

# Applied Static Analysis

---

## Data Flow Analysis

---

Software Technology Group  
Department of Computer Science  
Technische Universität Darmstadt  
[Dr. Michael Eichberg](#)

For background information see:

- Principles of Program Analysis; Flemming Nielson, Hanne Riis Nielson, and Chris Hankin; Springer, 2005
  - [Static Program Analysis](#); Anders Møller, and Michael I. Schwartzbach; May 27, 2015
-

# Lattice Theory

---

Many static analyses are based on the mathematical theory of lattices.

The lattice puts the facts (often, but not always, sets) computed by an analysis in a well-defined partial order.

Analyses are often **well-defined** functions over lattices and can then be combined and reasoned about.

---

# Example: Sign Analysis

---

- Let's assume that we want to compute the sign of an integer value. The analysis should only return the information is definite. I.e.,
- Instead of computing with concrete values, our analysis performs it computations using abstract values:
  - positive (+)
  - negative (-)
  - zero
- Additionally, we have to add an abstract value  $T$  that represents the fact that we don't know the sign of the value.
- Values that are not initialized are represented using  $\perp$ .

$T$  is called *top*. (The least precise information. *The sound over-approximation.*)

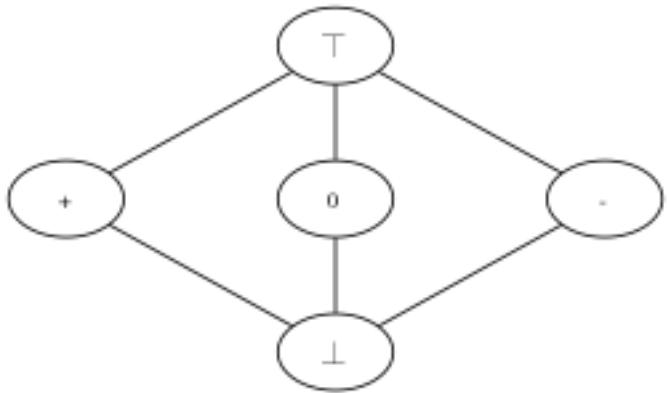
$\perp$  is called *bottom*. (The most precise information.)

---

# Example: Sign Analysis - the lattice

---

The lattice for the previous domain is:



The ordering reflects that  $\top$  reflects all types of integer values.

---

# Example: Sign Analysis - example program

---

```
def select(c : Boolean): Int = {  
    val a = 42  
    val b = 333  
    var x = 0;  
    if (c)  
        x = a + b;  
    else  
        x = a - b;  
    x  
}
```

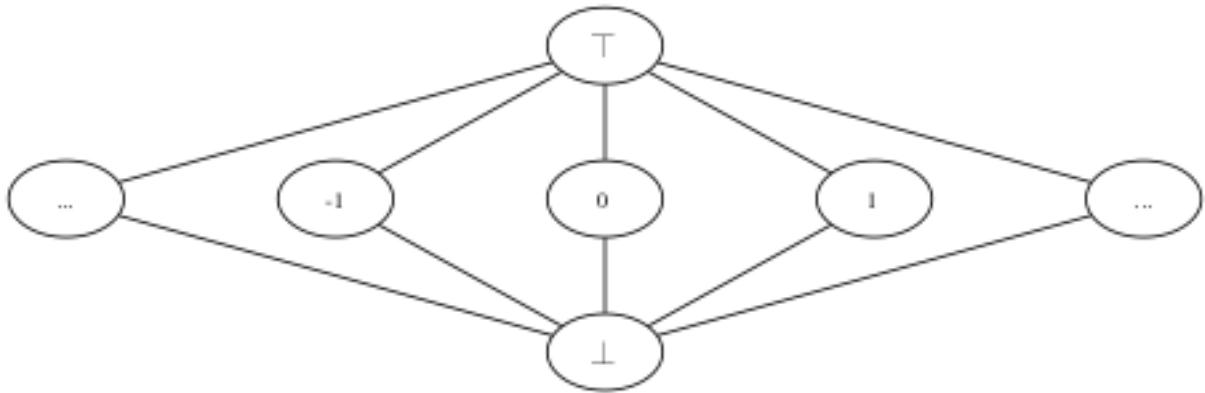
A possible result of the analysis could be that `a` and `b` are always positive; `x` is either positive or negative (or zero) ( $\top$ ).

---

# Example: Constant Propagation - the lattice

---

The lattice would be:



The ordering reflects that T reflects that the value is any.

Note that this lattice is not finite, but has finite height

Again  $\perp$  denotes uninitialized (most precise) values and T denotes that the value is not constant.

---

# Example: Constant Propagation - example program

---

```
val z = 3
var x = 1
while(x > 0) {
    if(x == 1) {
        y = 7
    } else {
        y = z + 4
    }
    x = 3
}
```

---

A possible result of the analysis could be that `y` is always 7 (and `z` 3.)

---

# Partial Orderings

---

- a partial ordering is a relation  $\sqsubseteq: L \times L \rightarrow \{\text{true}, \text{false}\}$ , which
  - is reflexiv:  $\forall l : l \sqsubseteq l$
  - is transitive:  $\forall l_1, l_2, l_3 : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_3 \Rightarrow l_1 \sqsubseteq l_3$
  - is anti-symmetric:  $\forall l_1, l_2 : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_1 \Rightarrow l_1 = l_2$
- a partially ordered set  $(L, \sqsubseteq)$  is a set  $L$  equipped with a partial ordering  $\sqsubseteq$

When  $x \sqsubseteq y$  we say  $x$  is at least as precise as  $y$  or  $y$  over-approximates  $x/y$  is an over-approximation of  $y$ .

---

# Upper Bounds

---

- for  $Y \subseteq L$  and  $l \in L$ 
  - $l$  is an upper bound of  $Y$ , if  $\forall l' \in Y : l' \sqsubseteq l$
  - $l$  is a **least upper bound** of  $Y$ , if  $l \sqsubseteq l_0$  whenever  $l_0$  is also an upper bound of  $Y$
- if a least upper bound exists, it is unique ( $\sqsubseteq$  is anti-symmetric)
- the least upper bound of  $Y$  is denoted  $\sqcup Y$   
we write:  $l_1 \sqcup l_2$  for  $\sqcup\{l_1, l_2\}$

$\sqcup$  is also called the join operator.

---

# Lower Bounds

---

- for  $Y \subseteq L$  and  $l \in L$ 
  - $l$  is a lower bound of  $Y$ , if  $\forall l' \in Y : l \sqsubseteq l'$
  - $l$  is a **greatest lower bound** of  $Y$ , if  $l_0 \sqsubseteq l$  whenever  $l_0$  is also a lower bound of  $Y$
- if a greatest lower bound exists, it is unique ( $\sqsubseteq$  is anti-symmetric)
- the greatest lower bound of  $Y$  is denoted  $\sqcap Y$   
we write:  $l1 \sqcap l2$  for  $\sqcap \{l1, l2\}$

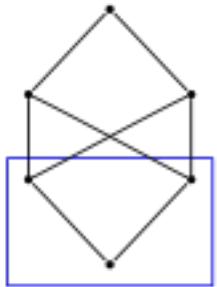
$\sqcap$  is also called the meet operator.

---

# Upper/Lower Bounds

---

A subset  $Y$  of a partially ordered set  $L$  need not have least upper or greatest lower bounds.



Here, the subset is depicted in blue.

---

# (complete) Lattice

---

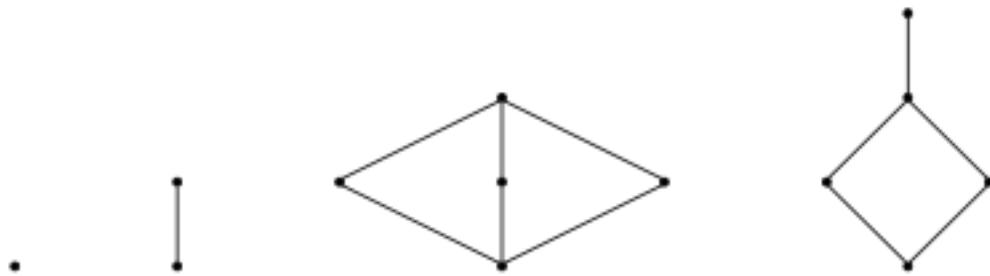
- complete Lattice  $\mathbf{L} = (\mathbf{L}, \sqsubseteq, \sqcap, \sqcup, \top, \perp)$
- is a partially ordered set  $(\mathbf{L}, \sqsubseteq)$  such that each subset  $\mathbf{Y}$  has a greatest lower bound and a least upper bound.
  - $\perp = \sqcup \emptyset = \sqcap \mathbf{L}$
  - $\top = \sqcap \emptyset = \sqcup \mathbf{L}$

A lattice must have a unique largest element  $\top$  and a unique smallest element  $\perp$ .

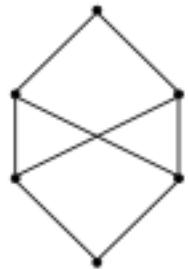
The least upper bound (the greatest lower bound) of a set  $\mathbf{Y} \subseteq \mathbf{L}$  is always an element of  $\mathbf{L}$  but not necessarily an element of  $\mathbf{Y}$ .

---

**Valid lattices:**



**No lattice:**



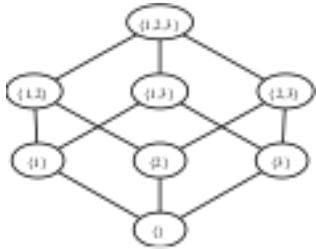
Do ask yourself why the lower diagram is not a lattice?

---

# (complete) Lattice - example

---

Example  $(\mathcal{P}(S), \subseteq)$ ,  $S = \{1, 2, 3\}$



The above diagram is called a *Hasse diagram*.

Every finite set  $S$  defines a lattice  $(\mathcal{P}S, \subseteq)$  where  $\perp = \emptyset$  and  $\top = S$ ,  $x \sqcup y = x \cup y$ , and  $x \sqcap y = x \cap y$

A Hasse diagram is a graphical representation of the relation of elements of a partially ordered set.

---

# Height of a lattice

---

The length of the longest path from  $\perp$  to  $\top$ .

In general, the powerset lattice has height  $|S|$ .

The height of the previous lattice is 3.

---

# Closure Properties

---

Construction of complete lattices:

If  $L_1, L_2, \dots, L_n$  are lattices with finite height, then so is the (cartesian) product:

$$L_1 \times L_2 \times \cdots \times L_n = \\ (x_1, x_2, \dots, x_n) | X_i \in L_i$$

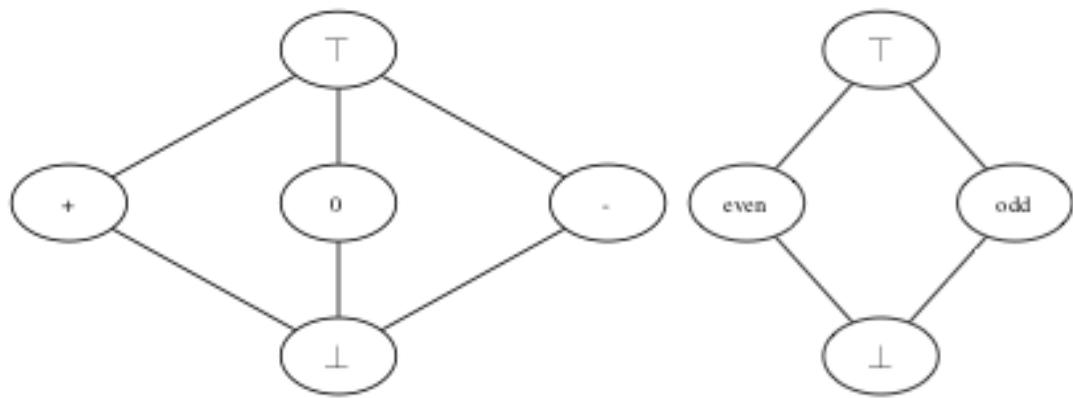
$$\text{height}(L_1 \times \cdots \times L_n) = \text{height}(L_1) + \cdots + \text{height}(L_n)$$

$\sqsubseteq$  is defined pointwise (i.e.,  $(l_{11}, l_{21}) \sqsubseteq (l_{12}, l_{22})$  iff  $l_{11} \sqsubseteq_1 l_{21} \wedge l_{11} \sqsubseteq_2 l_{21}$ ) and  $\sqcup$  and  $\sqcap$  can be computed pointwise.

---

# Two basic domains

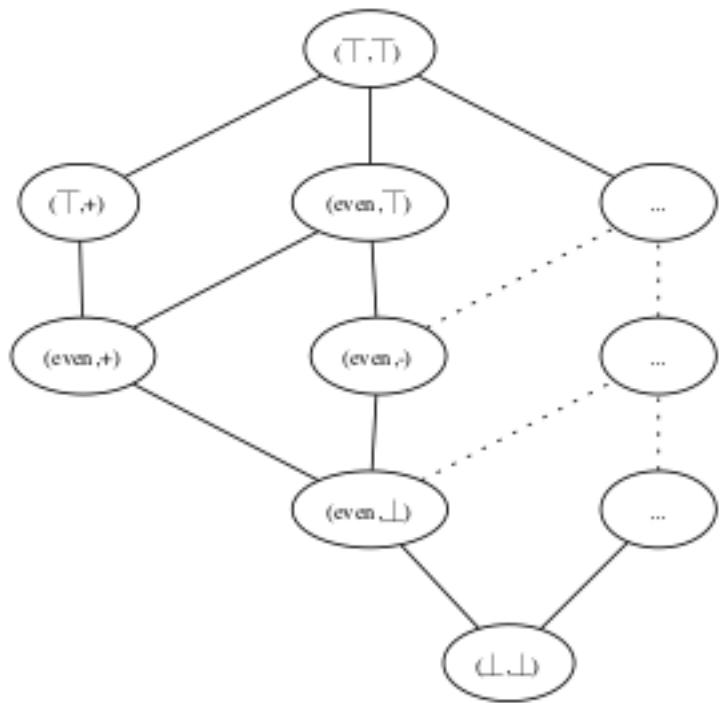
---



# Creating the cross-product

---

Creating the cross product of the sign and even-odd lattices.



# Properties of Functions

---

A function  $f : L_1 \rightarrow L_2$  between partially ordered sets is **monotone** if:

$$\forall l, l' \in L_1 : l \sqsubseteq_1 l' \Rightarrow f(l) \sqsubseteq_2 f(l')$$

The composition of monotone functions is monotone. However, being monotone does not imply being extensive ( $\forall l \in L : l \sqsubseteq f(l)$ ). A function that maps all values to  $\perp$  is clearly monotone, but not extensive.

The function  $f$  is **distributiv** if:

$$\forall l_1, l_2 \in L_1 : f(l_1 \sqcup l_2) = f(l_1) \sqcup f(l_2)$$

---

# Chains

---

A subset  $\mathbf{Y} \subseteq \mathbf{L}$  of a partially ordered set  $\mathbf{L} = (\mathbf{L}, \sqsubseteq)$  is a chain if

$$\forall l_1, l_2 \in Y : (l_1 \sqsubseteq l_2) \vee (l_2 \sqsubseteq l_1)$$

That is, the values  $l_1$  and  $l_2$  are comparable; a chain is a possibly empty subset of  $\mathbf{L}$  that is totally ordered.

The chain is finite if  $\mathbf{Y}$  is a finite subset of  $\mathbf{L}$ .

A sequence  $(l_n)_n = (l_n)_{n \in N}$  of elements in  $\mathbf{L}$  is an ascending chain if

$$n \leq m \Rightarrow l_n \sqsubseteq l_m$$

A descending chain is defined accordingly.

A sequence  $(l_n)_n$  eventually stabilizes iff  $\exists n_0 \in N : \forall n \in N : n \geq n_0 \Rightarrow l_n = l_{n_0}$

---

# Interval analysis - example

---

Let's compute the range of values that an integer variable can assume at runtime.

```
var x = 0
while (true) {
    x = x + 1
    println(x)
}
```

---

Ask yourself how the lattice would look like?

# Ascending/Descending Chain Condition

---

- A partially ordered set  $L$  satisfies the Ascending Chain Condition if and only if all ascending chains eventually stabilize.
- A partially ordered set  $L$  satisfies the Descending Chain Condition if and only if all descending chains eventually stabilize.

(A lattice must not be finite to satisfy the ascending chain condition).

---

# Fixed Point

---

- $l \in L$  is a fixed point for  $f$  if  $f(l) = l$
  - A least fixed point  $l_1 \in L$  for  $f$  is a fixed point for  $f$  where  $l_1 \sqsubseteq l_2$  for every fixed point  $l_2 \in L$  for  $f$ .
-

# Equation system

---

$$x_1 = F_1(x_1, \dots, x_n)$$

⋮

$$x_n = F_n(x_1, \dots, x_n)$$

where  $x_i$  are variables and  $F_i : L^n \rightarrow L$  is a collection of functions. If all functions are monotone then the system has a unique least solution which is obtained as the least-fixed point of the function  $F : L^n \rightarrow L^n$  defined by:

$$F(x_1, \dots, x_n) = (F_1(x_1, \dots, x_n), \dots, F_n(x_1, \dots, x_n))$$

In a lattice  $L$  with finite height, every monotone function  $f$  has a unique least fixed point given by:

$$\text{fix}(f) = \bigcup_{i \geq 0} f^i(\perp).$$

---

# Data Flow analysis: Available Expressions

Determine for each program point, which expressions must have already been computed and not later modified on all paths to the program point.

The following discussion of data-flow analyses uses the more common equational approach.

## Available Expressions - Example

```
def m(initialA: Int, b: Int): Int = {  
  /*pc 0*/  var a = initialA  
  /*pc 1*/  var x = a + b;  
  /*pc 2*/  val y = a * b;  
  /*pc 3*/  while (y > a + b) {  
    /*pc 4*/    a = a + 1  
    /*pc 5*/    x = a + b  
  }  
  /*pc 6*/  a + x  
}
```

The expression `a + b` is available every time execution reaches the test ( pc 4 ).

## Available Expressions - gen/kill functions

- An **expression is killed** in a block if any of the variables used in the (arithmetic) expression are modified in the block. The function  $\text{kill} : \text{Block} \rightarrow \mathcal{P}(\text{ArithExp})$  produces the set of killed arithmetic expressions.
- A **generated expression** is a non-trivial (arithmetic) expression that is evaluated in the block and where none of the variables used in the expression are later modified in the block. The function  $\text{gen} : \text{Block} \rightarrow \mathcal{P}(\text{ArithExp})$  produces the set of generated expressions.

Typically, a block is a single statement in a program's three-address code.

The underlying lattice is the powerset of all expression of the program.

If you know that a function is pure or a field is (effectively) final it is directly possible to also take these expressions into consideration.

## Available Expressions - data flow equations

Let  $S$  be our program and  $flow$  be a flow in the program between two statements  $(pc_i, pc_j)$ .

$$AE_{entry}(pc_i) = \begin{cases} \emptyset & \text{if } i = 0 \\ \bigcap AE_{exit}(pc_h) | (pc_h, pc_i) \in flow(S) & \text{otherwise} \end{cases}$$

$$AE_{exit}(pc_i) = (AE_{entry}(pc_i) \setminus kill(block(pc_i)) \cup gen(block(pc_i)))$$

## Available Expressions - Example continued I

```
/*pc 1*/ var x = a + b;
/*pc 2*/ val y = a * b;
/*pc 3*/ while (y > a + b) {
/*pc 4*/     a = a + 1
/*pc 5*/     x = a + b
    }
```

The kill/gen functions:

pc	kill	gen
1	$\emptyset$	$\{a + b\}$
2	$\emptyset$	$\{a * b\}$
3	$\emptyset$	$\{a + b\}$
4	$\{a + b, a * b, a + 1\}$	$\emptyset$
5	$\emptyset$	$\{a + b\}$

We get the following equations:

$$AE_{entry}(pc_1) = \emptyset$$

$$AE_{entry}(pc_2) = AE_{exit}(pc_1)$$

$$AE_{entry}(pc_3) = AE_{exit}(pc_2) \cap AE_{exit}(pc_5)$$

$$AE_{entry}(pc_4) = AE_{exit}(pc_3)$$

$$AE_{entry}(pc_5) = AE_{exit}(pc_4)$$

$$AE_{exit}(pc_1) = AE_{entry}(pc_1) \cup \{a + b\}$$

$$AE_{exit}(pc_2) = AE_{entry}(pc_2) \cup \{a * b\}$$

$$AE_{exit}(pc_3) = AE_{entry}(pc_3) \cup \{a + b\}$$

$$AE_{exit}(pc_4) = AE_{entry}(pc_4) \setminus \{a + b, a * b, a + 1\}$$

$$AE_{exit}(pc_5) = AE_{entry}(pc_5) \cup \{a + b\}$$

# Data Flow Analysis: Naive algorithm

---

to solve the combined function:  $F(x_1, \dots, x_n) = (F_1(x_1, \dots, x_n), \dots, F_n(x_1, \dots, x_n))$ :

This is just a conceptual algorithm which is not to be implemented:

$x = (\perp, \dots, \perp); do\{ t = x; x = F(x); \}while(x \neq t);$

---

# Data Flow Analysis: Chaotic Iteration

We exploit the fact that our lattice has the structure  $L^n$  to compute the solution  $(x_1, \dots, x_n)$ :

$x_1 = \perp; \dots; x_n = \perp; \text{while}(\exists i : x_i \neq F_i(x_1, \dots, x_n))\{ x_i = F_i(x_1, \dots, x_n); \}$

## Available Expressions - Example continued II

```
/*pc 1*/ var x = a + b;
/*pc 2*/ val y = a * b;
/*pc 3*/ while (y > a + b) {
/*pc 4*/     a = a + 1
/*pc 5*/     x = a + b
}
```

Solution:

pc	$AE_{entry}$	$AE_{exit}$
1	$\emptyset$	$\{a + b\}$
2	$\{a + b\}$	$\{a + b, a * b\}$
3	$\{a + b\}$	$\{a + b\}$
4	$\{a + b\}$	$\emptyset$
5	$\emptyset$	$\{a + b\}$

# Data Flow Analysis: Reaching Definitions

---

For each program point, which assignments *may* have been made and not overwritten, when program execution reaches this point along some path.

Reaching definitions is (also) a forward analysis.

Reaching definitions analysis doesn't make sense, if your code representation is in SSA form!

(Reaching definitions analysis is typically used to construct so called *du* (definition → use) and *ud* (use → definition) chains.

---

# Reaching Definitions - Example

```
def m(): Int = {  
    /*pc 1*/ var x = 5;  
    /*pc 2*/ var y = 1;  
    /*pc 3*/ while (x > 1) {  
        /*pc 4*/     y = x * y  
        /*pc 5*/     x = x - 1  
    }  
    /*pc 6*/     x + y  
}
```

All assignments reach pc 4 ; only the assignments at pc 1, 4 and 5 reach pc 5.

The underlying lattice is defined over the powerset of all variables cross definition sites ( DefSite ) ( $\mathcal{P}(\text{Var} \times \text{DefSite})$ ); a DefSite is typically identified by the program counter of the assignment statement. For parameters to a method multiple solutions are possible: either explicit initialization statements are added which make the parameters explicitly available in the method body or special values which cannot be confused with program counters are used.

## Reaching Definitions - gen/kill functions

- An assignment is destroyed if the block assigns a new value to the variable (the left-hand side of an assignment.)
- An assignment generates definitions.

If the code that you analyze may contain uninitialized variables then you have to extend your lattice to cover uninitialized values.

## Reaching Definitions - data flow equations

Let  $S$  be our program and  $flow$  be a forward flow in the program between two statements  $(pc_i, pc_j)$ .

$$RD_{entry}(pc_i) = \begin{cases} \emptyset & \text{if } i = 0 \\ \bigcup RD_{exit}(pc_h) | (pc_h, pc_i) \in flow(S) & \text{otherwise} \end{cases}$$

$$RD_{exit}(pc_i) = (RD_{entry}(pc_i) \setminus kill(block(pc_i))) \cup gen(block(pc_i)))$$

If the analyzed code may contain uninitialized variables then the case that handles `i == 0` should initialize the location where the variables are initialized to some special place!

---

## Data Flow Analysis: Worklist Algorithm

---

Data flow problems can more efficiently be solved using a worklist algorithm that, when a node is updated, we only consider those other nodes which depend on it.

Let's define a helper function which determines for a node  $v \in V$  the set of nodes on which the node depends.  $\text{dep} : V \rightarrow \mathcal{P}(V)$

Let  $W$  be a worklist:

```
x1 = ⊥; . . . ; xn = ⊥;  
W = {v1, . . . , vn};  
while(w ≠ ∅){  
    i = W.removeNext();  
    y = Fi(x1, . . . , xn);  
    if(xi = y){  
        for(vj ∈ dep(vi)) W.add(vj);  
        xi = y;  
    }  
}
```

---

## Reaching Definitions - example continued I

---

```
def m(): Int = {  
/*pc 1*/  var x = 5;  
/*pc 2*/  var y = 1;  
/*pc 3*/  while (x > 1) {  
/*pc 4*/    y = x * y  
/*pc 5*/    x = x - 1  
    }  
/*pc 6*/  x + y  
}
```

The kill/gen functions:

pc	kill	gen
1	$\{(x, 1), (x, 5)\}$	$\{(x, 1)\}$
2	$\{(y, 2), (y, 4)\}$	$\{(y, 2)\}$
3	$\emptyset$	$\emptyset$
4	$\{(y, 2), (y, 4)\}$	$\{(y, 4)\}$
5	$\{(x, 1), (x, 5)\}$	$\{(x, 5)\}$

We get the following equations:

$$\begin{aligned}
 RD_{entry}(pc_1) &= \emptyset \\
 RD_{entry}(pc_2) &= RD_{exit}(pc_1) \\
 RD_{entry}(pc_3) &= RD_{exit}(pc_2) \cup RD_{exit}(pc_5) \\
 RD_{entry}(pc_4) &= RD_{exit}(pc_3) \\
 RD_{entry}(pc_5) &= RD_{exit}(pc_4) \\
 RD_{exit}(pc_1) &= (RD_{entry}(pc_1) \setminus \{(x, 1), (x, 5)\}) \cup \{(x, 1)\} \\
 RD_{exit}(pc_2) &= (RD_{entry}(pc_2) \setminus \{(y, 2), (y, 4)\}) \cup \{(y, 2)\} \\
 RD_{exit}(pc_3) &= RD_{entry}(pc_3) \\
 RD_{exit}(pc_4) &= (RD_{entry}(pc_4) \setminus \{(y, 2), (y, 4)\}) \cup \{(y, 4)\} \\
 RD_{exit}(pc_5) &= (RD_{entry}(pc_5) \setminus \{(x, 1), (x, 5)\}) \cup \{(x, 5)\}
 \end{aligned}$$


---

## Reaching Definitions - example continued II

Solution:

pc	$RD_{entry}$	$RD_{exit}$
1	$\emptyset$	$\{(x, 1)\}$
2	$\{(x, 1)\}$	$\{(x, 1), (y, 2)\}$
3	$\{(x, 1), (y, 2), (y, 4), (x, 5)\}$	$\{(x, 1), (y, 2), (y, 4), (x, 5)\}$
4	$\{(x, 1), (y, 2), (y, 4), (x, 5)\}$	$\{\{(x, 1), (y, 4), (x, 5)\}\}$
5	$\{(x, 1), (y, 4), (x, 5)\}$	$\{(y, 4), (x, 5)\}$

---

# Data Flow Analysis: Very Busy Expressions Analysis

---

For each program point, which expressions must be very busy at the exit from the point.  
An expression is very busy at the exit of a block if – independent of the taken path – the expression must always be used before any of the variables occurring in it are redefined.

Very Busy Expressions Analysis is a backward analysis.

---

# Very Busy Expressions Analysis - Example

```
def m(a: Int, b: Int): Int = {
/*pc 0*/  var x, y = 0
/*pc 1*/  if(a > b) {
/*pc 2*/    x = b - a
/*pc 3*/    y = a - b
  else {
/*pc 4*/    y = b - a
/*pc 5*/    x = a - b
  }
/*pc 6*/  n(x, y)
}
```

The expression `b-a` and `a-b` are both very busy at `pc1`.

## Very Busy Expressions Analysis - gen/kill functions

- An assignment is destroyed if the block assigns a new value to the variable (the left-hand side - An **expression is killed** in a block if any of the variables used in the (arithmetic) expression are modified in the block. The function  $kill : Block \rightarrow \mathcal{P}(ArithExp)$  produces the set of killed arithmetic expressions.
- A **generated expression** is a non-trivial (arithmetic) expression that is evaluated in the block. The function  $gen : Block \rightarrow \mathcal{P}(ArithExp)$  produces the set of generated expressions.

## Very Busy Expressions Analysis - data flow equations

Let  $S$  be our program,  $flow^R$  be a backward flow in the program between two statements  $(pc_i, pc_j)$  and  $final$  be a function that returns the set of nodes of the program which terminate the program.

$$VB_{exit}(pc_i) = \begin{cases} \emptyset & \text{if } i \in final(S) \\ \bigcap_{VB_{entry}(pc_h) | (pc_h, pc_i) \in flow^R(S)} VB_{entry}(pc_h) & \text{otherwise} \end{cases}$$

$$VB_{entry}(pc_i) = (VB_{exit}(pc_i) \setminus kill(block(pc_i))) \cup gen(block(pc_i)))$$

The result of the analysis depends on the set of nodes which are considered to terminate the method. E.g., an expression such as `a/b` may lead to an abnormal return of a method.

## Very Busy Expressions Analysis - example continued I

```
def m(a: Int, b: Int): Int = {  
  /*pc 0*/ var x, y = 0  
  /*pc 1*/ if(a > b) {  
    /*pc 2*/   x = b - a  
    /*pc 3*/   y = a - b  
    else {  
      /*pc 4*/     y = b - a  
      /*pc 5*/     x = a - b  
    }  
  /*pc 6*/  n(x, y)  
}
```

The kill/gen functions:

pc	kill	gen
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{b - a\}$
3	$\emptyset$	$\{a - b\}$
4	$\emptyset$	$\{b - a\}$
5	$\emptyset$	$\{a - b\}$

We get the following equations:

$$\begin{aligned} VB_{entry}(pc_1) &= VB_{exit}(pc_1) \\ VB_{entry}(pc_2) &= VB_{exit}(pc_2) \cup \{b - a\} \\ VB_{entry}(pc_3) &= \{a - b\} \\ VB_{entry}(pc_4) &= VB_{exit}(pc_4) \cup \{b - a\} \\ VB_{entry}(pc_5) &= \{a - b\} \\ VB_{exit}(pc_1) &= VB_{entry}(pc_2) \cap VB_{entry}(pc_4) \\ VB_{exit}(pc_2) &= VB_{entry}(pc_3) \\ VB_{exit}(pc_3) &= \emptyset \\ VB_{exit}(pc_4) &= VB_{entry}(pc_5) \\ VB_{exit}(pc_5) &= \emptyset \end{aligned}$$

## Very Busy Expressions Analysis - example continued II

Solution:

$pc$	$VB_{entry}$	$VB_{exit}$
1	$\{a - b, b - a\}$	$\{a - b, b - a\}$
2	$\{a - b, b - a\}$	$\{a - b\}$
3	$\{a - b\}$	$\emptyset$
4	$\{a - b, b - a\}$	$\{a - b\}$
5	$\{a - b\}$	$\emptyset$

## Very Busy Expressions Analysis - applied

Given the following scala program. Which arithmetic expressions are very busy at which program point and what are the challenges?

```
def m(b1 : Boolean, b2 : Boolean, a: Int, b : Int ): Int = {
  if(b1){
    m()
    a/b
  } else {
    if(b2) {
      val c = a + b
      a/b * c
    } else {
      val c = a * b
      a/b - c
    }
  }
}
```

# Data Flow Analysis: Live Variable Analysis

---

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

A variable is live at the exit from a block if there exists a path to the use of the variable that does not re-define the variable.

Live Variable Analysis is a backward analysis.

---

# Live Variable Analysis - example

```
def m(): Int = {  
    /*pc 1*/ var x = 2  
    /*pc 2*/ var y = 4  
    /*pc 3*/ x = 1  
    /*pc 4*/ val z = if( y > x ) {  
        /*pc 5*/ y  
        /*pc 6*/ } else {  
            /*pc 7*/ y * y  
        }  
    /*pc 8*/ x = z + 1  
}
```

The variable `x` @ pc 1 is not live at exit of pc 1 ; both: `x` and `y` are live at the exit of pc 3 .

## Live Variable Analysis - gen/kill functions

- The variable that appears on the left hand side of an assignment is killed. The function  $kill : Block \rightarrow \mathcal{P}(Var)$  produces the set of killed variables.
- All variables that appear in a block are **generated variables**. The function  $gen : Block \rightarrow \mathcal{P}(Var)$  produces the set of generated variavbles.

## Live Variable Analysis - data flow equations

Let  $S$  be our program,  $flow^R$  be a backward flow in the program between two statements  $(pc_i, pc_j)$  and  $final$  be a function that returns the set of nodes of the program which terminate the program.

$$LV_{exit}(pc_i) = \begin{cases} \emptyset & \text{if } i \in final(S) \\ \bigcup_{pc_h} LV_{entry}(pc_h) | (pc_h, pc_i) \in flow^R(S) & \text{otherwise} \end{cases}$$

$$LV_{entry}(pc_i) = (LV_{exit}(pc_i) \setminus kill(block(pc_i))) \cup gen(block(pc_i)))$$

# Live Variable Analysis - example continued I

```

def m(): Int = {
/*pc 1*/ var x = 2
/*pc 2*/ var y = 4
/*pc 3*/ x = 1
/*pc 4*/ val z = if( y > x ) {
/*pc 5*/   val z =*/ y
            } else {
/*pc 6*/   val z =*/ y * y
            }
/*pc 7*/ x = z + 1
}

```

In the following the assignment `val z = if(c) {v1} else {v2}` is considered to be two assignments which are results of the two branches!

The kill/gen functions:

pc	kill	gen
1	{}	$\emptyset$
2	{}	$\emptyset$
3	{}	$\emptyset$
4	$\emptyset$	$\{x, y\}$
5	$\{z\}$	$\{y\}$
6	$\{z\}$	$\{y\}$
7	$\{x\}$	$\{z\}$

We get the following equations:

$$\begin{aligned}
LV_{entry}(pc_1) &= LV_{exit}(pc_1) \setminus \{x\} \\
LV_{entry}(pc_2) &= LV_{exit}(pc_2) \setminus \{y\} \\
LV_{entry}(pc_3) &= LV_{exit}(pc_3) \setminus \{x\} \\
LV_{entry}(pc_4) &= LV_{exit}(pc_4) \cup \{x, y\} \\
LV_{entry}(pc_5) &= (LV_{exit}(pc_5) \setminus \{z\}) \cup \{y\} \\
LV_{entry}(pc_6) &= (LV_{exit}(pc_6) \setminus \{z\}) \cup \{y\} \\
LV_{entry}(pc_7) &= \{z\} \\
LV_{exit}(pc_1) &= LV_{entry}(pc_2) \\
LV_{exit}(pc_2) &= LV_{entry}(pc_3) \\
LV_{exit}(pc_3) &= LV_{entry}(pc_4) \\
LV_{exit}(pc_4) &= LV_{entry}(pc_5) \cup LV_{entry}(pc_6) \\
LV_{exit}(pc_5) &= LV_{entry}(pc_7) \\
LV_{exit}(pc_6) &= LV_{entry}(pc_7) \\
LV_{exit}(pc_7) &= \emptyset
\end{aligned}$$

---

## Live Variable Analysis - example continued II

---

Solution:

pc	$LV_{entry}(pc_i)$	$LV_{exit}(pc_i)$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{y\}$
3	$\{y\}$	$\{x, y\}$
4	$\{x, y\}$	$\{y\}$
5	$\{y\}$	$\{z\}$
6	$\{y\}$	$\{z\}$
7	$\{z\}$	$\emptyset$