ETH Zürich
Chair of Software Engineering

Software Verification

Sebastian Nanz

Lecture 7: Data Flow Analysis, Slicing

Program Slicing

Program slicing

```
1    sum := 0
2    prod := 1
3    i := 0
4    while i < y do
5        sum := sum + x
6        prod := prod * x
7        i := i + 1
     end
8    print(sum)
9    print(prod)
```

```
1    sum := 0

3    i := 0
4    while i < y do
5        sum := sum + x

7        i := i + 1
     end
8    print(sum)
```

"What program statements potentially affect the value of variable sum at line 8 of the program?"

Program slicing

- Program slicing provides an answer to the question

> "What program statements potentially affect the values of the variables at program point l?"

- The resulting program statements are called the program slice.
- The program point l is called the slicing criterion.
- An observer focusing on the slicing criterion (i.e. only observing values of the variables at program point l) cannot distinguish a run of the program from the run of its slice.

Applications of program slicing

- **Debugging:** Slicing lets the programmer focus on the program part relevant to a certain failure, which might lead to quicker detection of a fault.
- **Testing:** Slicing can minimize test cases, i.e. find the smallest set of statements that produces a certain failure (good for regression testing).
- **Parallelization:** Slicing can determine parts of the program which can be computed independently of each other and can thus be parallelized.

Classification

- Static slicing vs. dynamic slicing
  - Static: general, not considering a particular input
  - Dynamic: computed for a fixed input, therefore smaller slices can be obtained
- Backward slicing vs. forward slicing
  - Backward: "Which statements affect the execution of a statement?"
  - Forward: "Which statements are affected by the execution of a certain statement?"
- In the following we present an algorithm for static backward slicing.

Program slice

A backward slice S of program P with respect to slicing criterion l is any executable program with the following properties:

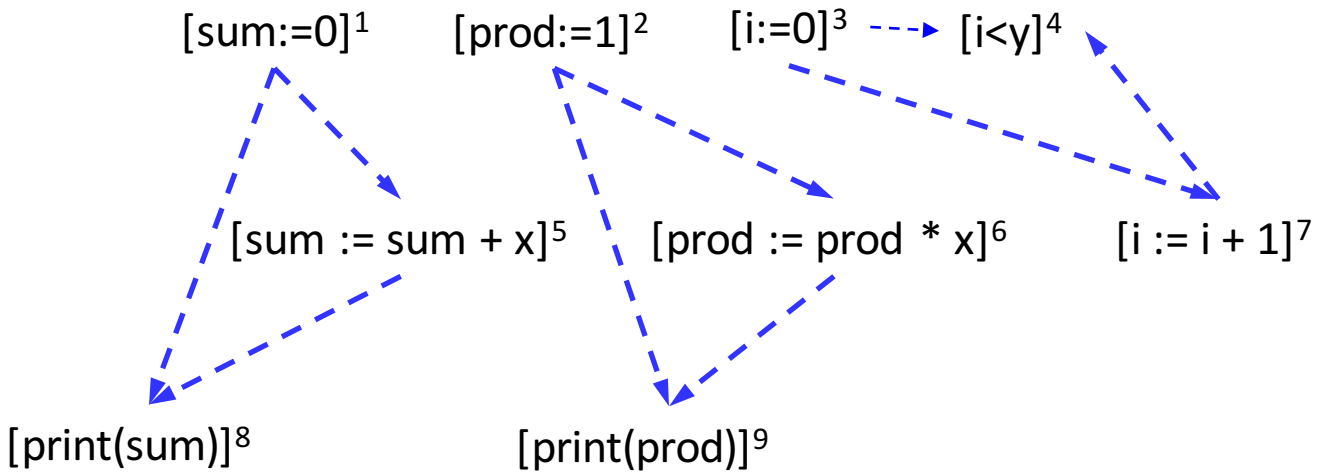
1. S can be obtained by deleting zero or more statements from P.
2. If P halts on input I, then the values of the variables at program point l are the same in P and in S every time program point l is executed.

Slicing algorithm

- We present a slicing algorithm for static backward slicing.
- Many different approaches, we show one that constructs a program dependence graph (PDG).
- A PDG is a directed graph with two types of edges:
  - Data dependencies: given by data-flow analysis
  - Control dependencies: program point l is control-dependent on program point l' if
    (1) l' labels the guard of a control structure
    (2) the execution of l depends on the outcome of the evaluation of the guard at l'

Example: Program dependence graph

1. Data dependence subgraph

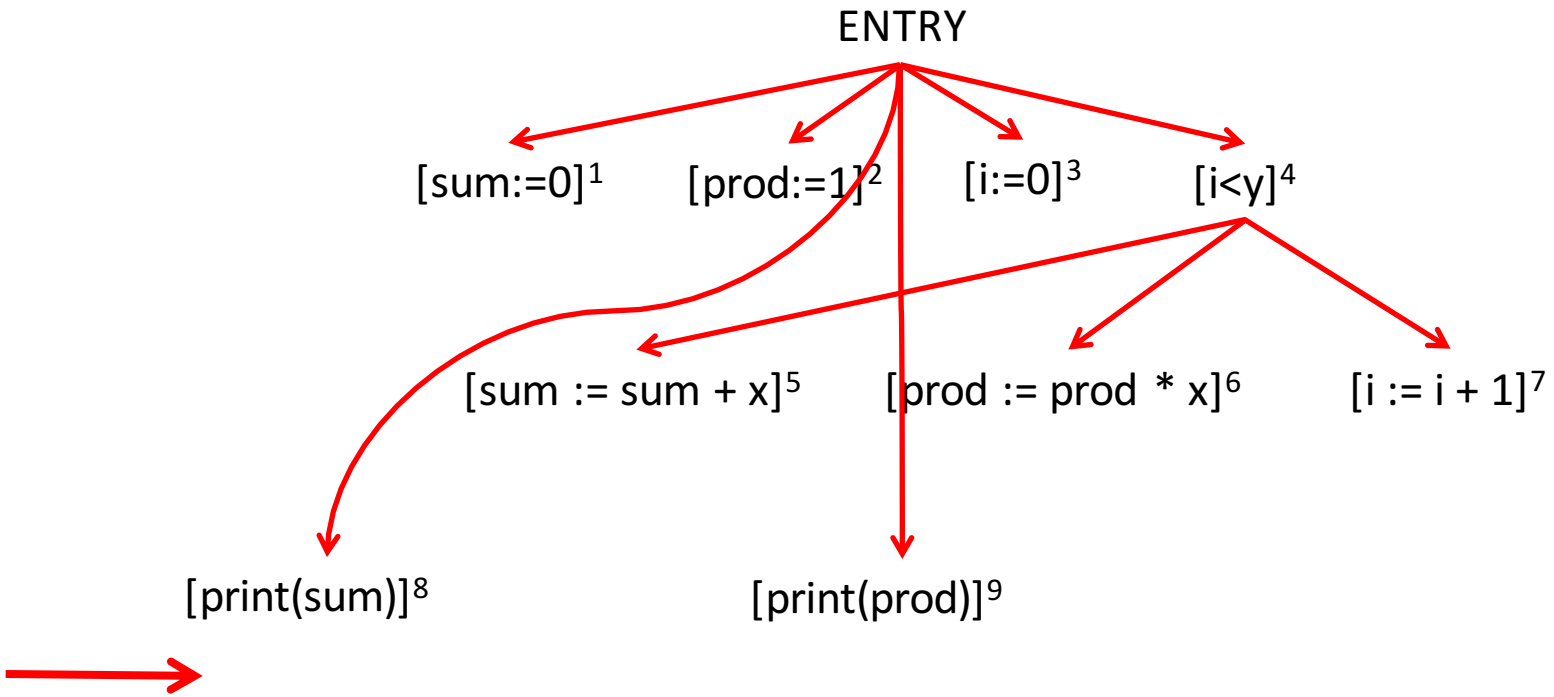


- - - - -> $\{(l, l') \mid l \in \bigcup_{\text{x used in block } l'} \mathbf{UD}(x, l') \text{ where } l' \text{ labels a block}\}$

(self-loops are omitted)

Example: Program dependence graph

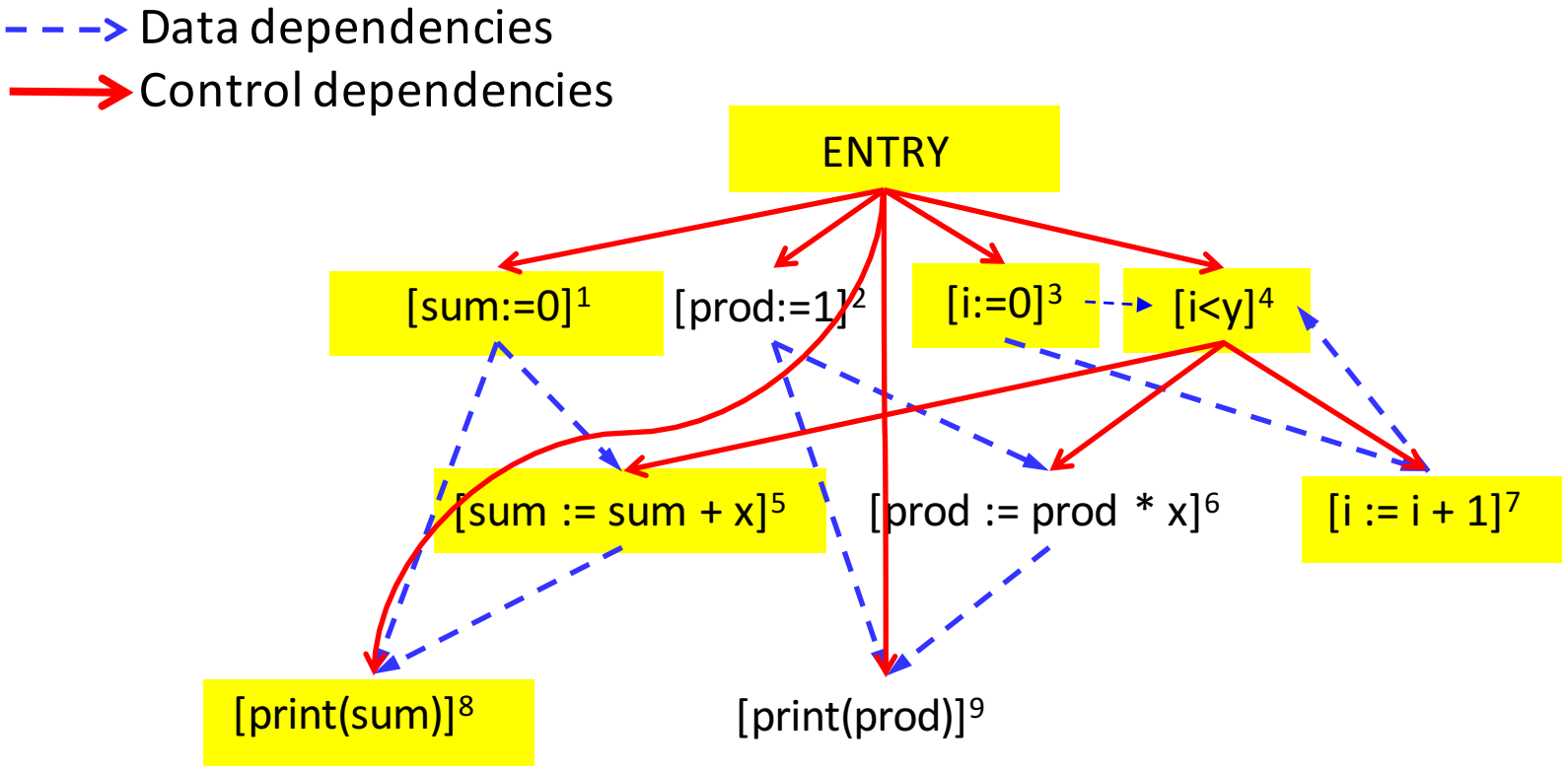
2. Control dependence subgraph



(1) Edge from special node ENTRY to any node not within any control structure (such as while, if-then-else)

(2) Edge from any guard of a control structure to any statement within the control structure

Example: Computing the program slice



Slicing using the PDG:

(1) Take as initial node the one given by the slicing criterion

(2) Include all nodes which the initial node transitively depends upon (use both data- and control-dependencies)

Data Flow Analysis

Existence of solutions

Fixed point solutions (recap)

- The equation system of the example defines the 14 sets

$$LV_{entry}(1),\ LV_{entry}(2),\ \ldots,\ LV_{exit}(7)$$

in terms of each other.

- When writing $\underline{LV}$ for the vector of these 14 sets, the equation system can be written as a function F where

$$\underline{LV} = F(\underline{LV})$$

- Using a vector of variables $\underline{X} = (X_1, \ldots, X_{14})$, the function can be defined as

$$F(\underline{X}) = (f_1(\underline{X}),\ \ldots,\ f_{14}(\underline{X}))$$

where for example

$$f_{11}(X_1,\ \ldots,\ X_{14}) = X_5 \cup X_6$$

- From the above equation it is clear that the solution $\underline{LV}$ we are looking for is the (least) fixed point of the function F.

Why are we interested in the least solution?

Remember the formulation of the goal of the Live Variables Analysis:

> "For each program point, which variables *may* be live at the exit from the point."

Clearly, larger solutions can always be accepted – even the set of all variables would do! – but the least ("smallest") solution is the most precise one.

Partially ordered sets (recap)

For any analysis, we are interested in expressing that one analysis result is "better" (more precise) than another.

In other words, we want the analysis domain to be partially ordered.

A partial ordering is a relation $\sqsubseteq$ that is

- reflexive: $\forall d : d \sqsubseteq d$
- transitive: $\forall d_1, d_2, d_3 : d_1 \sqsubseteq d_2$ and $d_2 \sqsubseteq d_3$ imply $d_1 \sqsubseteq d_3$
- anti-symmetric: $\forall d_1, d_2 : d_1 \sqsubseteq d_2$ and $d_2 \sqsubseteq d_1$ imply $d_1 = d_2$

A partially ordered set $(D, \sqsubseteq)$ is a set D with a partial ordering $\sqsubseteq$.

Examples: Real numbers $(\mathbf{R}, \leq)$, power sets $(\wp(S), \subseteq)$, ...

Existence of the fixed point

- To ensure that we can always obtain a result, we would like to know whether a fixed point of F always exists.

- To decide this, we need background on properties of:
  - the analysis domain used to represent the data flow information, i.e. in the case of the LV analysis the domain

    $\wp(Var_*)$, the power set of all variables occurring in the program
  - the function F, as defined before

Note:

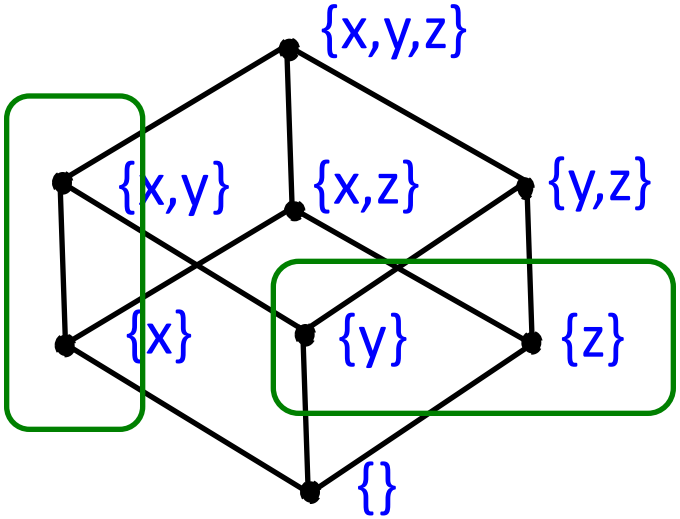
- Var: set of all variables
- Var_*: set of all variables *occurring in the given program*

Complete lattices

We are aiming for a specific kind of partially ordered set with even nicer properties: complete lattices.

- $d \in D$ is an upper bound of Y if $\forall d' \in Y : d' \sqsubseteq d$
- A least upper bound d of Y is an upper bound of Y that satisfies $d \sqsubseteq d_0$ whenever d_0 is another upper bound of Y
- A complete lattice is a partially ordered set $(D, \sqsubseteq)$ such that each subset Y has a least upper bound $\bigsqcup Y$ (and a greatest lower bound $\bigsqcap Y$)

Example: Power sets



Least upper bound operator, top, bottom

- The least upper bound (lub) operator $\bigsqcup : \wp(D) \to D$ (also: join operator) is used to combine analysis information from different paths.
- For example, in the case of the LV analysis, the join is given by ordinary set union $\cup$, and we were using it to combine information from both if-branches:

$$LV_{exit}(4) = LV_{entry}(5) \cup LV_{entry}(6)$$

- Every complete lattice has a least and a greatest element, they are called bottom $\bot$ and top $\top$, respectively.
- A complete lattice is never empty.

Tarski's Fixed Point Theorem

Monotone function

A function F : D -> D is called monotone over (D, $\sqsubseteq$) if

d $\sqsubseteq$ d' implies F(d) $\sqsubseteq$ F(d') for all d, d' $\in$ D

Fixed point

Assume F : D -> D. A value d $\in$ D such that F(d) = d is called a fixed point of F.

Tarski's Fixed Point Theorem

Let (D, $\sqsubseteq$) be a complete lattice and let F : D -> D be a monotone function. Then the set of all fixed points of F is a complete lattice with respect to $\sqsubseteq$.

In particular, F has a least and a greatest fixed point.

Existence of the least solution

- Using Tarski's fixed point theorem, we know that a least solution exists if
  - the function F describing the equation system is monotone
  - the analysis domain is a complete lattice
- In the case of the LV analysis these properties are easily checked:
  - To prove the monotonicity of F, we prove the monotonicity of each function f_i
  - The domain $\wp(Var_*)$ is trivially a complete lattice (it is a power set)

Equation solving (recap)

- How can we obtain the least fixed point practically?
- For the least element $\bot \in D$ of a partially ordered set D we have

$$\bot \sqsubseteq F(\bot)$$

- Since F is monotone, we have by induction for all $n \in \mathbf{N}$

$$F^n(\bot) \sqsubseteq F^{n+1}(\bot)$$

- All elements of the sequence are in the domain D, and therefore, if D is finite, there exists an $n \in \mathbf{N}$ such that

$$F^n(\bot) = F(F^n(\bot))$$

(Requires special properties of D and F, shown later.)

- But this means that $F^n(\bot)$ is a fixed point! (And indeed a least fixed point.)

Data Flow Analysis

Available Expressions Analysis

Available expressions analysis

Another example of a data flow analysis: available expressions (AE) analysis.

- The aim of the available expressions analysis is to determine

> "For each program point, which expressions *must* have already been computed, and not later modified, on all paths to the program point."

Example:

$[x:=a+b]^1$; $[y:=a*b]^2$;

while $[y>a+b]^3$ **do** $[a:=a+1]^4$; $[x:=a+b]^5$ **end**

Which expression is always available at the entry to 3?

Formalization: Data flow equations

The available expressions analysis can be specified following the scheme for LV analysis.

Analysis domain: $\wp(AExp_*)$, i.e. sets of arithmetic expressions.

$$AE_{entry}(l') = \bigcap_{(l,\, l') \in CFG} AE_{exit}(l)$$

(and $AE_{entry}(l) = \{\}$ if l is the initial label)

$$AE_{exit}(l) = (AE_{entry}(l) \setminus kill_{AE}(l)) \cup gen_{AE}(l)$$

$kill_{AE}([x:=a]^l)$ = {all expressions containing x}

$kill_{AE}([b]^l)$ = {}

$gen_{AE}([x:=a]^l)$ = {all subexpressions of *a* not containing x}

$gen_{AE}([b]^l)$ = {all subexpressions of *b*}

Application: Common subexpression elimination

Goal: Find computations that are always performed at least twice on a given execution path and then eliminate the second and later occurrences.

Example:

$[x:=a+b]^1$; $[y:=a*b]^2$;

while $[y>a+b]^3$ **do** $[a:=a+1]^4$; $[x:=a+b]^5$ **end**

is transformed into

$[u:=a+b]$; $[x:=u]^1$; $[y:=a*b]^2$;

while $[y>u]^3$ **do** $[a:=a+1]^4$; $[u:=a+b]$; $[x:=u]^5$ **end**

Differences of the LV, RD, and AE analyses

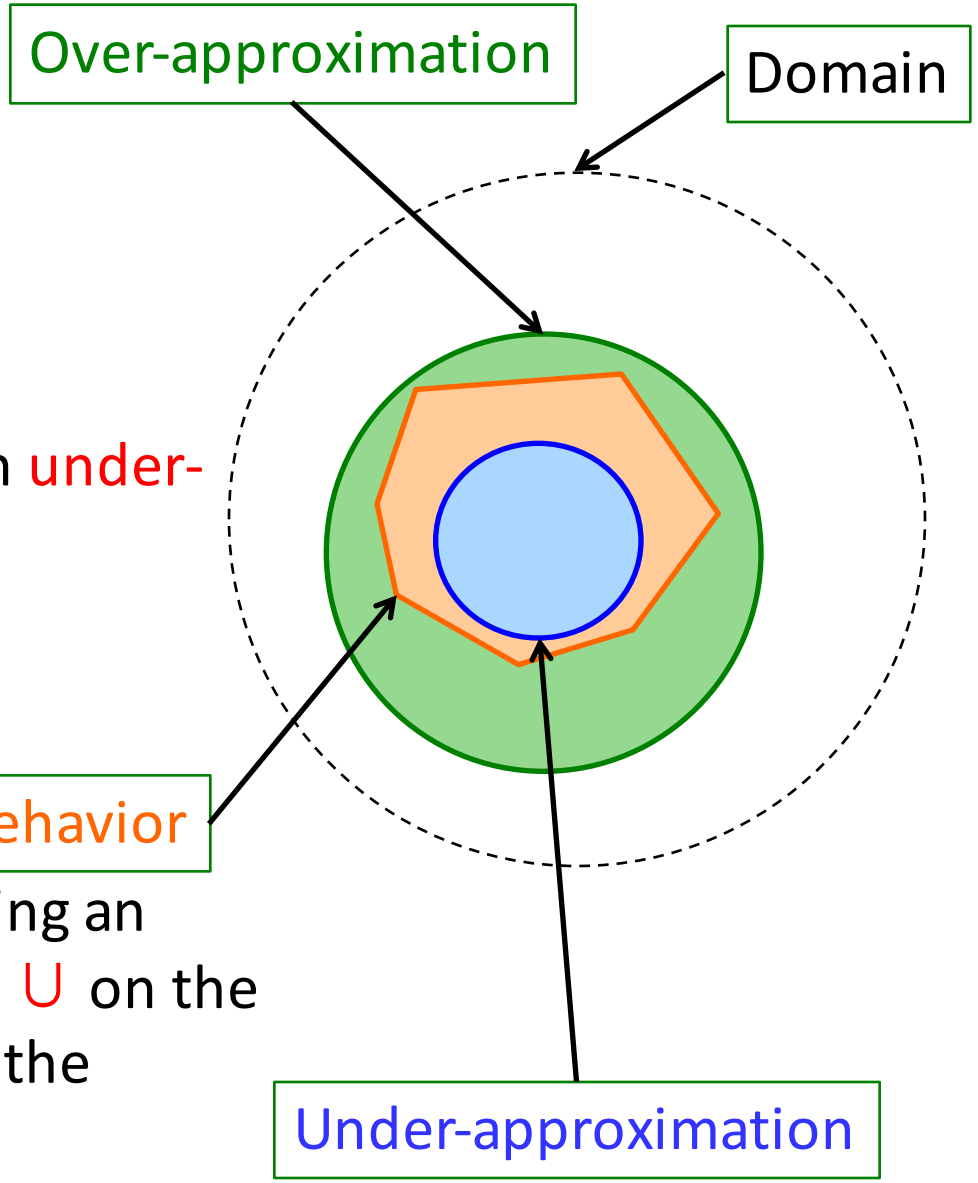
The flow has been reversed:

- LV: backward analysis
- AE, RD: forward analysis

Also, we are now interested in an under-approximation ("must"):

- LV, RD: may analysis
- AE: must analysis

For that reason, in AE we are taking an intersection ∩ instead of a union ∪ on the paths. We are then interested in the greatest solution.



Data Flow Analysis

Bit Vector Frameworks

Four classical analyses

Live Variables

Variables that may be live at a program point.

Reaching Definitions

Assignments that may have been made and not overwritten along some path to a program point.

Available Expressions

Expressions that must have already been computed and not later modified on all paths to a program point.

Very Busy Expressions

Expressions that must be very busy at a program point.

A general schema

The four classical analyses, and many more data flow analyses follow a general schema.

- The analysis domain is always a power set of some finite set, e.g. sets of variables in case of LV.
- The functions that specify how data is propagated through elementary blocks (so-called transfer functions) are all of the form

$$f(d) = (d \setminus kill) \cup gen$$

(It's easy to prove that functions of this form are monotone.)

Bit Vector Frameworks

These properties of classical analyses make for efficient implementation using bit vectors to represent sets.

Example:

LV analysis for a program with variables x, y, z

- Representation:

  {} = 000, {x} = 100, {y} = 010, ..., {x, z} = 101

- Join is very efficient (use boolean **or**):

  {x, y} ∪ {x, z} = {x, y, z}

  110 **or** 101 = 111