NTNU | Norwegian University of Science and Technology

Compiler Construction

Lecture 19 part 1: Data flow analyses – Overview

Michael Engel

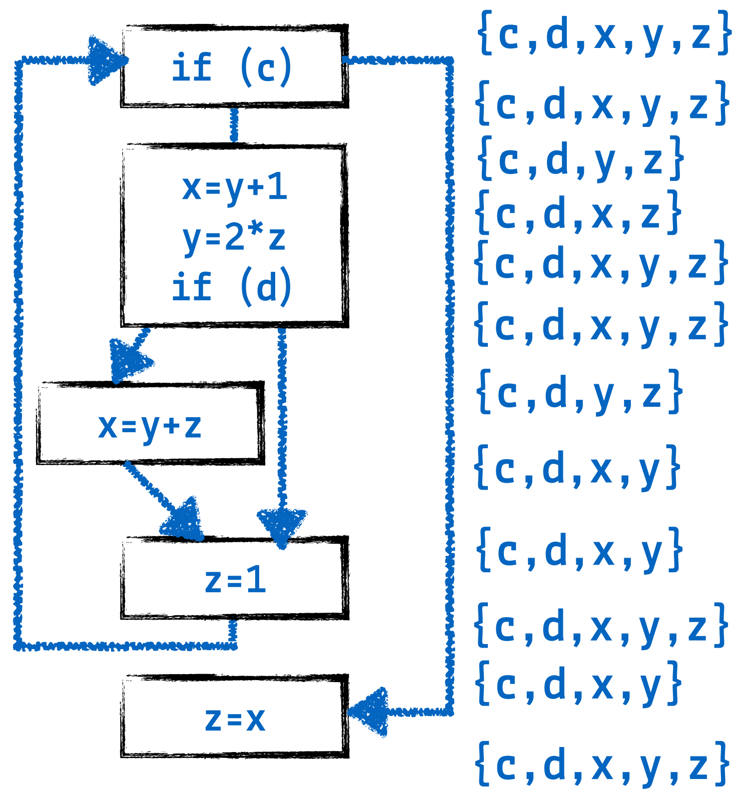
Classical bit-vector data-flow analyses

- Origins of data flow analysis were the so-called “bit vector” data flow frameworks [1]
  - called “bit vector” since data flow and additional information are represented using bit vectors
  - the analysis can be performed using bit vector operations alone
- There are forms of data flow which require additional operations for performing analyses
  - the data flow information itself is still represented using bit vectors
  - we make this notion more precise later with the help of the examples presented here

Bit vectors as set representations

- Using bit vectors enables an **efficient** implementation of sets
- Example: set with 32 elements
  - The presence or absence of each element is represented by a specific bit set to 1 or 0, respectively
- Representation of variables `c`,`d`,`x`,`y`,`z` as 5 bit set:

Bit:	0	1	2	3	4	
	1	0	1	1	0	⇒ `{c,x,y}`
Variable:	c	d	x	y	z	



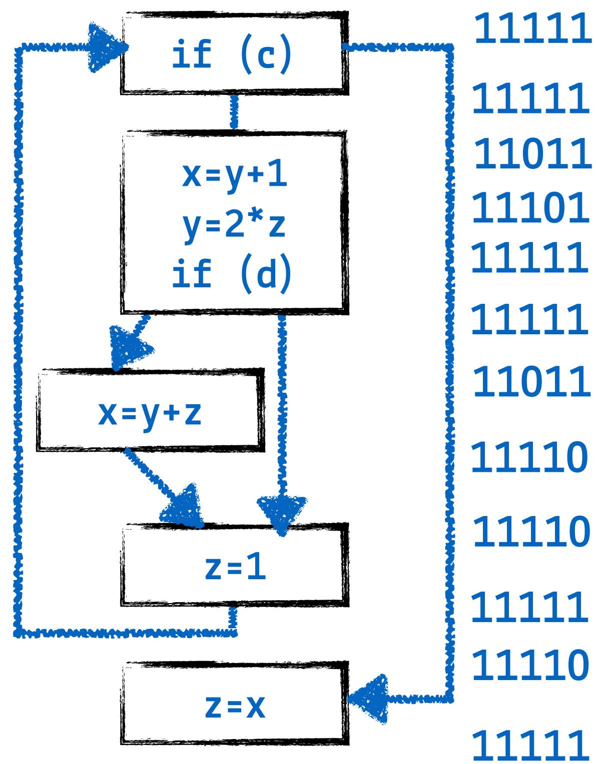
Bit vectors as set representations

- We can use bit vectors to represent the sets of **live variables** at the program points of the example in lecture 17

efficient as long as the number of elements fits in a machine word

Bit:	0	1	2	3	4
	1	0	1	1	0
Variable:	c	d	x	y	z

$\Rightarrow$ `{c,x,y}`



Set operations on bit vectors

- Typical set operations can now be implemented using boolean logic operators
- Example: **union** (join) of two sets (∪) using **OR**:

Variable:	c	d	x	y	z	
	1	0	1	1	0	⇒ {c,x,y}
OR	0	0	1	0	1	⇒ {x,z}
	1	0	1	1	1	⇒ {c,x,y,z}={c,x,y}∪{x,z}

- The set property that each element may only occur once in a set is guaranteed by mapping set elements to bits

Set operations on bit vectors

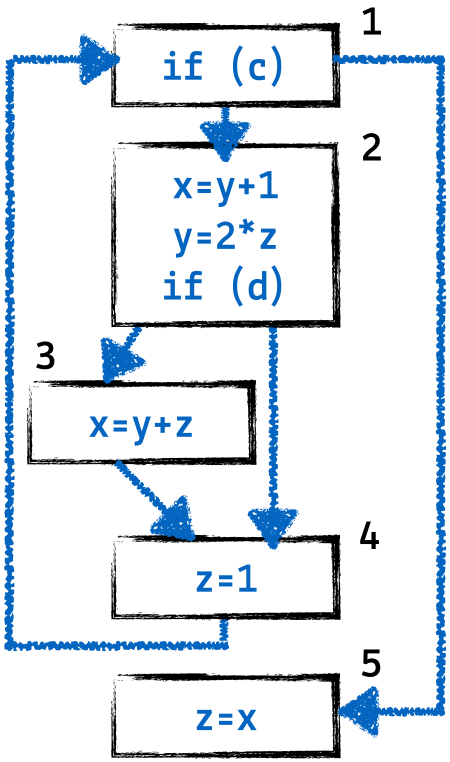
- Typical set operations can now be implemented using boolean logic operators
- Example: **intersection** (meet) of two sets (∩) using **AND**:

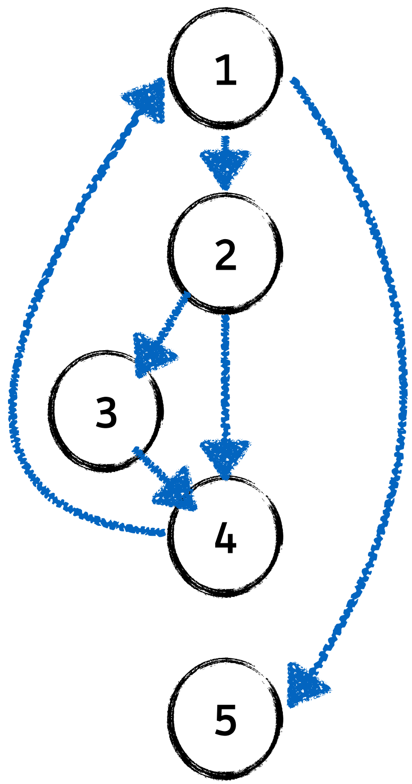
Variable:	c	d	x	y	z	
	1	0	1	1	0	⇒ {c,x,y}
AND	1	0	0	1	1	⇒ {c,y,z}
	1	0	0	1	0	⇒ {c,y} = {c,x,y}∩{c,y,z}

- Set complement can be implemented using **XOR**

Bit vectors to represent graphs

- We can also use bit vectors to represent graphs (e.g. CFG)
- Bit vectors can represent a graph’s ***adjacency matrix***



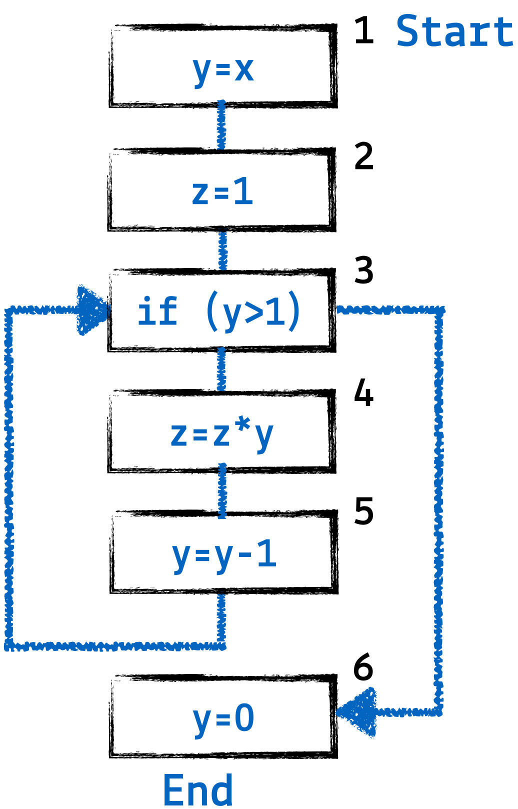


(i,j)=1
⇒ edge from node i to j

Basic Block (i \ j)	1	2	3	4	5
1	0	1	0	0	1
2	0	0	1	1	0
3	0	0	0	1	0
4	1	0	0	0	0
5	0	0	0	0	0

Properties of CFGs

- Edges in CFGs denote **predecessor** and **successor** relationships
- For an edge `n1→n2`:
  - `n1` is a **predecessor** of `n2` (`n1=pred(n2)`)
  - `n2` is a **successor** of `n1` (`n2=succ(n1)`)
- CFG has two distinguished unique nodes:
  - `Start` which has no predecessor
  - `End` which has no successor
- Every basic block `n` is reachable from the `Start` block and the `End` block is reachable from `n`



Overview of data-flow analyses

- Data flow analysis views computation of data through expressions and transition of data through assignments to variables
- Properties of programs are defined in terms of properties of program entities such as expressions, variables, and definitions appearing in a program
  - we restrict expressions to primitive expressions involving a single operator
  - variables are restricted to scalar variables and definitions are restricted to assignments made to scalar variables
    (let's keep it moderately simple…)

General approach

- For a given program entity such as an expression, data flow analysis of a program involves the following two steps
  - (a) discovering the effect of individual statements on the expression, and
  - (b) relating these effects across statements in the program
- For reasons of efficiency, both steps are often carried over a basic block instead of a single statement
- Step (a) is called **local** data flow analysis and is performed for a basic block only once
- Step (b) constitutes **global** data flow analysis and may require repeated traversals over basic blocks in a CFG

***"global"* here means: inside a single procedure!**

Discovering local data flow information

- The modelling of the effect of a statement varies between different analyses
- However, there is a common pattern of generation of data flow information or invalidation of data flow information

Entity	**Operations**	
Variable **x**	Reading the value of **x** ***(use)***	Modifying the value of **x**
Expression **e**	Computing **e**	Modifying an operand of **e**
Definition $\mathtt{d_i{:}x{=}e}$	Occurence of $\mathtt{d_i}$	Any definition of **x**

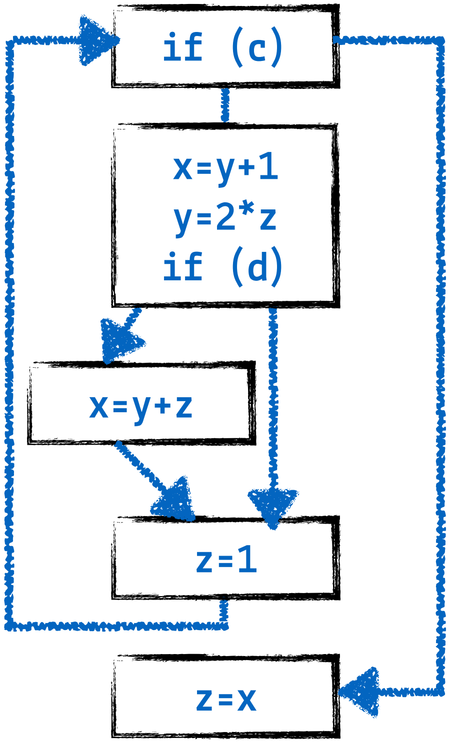
Entities and operations

- A **variable** may be used or an **expression** may be computed
  (a) in the right hand side of an assignment statement,
  (b) in a condition for altering the control flow,
  (c) as an actual parameter in a function call, or
  (d) as a return value from a function
- All other operations involve an **assignment** statement to a relevant variable
- Note that **reading** a value of a variable **from input** can be safely considered as an assignment statement assigning an unknown value to the variable

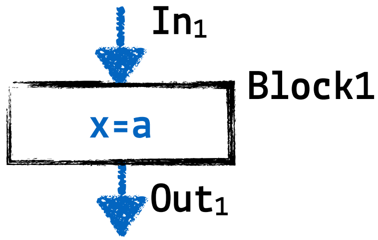
Entity	Operations	
Variable `x`	Reading the value of `x` ***(use)***	Modifying the value of `x`
Expression `e`	Computing `e`	Modifying an operand of `e`
Definition d_i`:x=e`	Occurence of d_i	Any definition of `x`

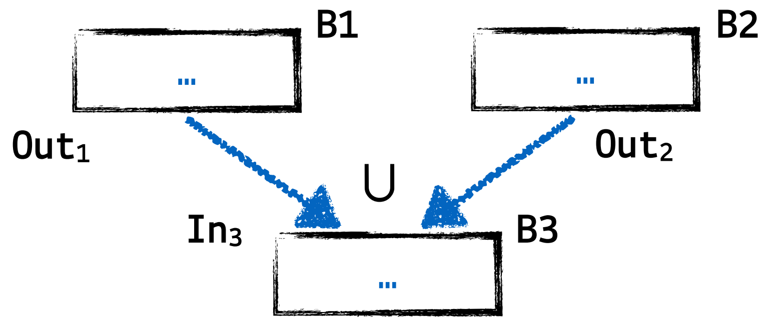
Relationship of CFG information

- The relationship between local and global data flow information for a block and between global data flow information across different blocks is captured by **data flow equations**
    - this is a system of linear simultaneous equations
- In general, these equations have multiple solutions



Data flow equations

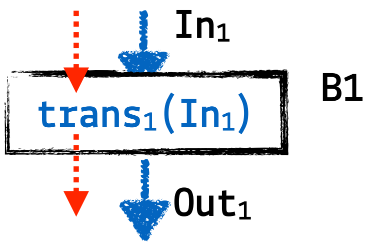




- In forward flow analysis, the exit state of a basic block ***b*** is a function *(**data flow equation**)* of the block's entry state: $\boldsymbol{Out_b = trans_b(In_b)}$
  - composition of the effects of the statements in the block
  - $\boldsymbol{trans_b}$ is the ***transition function*** of block ***b***
- The entry state of a basic block is a function *(**data flow equation**)* of the exit states of its predecessors: $\boldsymbol{In_b} = join_{p \in pred(b)}\boldsymbol{(Out_p)}$
  - The join operation $join_{p \in pred(n)}$ combines the exit states of all predecessors ***p*** of ***b***, yielding the entry state of ***b***

Data-flow analysis directions

Each type of data-flow analysis has a **specific transfer function and join operation**

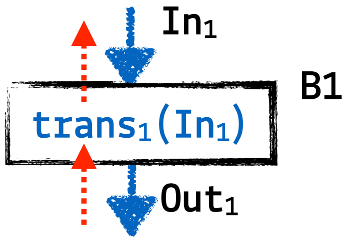


Forward analyses traverse the CFG **along** the direction of the control flow

- e.g. reaching definitions, available expressions

Backward analyses traverse the CFG **against** the direction of the control flow

- e.g. live variables analysis, very busy expressions
- Here, the transfer function is applied to the exit state yielding the entry state
  - the join operation works on the entry states of the successors to yield the exit state



Bit vector analysis: Gen and Kill sets

Bit vector dataflow analyses works on ***sets of facts***

- these sets can be represented efficiently as bit vectors

- Join and transfer functions are implemented as logical bitwise ops
  - join is typically ∪ or ∩, implemented by logical or / logical and
  - transfer functions can be decomposed into ***Gen*** and ***Kill*** sets

- ***Gen****:* points in the graph where a fact you care about becomes ***true***
  - ***Gen_b*** describes data flow info. generated within block **b**

- ***Kill****:* points in the graph where a fact you care about becomes ***false***
  - ***$Kill_b$*** describes data flow inf. which becomes invalid in block **b**
- ***Gen_b*** and ***$Kill_b$*** points thus depend on the facts you care about

Example: gen and kill sets

- ***Example***: in ***live-variable analysis***, the **join** operation is **union**
- **Kill** set: **variables that are written** in a block
- **Gen** set: **variables that are read without being written** first
- The related data-flow equations are thus:

$$\boldsymbol{Out_b} = \bigcup_{s \in succ(b)} \boldsymbol{In_s} \qquad \boldsymbol{In_b = (Out_b - Kill_b) \cup Gen_b}$$

- In logical operations:

```
out(b) = 0          // empty set
for s in succ(b) {
    out(b) = out(b) or in(s)
}
in(b) = (out(b) and not kill(b))
        or gen(b)
```

References

[1] Uday P. Khedker, Amitabha Sanyal, Bageshri Karkare. Data Flow Analysis: Theory and Practice. CRC Press, 2009 (Chapter 2, Classical Bit Vector Data Flow Analysis)

[2] Flemming Nielson, Hanne Riis Nielson, Chris Hankin. Principles of Program Analysis. Springer, 2nd edition, 2005 (Chapter 2, Data Flow Analysis)

[3] Robert Morgan. Building an Optimizing Compiler. Digital Press, 1998 (Chapter 4.12, Global Available Temporary Information)

[4] Gary Kildall. A Unified Approach to Global Program Optimization. Proceedings of the 1st ACM Symposium on Principles of Programming Languages (POPL), 1973