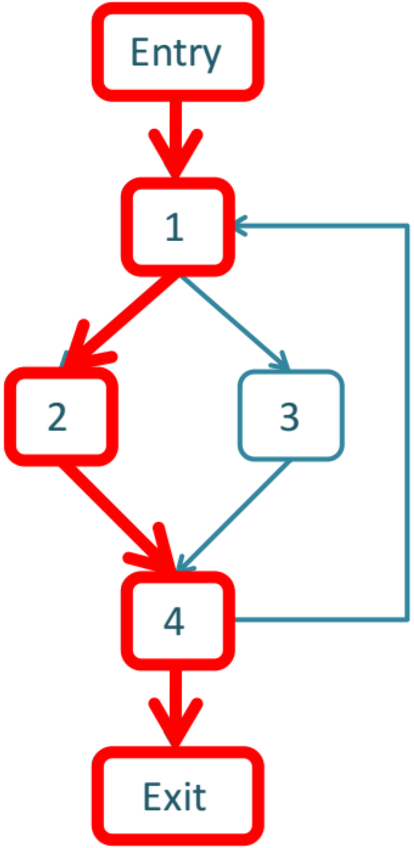
Motivation

- Why do we need *secure software development lifecycle*-- at large-scale, it is not easy to find bugs manually which are more accessible now.
  - Heartbleed, WhatsApp double-free, Facebook account deletion bug etc.
- How can you tell whether software you Develop or buy is safe to install and run?
  - Analyzing it? But...



• • •

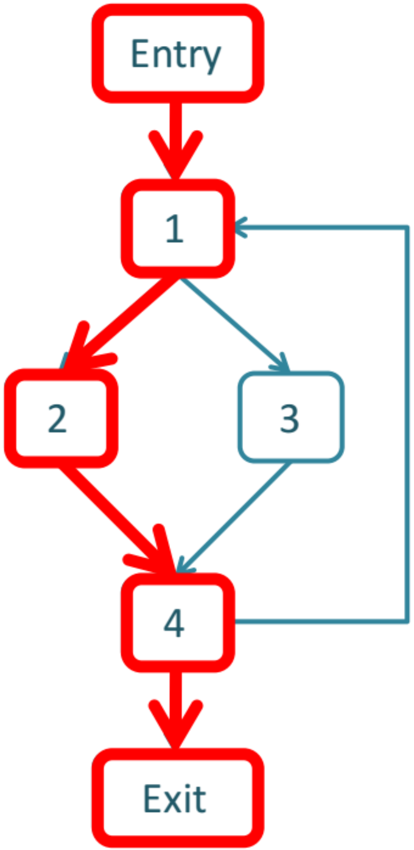
Software



Software

• • •

Behaviors

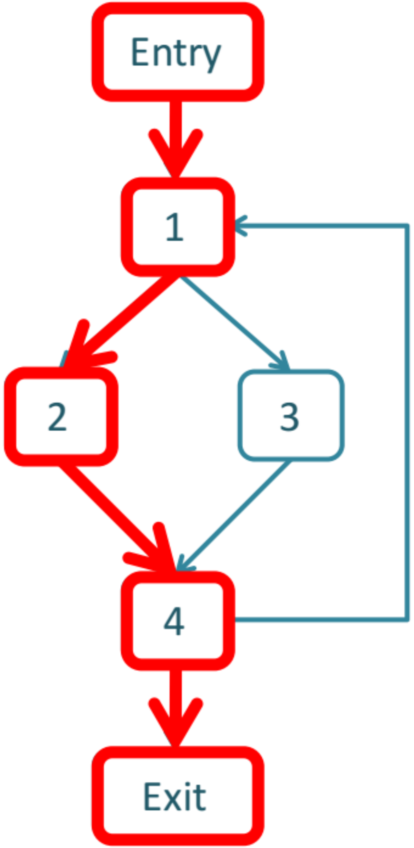


Software

1 → 2 → 4

• • •

Behaviors



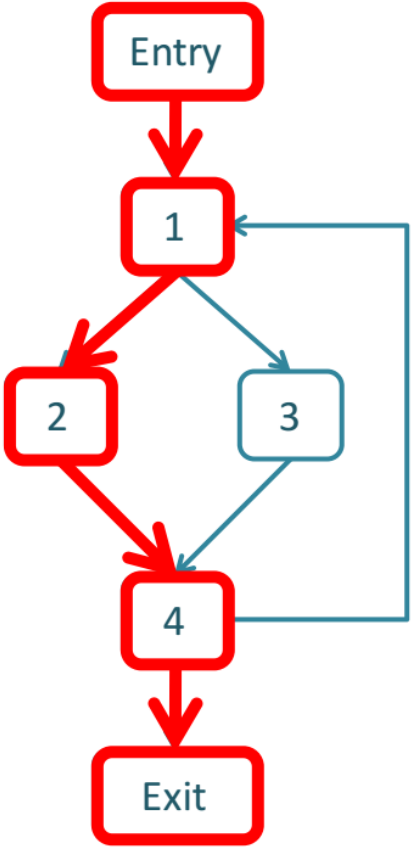
Software

1 → 2 → 4

1 → 3 → 4

• • •

Behaviors



Software

1 → 2 → 4

1 → 3 → 4

1 → 2 → 4 → 1 → 2 → 4

1 → 2 → 4 → 1 → 3 → 4

1 → 2 → 3 → 1 → 2 → 4 → 1 → 3 → 4

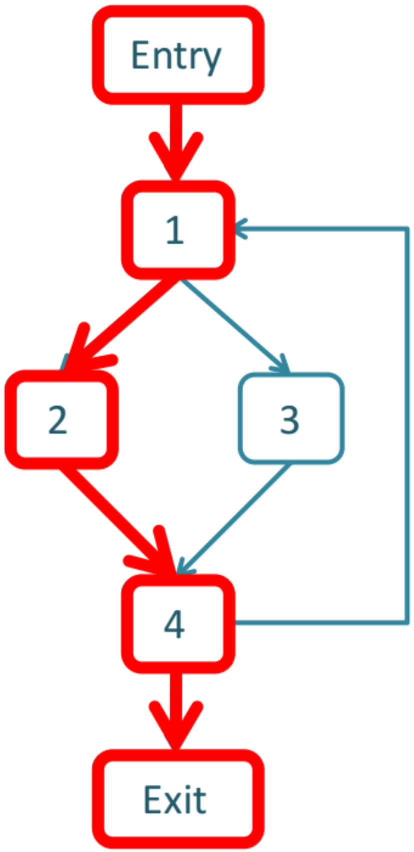
1 → 2 → 4 → 1 → 2 → 3 → 1 → 3 → 4

1 → 2 → 3 → 1 → 2 → 3 → 1 → 3 → 4

1 → 2 → 4 → 1 → 2 → 4 → 1 → 3 → 4

• • •

Behaviors



Software

1 → 2 → 4

1 → 3 → 4

1 → 2 → 4 → 1 → 2 → 4

1 → 2 → 4 → 1 → 3 → 4

1 → 2 → 3 → 1 → 2 → 4 → 1 → 3 → 4

1 → 2 → 4 → 1 → 2 → 3 → 1 → 3 → 4

1 → 2 → 3 → 1 → 2 → 3 → 1 → 3 → 4

1 → 2 → 4 → 1 → 2 → 4 → 1 → 3 → 4

• • •

Manual testing only examines small subset of behaviors

Behaviors

Real-world example….

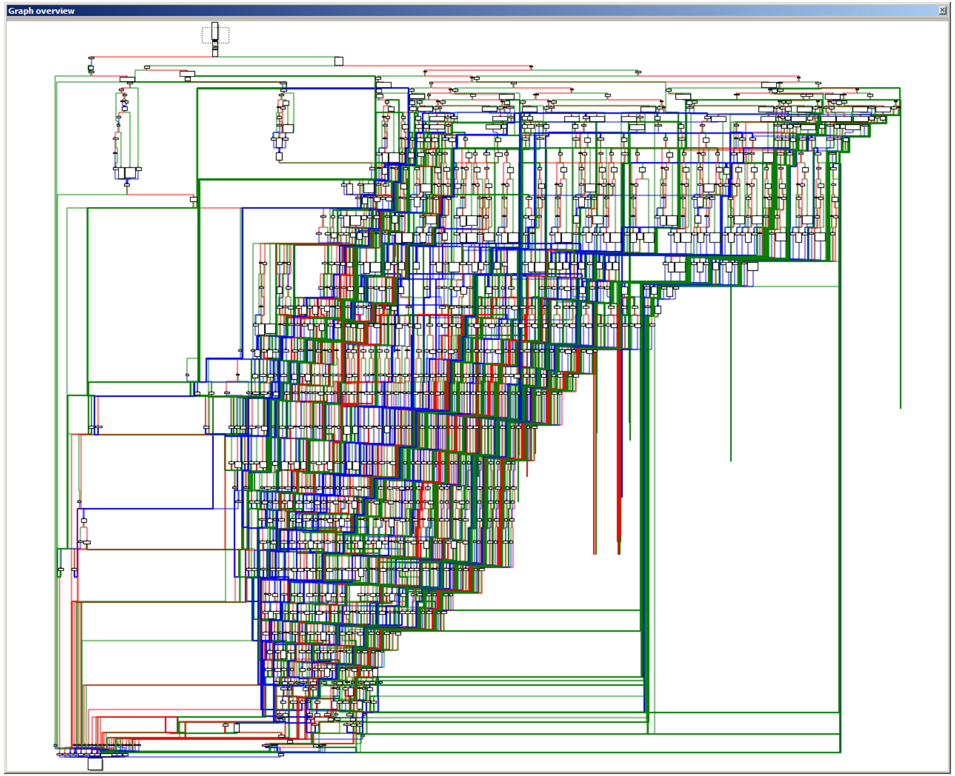


Image: stackoverflow

Definition

- Program analysis is the process of automatically analysing the behaviour of computer programs regarding a property such as correctness, reliability, safety and security.
- Traditionally, it focuses on program optimization (as in compilers) and program correctness (as in verification). However, the same techniques are applied to other domains. So, we will study the code techniques first, irrespective of domain of application.
- Types:
  - Static analysis: performed without executing the program.
  - Dynamic Analysis: performed at runtime.
  - Hybrid: a mix of the above.

Static Program Analysis

Static Program Analysis

- Analysing the code (source or assembly) of the program without executing it.

Static Program Analysis

- Analysing the code (source or assembly) of the program without executing it.
- Scalability and precision are big challenges– for binary code, it is even more challenging (why?)

Static Program Analysis

- Analysing the code (source or assembly) of the program without executing it.
- Scalability and precision are big challenges– for binary code, it is even more challenging (why?)
- Compilers make heavy use of such analyses.

Static Program Analysis

- Analysing the code (source or assembly) of the program without executing it.
- Scalability and precision are big challenges– for binary code, it is even more challenging (why?)
- Compilers make heavy use of such analyses.
- ***Given the availability of the code, it can analyse every component and path of the application.***

Static Program Analysis

- Analysing the code (source or assembly) of the program without executing it.
- Scalability and precision are big challenges– for binary code, it is even more challenging (why?)
- Compilers make heavy use of such analyses.
- ***Given the availability of the code, it can analyse every component and path of the application.***
- Tools:
  - **LLVM** provides a very robust platform to perform several static analyses (on source code).
  - For binary code, there exist several tools– IDA, Ghidra, Miasm, angr etc.
- Typical examples–
  - data-flow analysis, abstract interpretation, type system, model checking.
- For this course, we will be using **Ghidra** for learning some binary code analysis!

Dynamic Program Analysis

Dynamic Program Analysis

- Analyzing the program at runtime (Yes, just like a debugger!).

Dynamic Program Analysis

- Analyzing the program at runtime (Yes, just like a debugger!).
- Very precise, with reasonable scalability.

Dynamic Program Analysis

- Analyzing the program at runtime (Yes, just like a debugger!).
- Very precise, with reasonable scalability.
- Analysis is limited to the executed code of the program. Thus coverage is a problem.

Dynamic Program Analysis

- Analyzing the program at runtime (Yes, just like a debugger!).
- Very precise, with reasonable scalability.
- Analysis is limited to the executed code of the program. Thus coverage is a problem.
- Have been very useful for profiling (e.g. linux perftool).

Dynamic Program Analysis

- Analyzing the program at runtime (Yes, just like a debugger!).
- Very precise, with reasonable scalability.
- Analysis is limited to the executed code of the program. Thus coverage is a problem.
- Have been very useful for profiling (e.g. linux perftool).
- Often used with static/dynamic instrumentation (We will talk about dynamic binary Instrumentation).

Dynamic Program Analysis

- Analyzing the program at runtime (Yes, just like a debugger!).
- Very precise, with reasonable scalability.
- Analysis is limited to the executed code of the program. Thus coverage is a problem.
- Have been very useful for profiling (e.g. linux perftool).
- Often used with static/dynamic instrumentation (We will talk about dynamic binary Instrumentation).

  Existing tools– Intel Pin, Dyninst, Valgrind etc.

  From security standpoint-- Fuzzing (we will talk about it.)

  For this course, we will be using Intel Pintool.

Soundness, Completeness

Property	Definition
Soundness	"Sound for reporting correctness" Analysis says no bugs → No bugs or equivalently Analysis says bug → It is a bug Analysis says True → True
Completeness	"Complete for reporting correctness" No bugs → Analysis says no bugs True → analysis says True

	Complete	Incomplete
Sound		
Unsound		

	Complete	Incomplete
Sound	Reports all errors Reports no false alarms	
Unsound		

	Complete	Incomplete
Sound	Reports all errors Reports no false alarms **Undecidable**	
Unsound		

	Complete	Incomplete
Sound	Reports all errors Reports no false alarms **Undecidable**	Reports all errors May report false alarms
Unsound		

	Complete	Incomplete
Sound	Reports all errors Reports no false alarms **Undecidable**	Reports all errors May report false alarms **Decidable**
Unsound		

	Complete	Incomplete
Sound	Reports all errors Reports no false alarms **Undecidable**	Reports all errors May report false alarms **Decidable**
Unsound	May not report all errors Reports no false alarms	

	Complete	Incomplete
Sound	Reports all errors Reports no false alarms **Undecidable**	Reports all errors May report false alarms **Decidable**
Unsound	May not report all errors Reports no false alarms **Decidable**	

	Complete	Incomplete
Sound	Reports all errors Reports no false alarms **Undecidable**	Reports all errors May report false alarms **Decidable**
Unsound	May not report all errors Reports no false alarms **Decidable**	May not report all errors May report false alarms

	Complete	Incomplete
Sound	Reports all errors Reports no false alarms **Undecidable**	Reports all errors May report false alarms **Decidable**
Unsound	May not report all errors Reports no false alarms **Decidable**	May not report all errors May report false alarms **Decidable**

	Complete	Incomplete
Sound	Reports all errors Reports no false alarms **Undecidable**	Reports all errors May report false alarms **Decidable**
Unsound	May not report all errors Reports no false alarms **Decidable**	May not report all errors May report false alarms **Decidable**

	Complete	Incomplete
Sound	Reports all errors Reports no false alarms **Undecidable**	Reports all errors May report false alarms **Decidable**
Unsound	May not report all errors Reports no false alarms **Decidable**	May not report all errors May report false alarms **Decidable**

Common Program Analysis Techniques

Objectives

- give some indications on general optimization techniques:
  - **data-flow analysis**
  - register allocation
  - software pipelining
  - Etc.
- describe the main data structures used:
  - Call graph
  - control flow graph
  - intermediate code (e.g., 3-address code)
  - Static Single Assignment form (SSA)
  - etc

Generic Approach

- Decide – Intraprocedural or Interprocedural
  - Intraprocedural-- per function analysis (disregarding side effect of function calls)
  - Interprocedural- function analysis with considering side effects of function calls
- generation of a control flow graph (CFG)
- (optional) generation of interprocedural CFG (ICFG)
- Data-flow analysis of the (I)CFG

IR: 3-address code (TAC mode)

- Intermediate Representation “high-level” assembly code:
  - binary logic and arithmetic operators
  - use of temporary memory location ti
  - assignments to variables, temporary locations
  - a label is assigned to each instruction

IR: 3-address code (TAC mode)

- Intermediate Representation “high-level” assembly code:
  - binary logic and arithmetic operators
  - use of temporary memory location ti
  - assignments to variables, temporary locations
  - a label is assigned to each instruction

```
var1= (var2+ var3) + func(A)   ⇒   L1: t1 = var2 + var3
                                   L2: t2 = func(A)
                                   L3: var1= t1+t2
```

Basic Block (BB)

- A maximal sequence of instructions with single entry and single exit
  - => execution of BB is *atomic* (under normal condition)

Control Flow Graph (CFG)

- A representation of how the execution may progress inside a given function

$\rightarrow$ a graph (V, E) such that:

$$
\begin{aligned}
V &= \{B_i \mid B_i \text{ is a basic block}\} \\
E &= \{(B_i, B_j) \mid \\
&\quad \text{“last inst. of } B_i \text{ is a jump to 1st inst of } B_j\text{”} \vee \\
&\quad \text{“1st inst of } B_j \text{ follows last inst of } B_i \text{ in the TAC”}\}
\end{aligned}
$$

Call Graph

- Computed for a whole program (i.e. interprocedural by definition)
- Represented as a directed graph (V,E)
- where $V = \{F_i | F_i \text{ is a function}\}$
  - $E = \{(F_i, F_j) | F_i \text{ calls } F_j\}$

Intraprocedural Computation over CFG

- associate (local) properties to entry/exit points of Bbs, e.g. set of active variables, set of available expressions, etc.)
- propagate them along CFG paths
- (context- sensitivity adds precision, but difficult”)
- update each BB (and CFG edges) according to these global properties

Dataflow Analysis

Static computation of the data related properties of the program

Dataflow Analysis

Static computation of the data related properties of the program

(local) properties φ_i associated to some pgm locations i

Dataflow Analysis

Static computation of the data related properties of the program

(local) properties φ_i associated to some pgm locations i

set of data-flow equations:

$\rightarrow$ how φ_i are transformed along pgm execution

Rks:

- forward vs backward propagation (depending on φ_i)
- cycles inside the control flow $\Rightarrow$ fix-point equations !

Dataflow Analysis

Static computation of the data related properties of the program

(local) properties φ_i associated to some pgm locations i

set of data-flow equations:

$\rightarrow$ how φ_i are transformed along pgm execution

Rks:

- forward vs backward propagation (depending on φ_i)
- cycles inside the control flow $\Rightarrow$ fix-point equations !

a solution of this equation system:

$\rightarrow$ assigns "globaly consistent" values to each φ_i

Rk: such a solution may not exist . . .

Example: redundant Expression

Example: redundant Expression

An expression e is redundant at location i iff

- it is computed at location i
- this expression is computed on every path going from the initial location to location i
  Rk: we consider here syntactic equality
- on each of these paths: operands of e are not modified between the last computation of e and location i

Example: redundant Expression

An expression e is redundant at location i iff

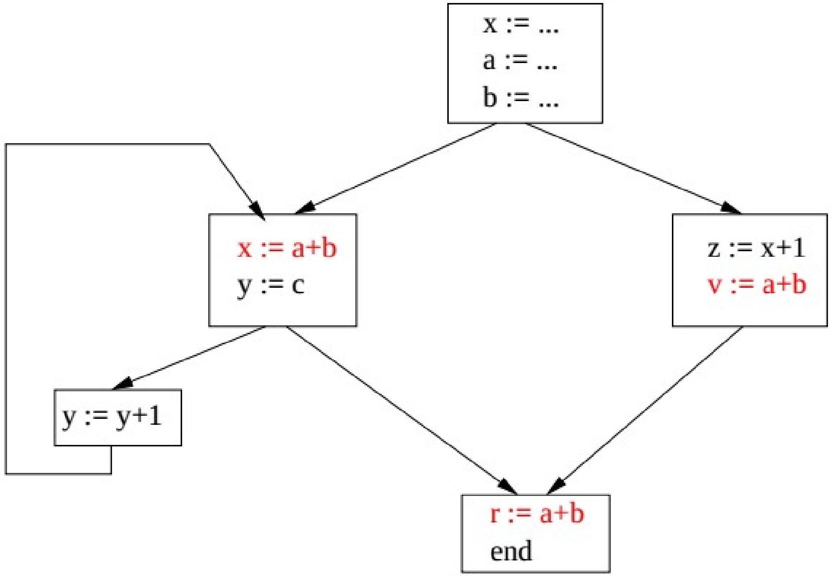
- it is computed at location i
- this expression is computed on every path going from the initial location to location i
  Rk: we consider here syntactic equality
- on each of these paths: operands of e are not modified between the last computation of e and location i

Optimization is performed as follows:

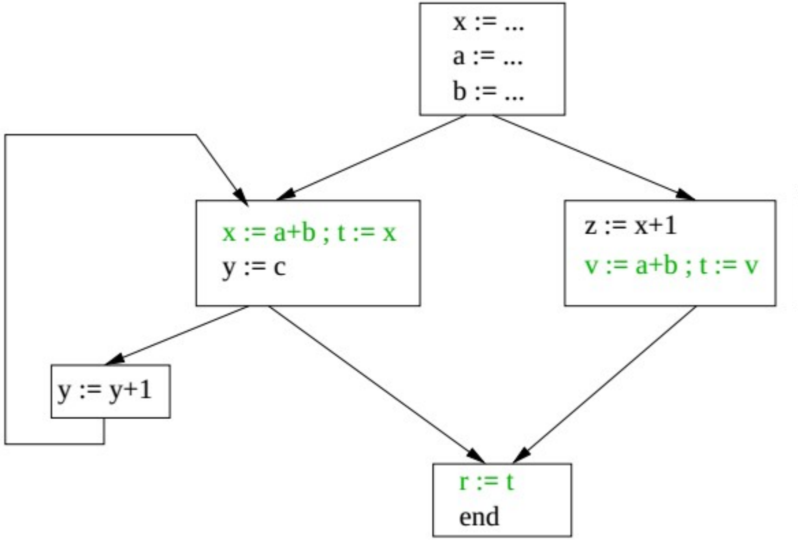
1. computation of available expressions (data-flow analysis)
2. $x := e$ is redundant at loc i if e is available at i
3. $x := e$ is replaced by $x := t$
   (where t is a temp. memory containing the value of e)

Example conti..

Example conti..



Example conti..



Dataflow Eqn for available expressions 1/2

For a basic block b, we note:

- $In(b)$: available expressions when entering b
- $Kill(b)$: expressions made non available by b (because an operand of e is modified by b)
- $Gen(b)$: expressions made available by block b (computed in b, operands not modified afterwards)
- $Out(b)$: available expressions when exiting b

$$Out(b) = (In(b) \setminus Kill(b)) \cup Gen(b) = F_b(In(b))$$

F_b = transfer function of block b

Dataflow Eqn for available expressions 2/2

How to compute $In(b)$?

- if b is the initial block:

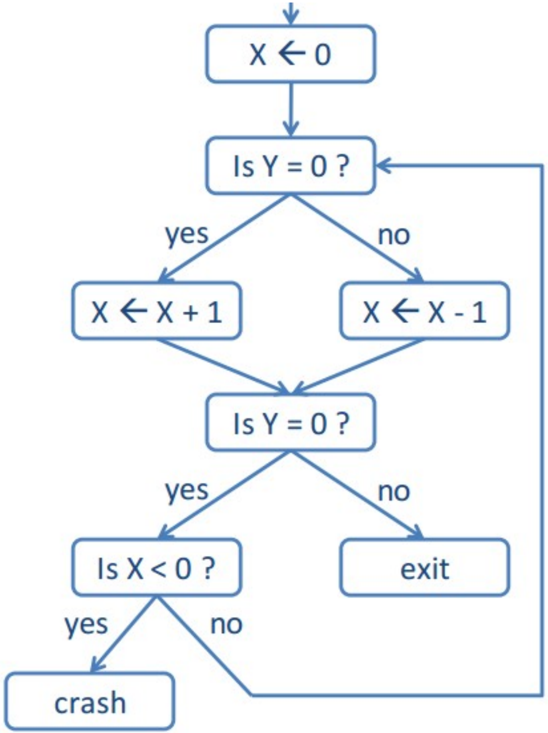
$$In(b) = \emptyset$$

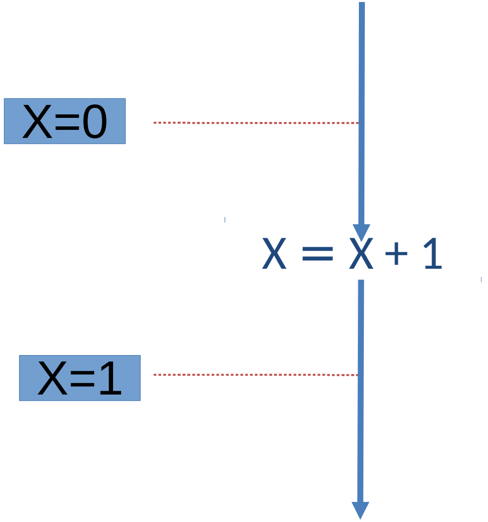
- if b is not the initial block:
  An expression e is available at its entry point iff it is available at the exit point of each predecessor of b in the CFG

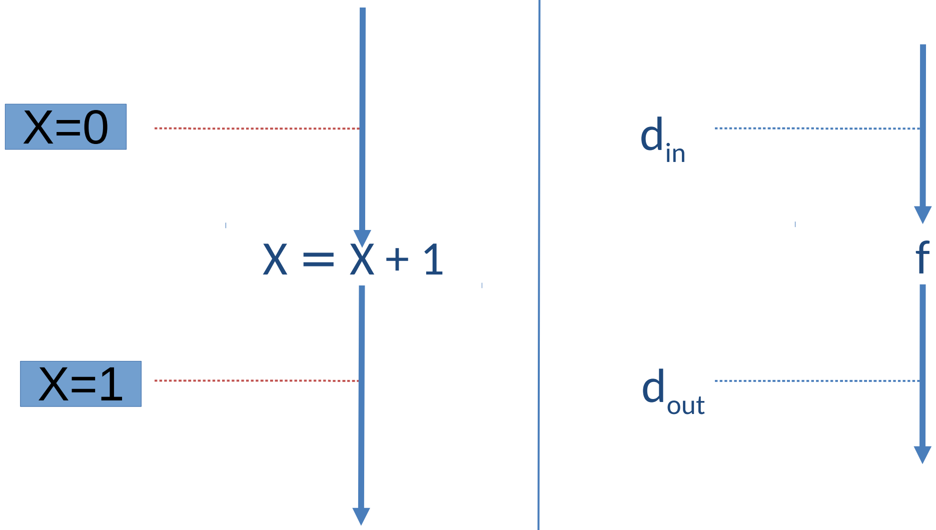
$$In(b) = \bigcap_{b' \in Pre(b)} Out(b')$$

$\Rightarrow$ forward data-flow analysis along the CFG paths

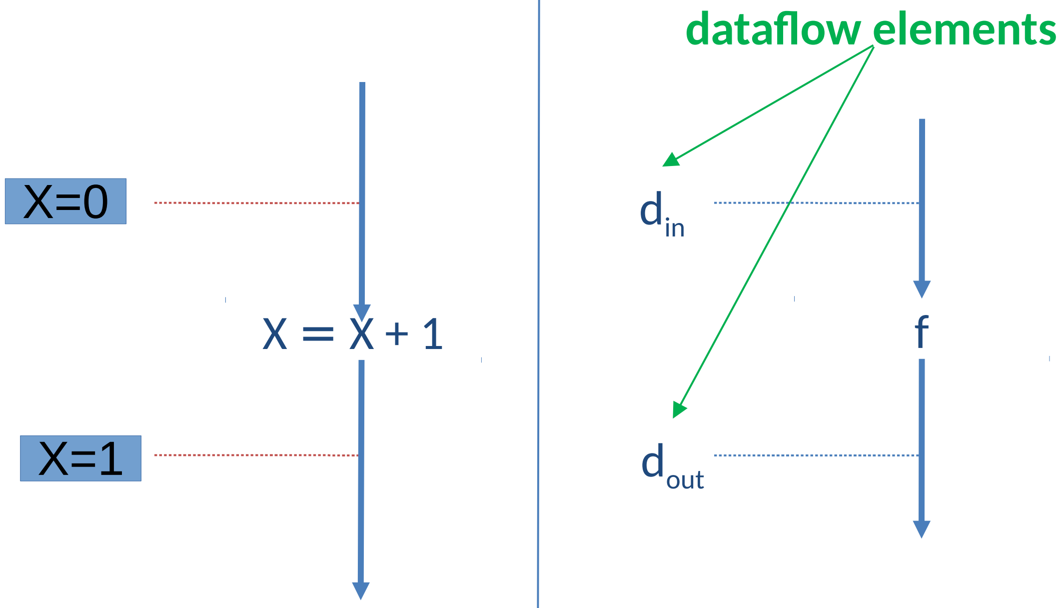
Simple Program



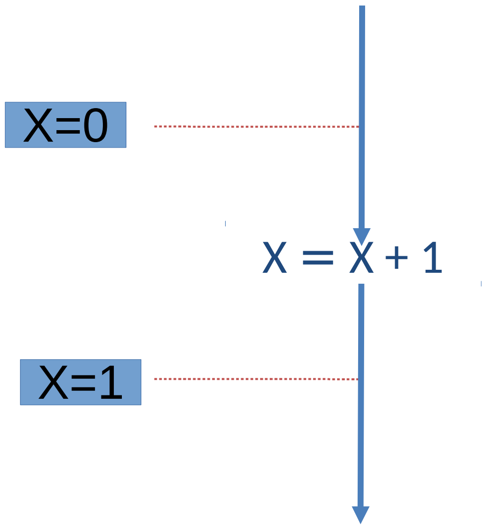

X=0
X = X + 1
X=1

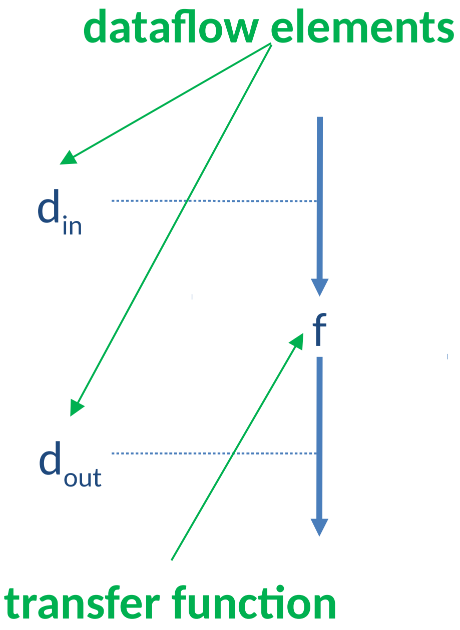


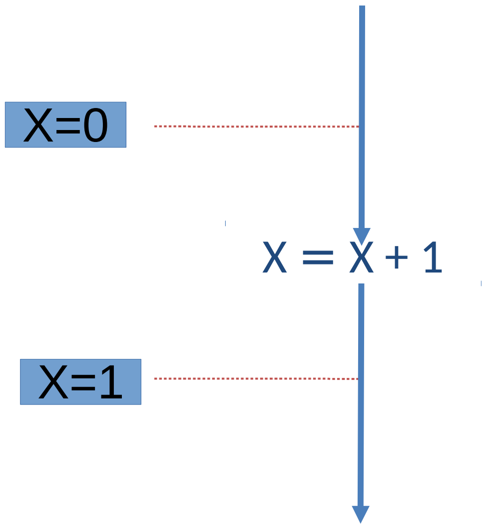
X=0
X = X + 1
X=1
d
in
f
d
out

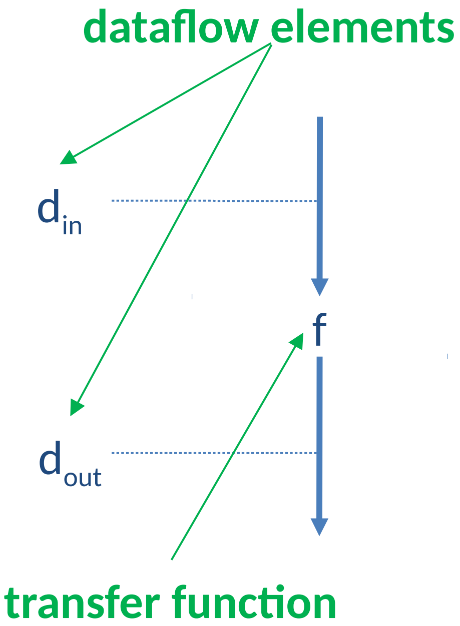


X=0
X = X + 1
X=1
dataflow elements
d
in
f
d
out

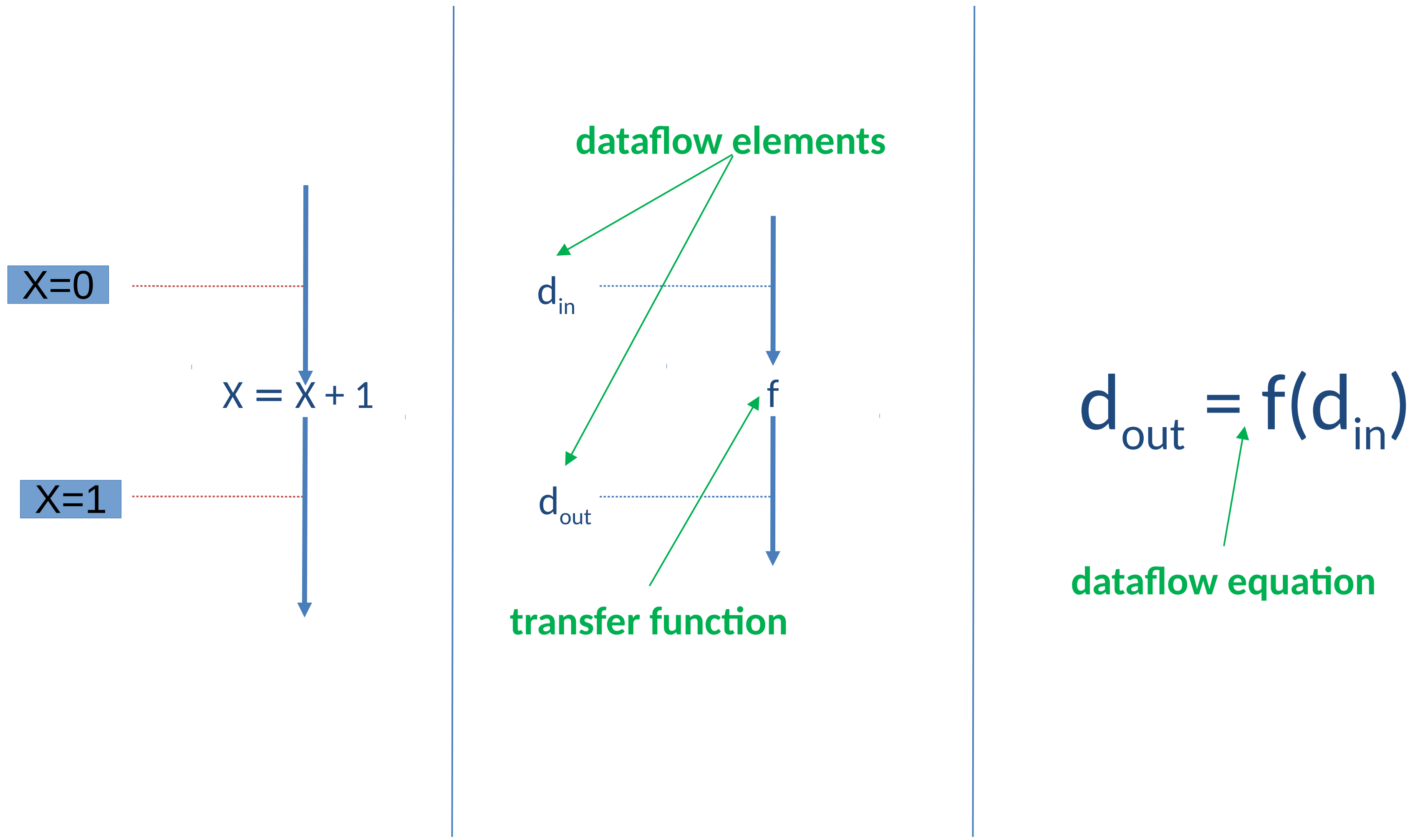

X=0
X = X + 1
X=1


dataflow elements
d in
f
d out
transfer function

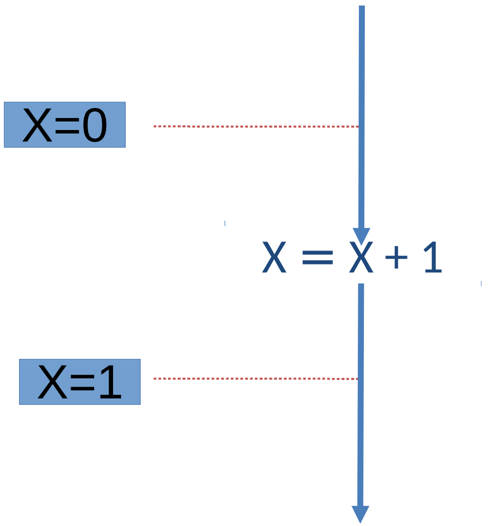


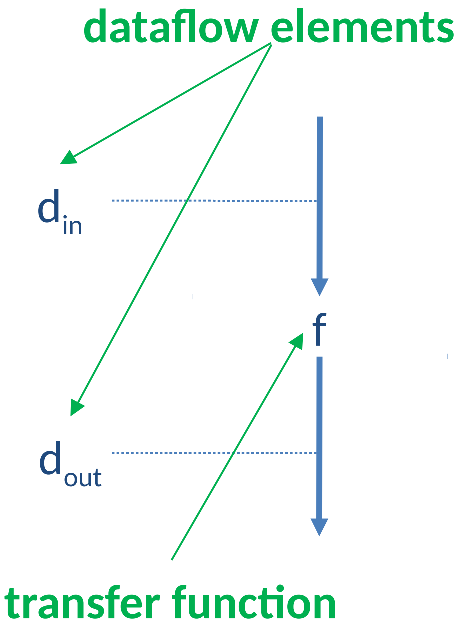


$$d_{out} = f(d_{in})$$



X=0
X = X + 1
X=1
dataflow elements
d_{in}
f
d_{out}
transfer function
$d_{out} = f(d_{in})$
dataflow equation

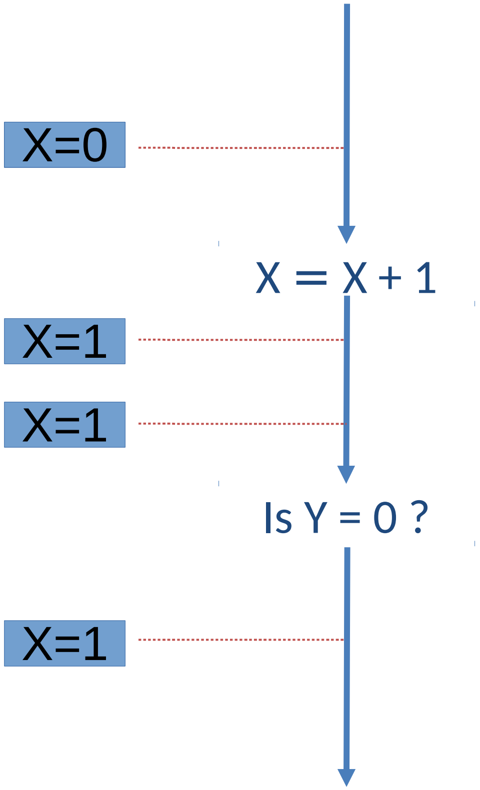


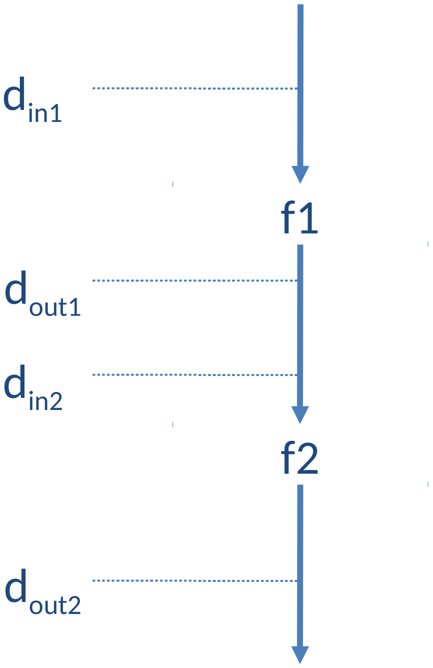


$$d_{out} = f(d_{in})$$

dataflow equation

$$Out(b) = (In(b) \setminus Kill(b)) \cup Gen(b) = F_b(In(b))$$

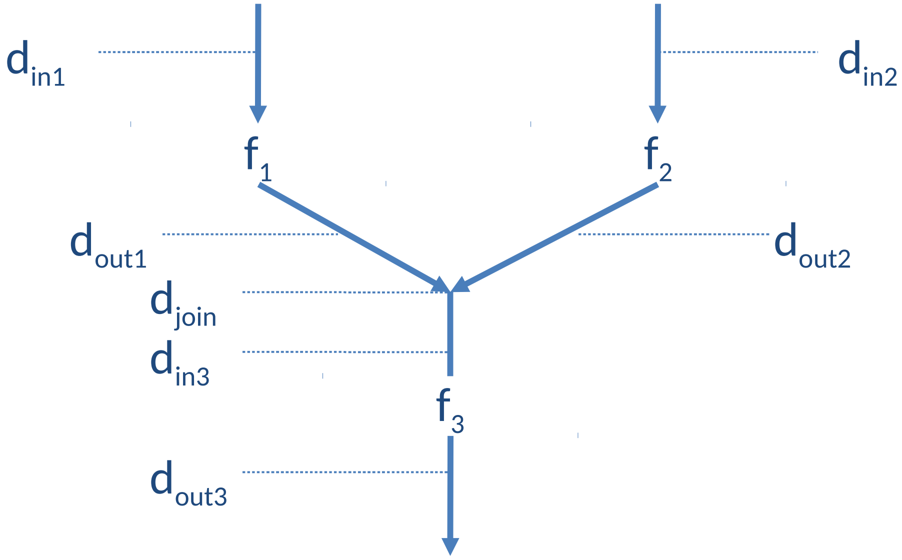




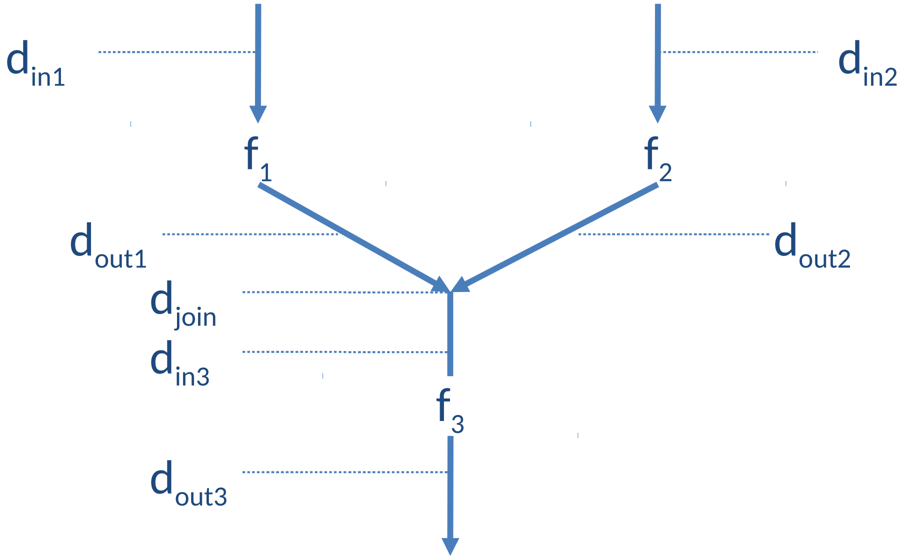
$$d_{out1} = f_1(d_{in1})$$

$$d_{in2} = d_{out1}$$

$$d_{out2} = f_2(d_{in2})$$



$$d_{out1} = f_1(d_{in1})$$
$$d_{out2} = f_2(d_{in2})$$
$$d_{join} = d_{out1} \sqcup d_{out2}$$
$$d_{join} = d_{in3}$$
$$d_{out3} = f_3(d_{in3})$$



$$d_{out1} = f_1(d_{in1})$$

$$d_{out2} = f_2(d_{in2})$$

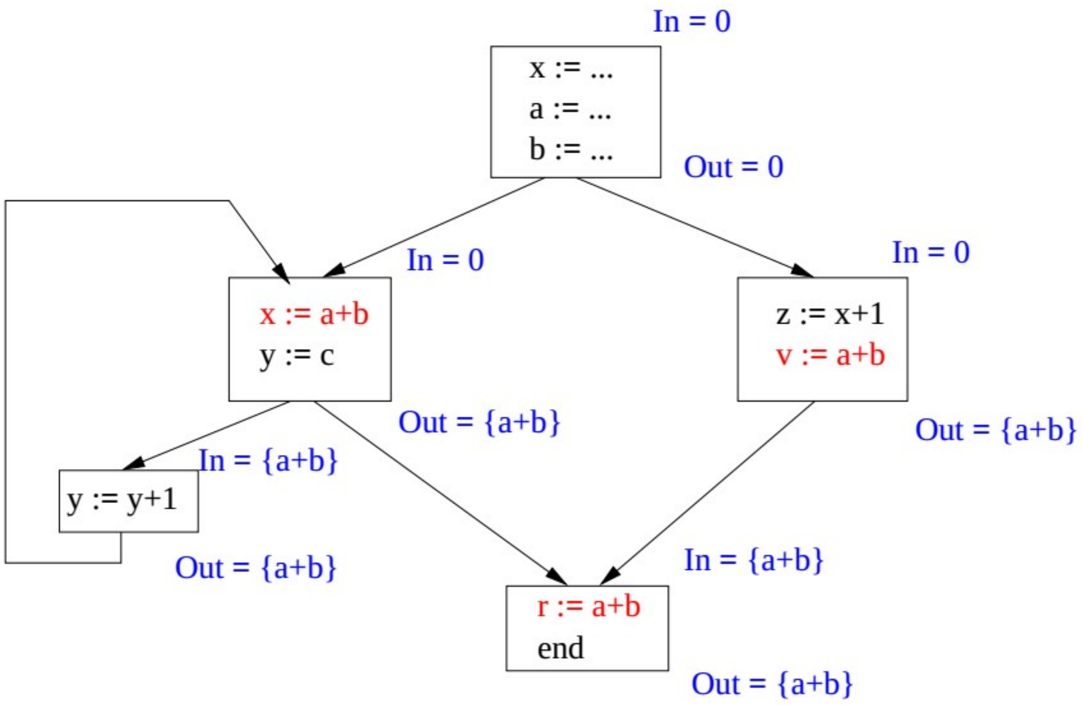
$$d_{join} = d_{out1} \sqcup d_{out2}$$

$$d_{join} = d_{in3}$$

$$d_{out3} = f_3(d_{in3})$$

least upper bound operator
Example: union of possible values

Back to Example



Generalization

Generalization

- Data-flow properties are expressed as finite sets associated to entry/exit points of basic blocs: `In`(b), `Out`(b)

Generalization

- Data-flow properties are expressed as finite sets associated to entry/exit points of basic blocs: `In`(b), `Out`(b)
- For a forward analysis:
  - property is "false" ($\perp$) at entry of initial block
  - $\texttt{Out}(b) = F_b(\texttt{In}(b))$
  - `In`(b) depends on `Out`(b'), where $b' \in Pred(b)$ ($\sqcap$ for "$\forall$ paths", $\sqcup$ for "$\exists$ path")

Generalization

- Data-flow properties are expressed as finite sets associated to entry/exit points of basic blocs: `In`(b), `Out`(b)
- For a forward analysis:
  - property is "false" ($\perp$) at entry of initial block
  - $\mathtt{Out}(b) = F_b(\mathtt{In}(b))$
  - `In`(b) depends on `Out`(b'), where $b' \in Pred(b)$
    ($\sqcap$ for "$\forall$ paths", $\sqcup$ for "$\exists$ path")
- For a backward analysis:
  - property is "false" ($\perp$) at exit of final block
  - $\mathtt{In}(b) = F_b(\mathtt{Out}(b))$
  - `Out`(b) depends on `In`(b'), where $b' \in Succ(b)$

Forward Analysis

Forward analysis, least fix-point	$\mathtt{In}(b) = \begin{cases} \bot & \text{if b is initial} \\ \bigsqcup_{b' \in Pre(b)} \mathtt{Out}(b') & \text{otherwise.} \end{cases}$ $\mathtt{Out}(b) = F_b(\mathtt{In}(b))$
Forward analysis, greatest fix-point	$\mathtt{In}(b) = \begin{cases} \bot & \text{if } b \text{ is initial} \\ \bigsqcap_{b' \in Pre(b)} \mathtt{Out}(b') otherwise. \end{cases}$ $\mathtt{Out}(b) = F_b(\mathtt{In}(b))$

Backward Analysis

Backward analysis, least fix-point	$\mathtt{Out}(b) = \begin{cases} \bot & \text{if } b \text{ is final} \\ \bigsqcup_{b' \in Succ(b')} \mathtt{In}(b') otherwise. \end{cases}$ $\mathtt{In}(b) = F_b(\mathtt{Out}(b))$
Backward analysis, greatest fix-point	$\mathtt{Out}(b) = \begin{cases} \bot & \text{if } b \text{ is final} \\ \bigsqcap_{b' \in Succ(b)} \mathtt{In}(b') otherwise. \end{cases}$ $\mathtt{In}(b) = F_b(\mathtt{Out}(b))$

Reaching Definition

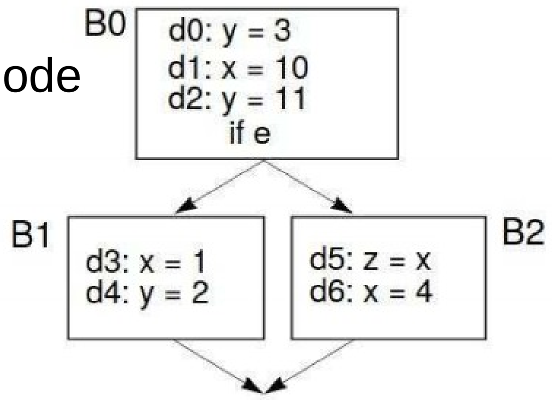
- Every assignment is a definition.
- A **definition** d **reaches** a point p if **there** exists a path from the point immediately following d to p such that d is not killed (overwritten) along that path.
- in terms of Gen and $Kill$, every assignment generates and another assignment of the same variable kills the previous assignment and generates the newer one.

Least fixed point
i.e. union at phi node

Reaching Definition

- Every assignment is a definition.
- A **definition** d **reaches** a point p if **there** exists a path from the point immediately following d to p such that d is not killed (overwritten) along that path.
- in terms of Gen and $Kill$, every assignment generates and another assignment of the same variable kills the previous assignment and generates the newer one.

Least fixed point
i.e. union at phi node



Bug Detection- Uninitialized use

Bug Detection- Uninitialized use

- A variable is used before it is defined → undefined behaviour class of bugs

Bug Detection- Uninitialized use

- A variable is used before it is defined → undefined behaviour class of bugs

```
fn(int a, int b)
{
    int i, j, k, x;
    i = a;
    j = b;
    if(i<j)
      k=j-i;
    if(j<i)
      k=i-j;
    x = 100+k;
    return x;
}
```

Bug Detection- Uninitialized use

- A variable is used before it is defined → undefined behaviour class of bugs

Reaching def. Type analysis,

```
fn(int a, int b)
{
    int i, j, k, x;
    i = a;
    j = b;
    if(i<j)
      k=j-i;
    if(j<i)
      k=i-j;
    x = 100+k;
    return x;
}
```

Bug Detection- Uninitialized use

- A variable is used before it is defined → undefined behaviour class of bugs

Reaching def. Type analysis,
but with greatest fixed-point, i.e. all paths

```
fn(int a, int b)
{
   int i, j, k, x;
   i = a;
   j = b;
   if(i<j)
     k=j-i;
   if(j<i)
     k=i-j;
   x = 100+k;
   return x;
}
```

Bug Detection- Uninitialized use

- A variable is used before it is defined → undefined behaviour class of bugs

```
fn(int a, int b)
{
   int i, j, k, x;
   i = a;
   j = b;
   if(i<j)
     k=j-i;
   if(j<i)
     k=i-j;
   x = 100+k;
   return x;
}
```

Reaching def. Type analysis,
but with greatest fixed-point, i.e. all paths

Dataflow element: set of defined (*Gen*) vars
i.e. a value is assigned

Bug Detection- Uninitialized use

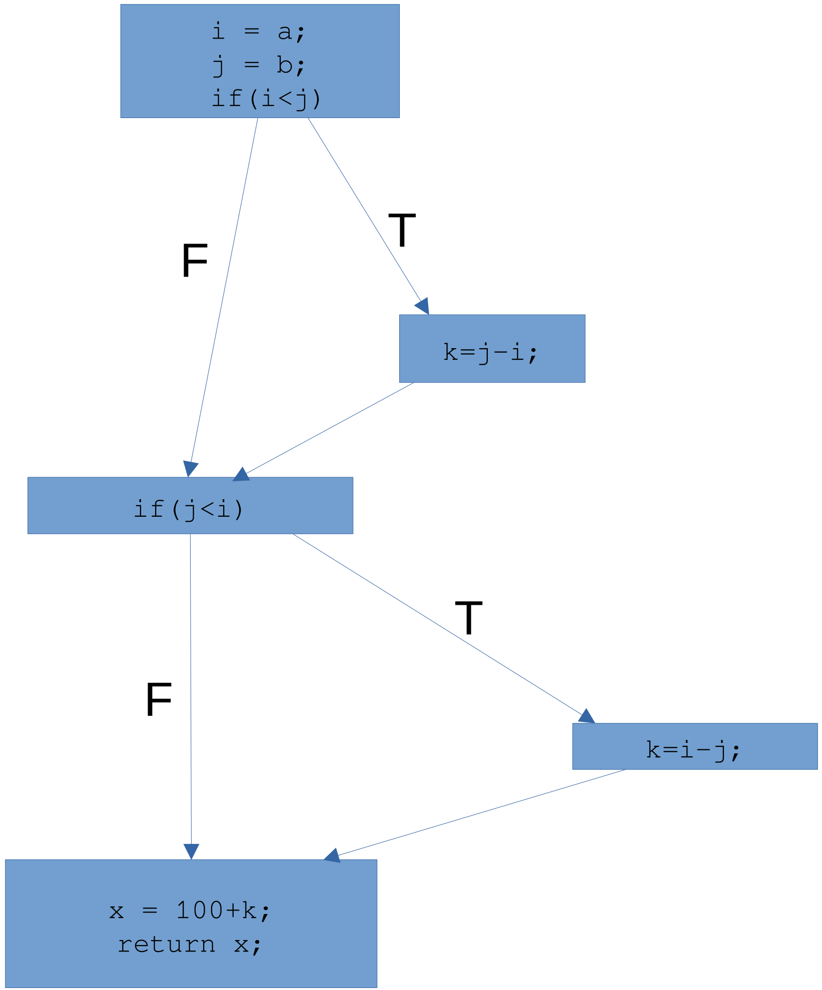
- A variable is used before it is defined → undefined behaviour class of bugs

```
fn(int a, int b)
{
   int i, j, k, x;
   i = a;
   j = b;
   if(i<j)
     k=j-i;
   if(j<i)
     k=i-j;
   x = 100+k;
   return x;
}
```

Reaching def. Type analysis,
but with greatest fixed-point, i.e. all paths

Dataflow element: set of defined (*Gen*) vars
i.e. a value is assigned

We stop when we do not get reaching def of a variable being used in a instruction.



i = a;
j = b;
if(i<j)
F
T
k=j-i;
if(j<i)
F
T
k=i-j;
x = 100+k;
return x;

Use-after-free

- Vulnerability:
  - Using (dereferencing) a pointer (memory) after it has been freed (i.e. `free(p)`)
  - The same logic works with the following dataflow:
    - Creating a pointer (malloc or stack variable) defines (*Gen*) a pointer
    - Freeing the pointer *Kill* it.
    - Meet is greatest fixed-point i.e. all paths
    - For every dereference, reaching def should be present.