

# Dataflow Analysis

Wei Le

August 28, 2022

Acknowledgement: this lecture used slides from Ben Steenhoek, Profs Jeff Foster and Stephen Chong

# Outline

- ▶ what is dataflow analysis? an intuitive understanding
- ▶ dataflow problems
- ▶ dataflow algorithm
- ▶ dataflow algorithm analysis
  - ▶ lattice and dataflow analysis
  - ▶ dataflow analysis termination
  - ▶ dataflow analysis accuracy: monotonicity, distributive, pointers
- ▶ dataflow algorithm implementation
- ▶ dataflow analysis and bug detection

# History

1. 1970s, we want to know more about code so we can optimize the code
  - ▶ there exists any re-computation in the code?
  - ▶ there exists any useless computations?
2. Nowadays, we also want to know if the program will run correctly, what computation the legacy code is performing, ...

# What is dataflow analysis?

What are the data usage patterns in a program?

- ▶ is this variable always holding a constant value?
- ▶ where a definition of a variable is used?
- ▶ do two variables always hold the same value?
- ▶ is this expression already "available" (computed) at this program point?

# What is dataflow analysis

1. *dataflow analysis*: determining variable and expression relationships throughout a function or a program
2. formulated into a mathematical framework
  - ▶ to reason about the termination of dataflow analysis
  - ▶ to generalize a set of dataflow problems so we can use one algorithm to address them all

# Dataflow problems

# Three classical dataflow analysis problems

- ▶ Reaching definitions (null-pointer dereference): what definitions can reach a given program point
- ▶ Available expressions (performance issue): for each program point, what are the expressions available
- ▶ Live variables (memory leak): which variables are live at a program point

See ppt slides for example details

# Generalizing dataflow problems

Defining a dataflow problem:

- ▶ What is the problem? (what are the dataflow facts to compute at each program point?)
- ▶ Applications in compiler optimization and software engineering
- ▶ The properties of dataflow problems:
  - ▶ backward or forward (determine the direction of the dataflow analysis)
  - ▶ may or must (determine how to merge values at the branches)
- ▶ Dataflow equations (local information): Gen, Kill, In, Out

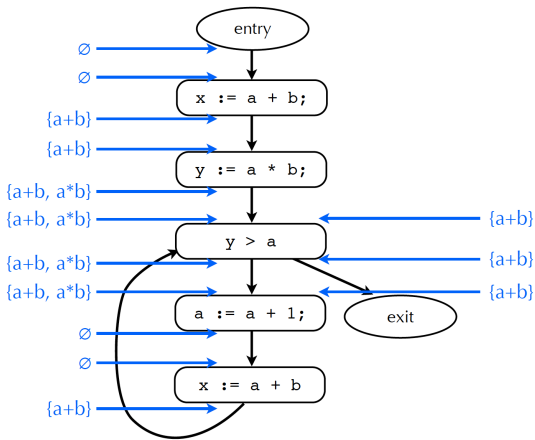


# Dataflow analysis

# Generalizing dataflow analysis

- ▶ **Goal:** solving dataflow equations and determining dataflow facts/program information for the program points in the entire program
  - ▶ local information can be propagated along program control flow to influence other nodes
  - ▶ the dataflow facts will be stabilized and the analysis can be terminated?
- ▶ **Framework:** a set of dataflow propagation algorithms for a set of dataflow problems
- ▶ **Key of the algorithm:**
  - ▶ dataflow equation computes dataflow facts locally
  - ▶ dataflow algorithms: connect dataflow facts globally, especially stabilizing the dataflow facts/solutions in presence of loops via, e.g., fixpoint

# Computing available expressions




# Forward must data flow algorithm

```
Out(s) =  $\top$  for all statements s
W := { all statements }           (worklist)
repeat {
    Take s from W
    In(s) :=  $\bigcap_{s' \in \text{pred}(s)} \text{Out}(s')$ 
    temp := Gen(s)  $\cup$  (In(s) - Kill(s))
    if (temp  $\neq$  Out(s)) {
        Out(s) := temp
        W := W  $\cup$  succ(s)
    }
} until W =  $\emptyset$ 
```

# Forward data flow again

```
Out(s) =  $\top$  for all statements s
W := { all statements }
repeat {
  Take s from W
  temp :=  $f_s(\prod_{s' \in \text{pred}(s)} \text{Out}(s'))$ 
  if (temp  $\neq$  Out(s)) {
    Out(s) := temp
    W := W  $\cup$  succ(s)
  }
} until W =  $\emptyset$ 
```

*Transfer function for statement s*



# Further reading after classes

- ▶ backward algorithm
- ▶ worklist algorithm

## Iterative Approach

Example - backward problem (live variables)

for  $i = 1$  to  $N$  do  $IN[B_i] = GEN[B_i]$

$OUT[B_{exit}] = \emptyset$

change = true

while change do

    change = false

    for each block  $B$  do

$OLDIN = IN[B]$

$OUT[B] = \bigcup_{S \in succ(B)} IN[S]$

$IN[B] = GEN[B] \cup (OUT[B] - KILL[B])$

        If  $OLDIN \neq IN[B]$  then change = true

    end for

end while

### Alternative Approach - Worklist Algorithm

Example - backward problem (very busy expressions)

for  $i=1$  to  $N$  do  $IN[B_i] = \{ \text{All expressions} \} - KILL[B_i]$

$OUT[B_{exit}] = \emptyset$

Worklist  $\leftarrow$  All blocks

while Worklist  $\neq \emptyset$  do

    get  $B$  from worklist

$OLDIN = IN[B]$

$OUT[B] = \bigcap_{S \in succ(B)} IN[S]$

$IN[B] = GEN(B) \cup (OUT[B] - KILL[B])$

    if  $OLDIN \neq IN[B]$  then

        Add  $Pred(B)$  to Worklist

endwhile

$\cap$  - start with largest solution & keep iterating till it stops shrinking.



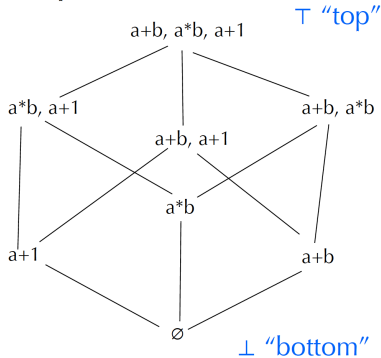
# Dataflow analysis and lattice [Kildall:1973] [Kam:1976]

For analyzing termination and for generalizing the algorithms to a set of dataflow problems

- ▶ *Lattice* ( $L$ ): the set of elements plus the order of these elements, it has a upper bound and a lower bound
- ▶ each dataflow problem has a lattice that describes the domain of the dataflow facts. That is, at each program point, the dataflow fact is an element of a lattice
- ▶ "order" among dataflow facts : element  $A$  is a conservative approximation of  $B$ ,  $A < B$ ,  $A$  and  $B$  are the set
- ▶ *transfer (flow) function* for dataflow problem: how each edge in lattice affects the dataflow facts  $F : L \rightarrow L$
- ▶ *merge function* for dataflow problem: reduce to meet operator  $\sqcap$  ( $\wedge$ ) on lattice

# Data flow facts and lattices

- Typically, data flow facts form lattices
- E.g., available expressions

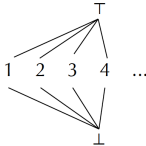


# Partial orders and lattices

- A **partial order** is a pair  $(P, \leq)$  such that
  - $\leq$  is a relation over  $P$  ( $\leq \subseteq P \times P$ )
  - $\leq$  is reflexive, anti-symmetric, and transitive
- A partial order is a **lattice** if every two elements of  $P$  have a unique least upper bound and greatest lower bound.
  - $\sqcap$  is the meet operator:  $x \sqcap y$  is the greatest lower bound of  $x$  and  $y$ 
    - $x \sqcap y \leq x$  and  $x \sqcap y \leq y$
    - if  $z \leq x$  and  $z \leq y$  then  $z \leq x \sqcap y$
  - $\sqcup$  is the join operator:  $x \sqcup y$  is the least upper bound of  $x$  and  $y$ 
    - $x \leq x \sqcup y$  and  $y \leq x \sqcup y$
    - if  $x \leq z$  and  $y \leq z$  then  $x \sqcup y \leq z$
- A join semi-lattice (meet semi-lattice) has only the join (meet) operator defined

# Useful lattices

- $(2^S, \subseteq)$  forms a lattice for any set  $S$ 
  - $2^S$  is powerset of  $S$ , the set of all subsets of  $S$ .
- If  $(S, \leq)$  is a lattice, so is  $(S, \geq)$ 
  - i.e., can “flip” the lattice
- Lattice for constant propagation



**Example:** Liveness analysis with 3 variables

$$S = \{v1, v2, v3\}$$

- V:  $2^S = \{\{v1, v2, v3\}, \{v1, v2\}, \{v1, v3\}, \{v2, v3\}, \{v1\}, \{v2\}, \{v3\}, \emptyset\}$

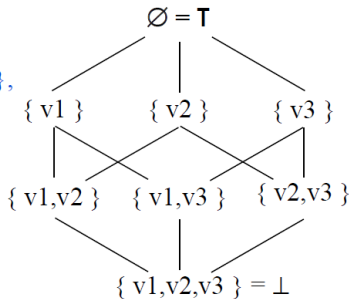
- Meet ( $\wedge$ ):  $\cup$

$$- \leq: \supseteq$$

$$- \text{Top}(T): \emptyset$$

$$- \text{Bottom}(\perp): \mathbf{v}$$

- F:  $\{f_n(X) = \text{Gen}_n \cup (X - \text{Kill}_n), \forall n\}$



# Which lattice to use?

- Available expressions
  - $P$  = sets of expressions
  - Meet operation  $\sqcap$  is set intersection  $\cap$
  - $\top$  is set of all expressions
- Reaching definitions
  - $P$  = sets of definitions (assignment statements)
  - Meet operation  $\sqcap$  is set union  $\cup$
  - $\top$  is empty set
- Monotonic **transfer function**  $f_s$  is defined based on gen and kill sets.

# Monotonicity

- A function  $f$  on a partial order is **monotonic** if
  - if  $x \leq y$  then  $f(x) \leq f(y)$
- Functions for computing  $\text{In}(s)$  and  $\text{Out}(s)$  are monotonic
  - $\text{In}(s) := \bigcap_{s' \in \text{pred}(s)} \text{Out}(s')$
  - $\text{temp} := \text{Gen}(s) \cup (\text{In}(s) - \text{Kill}(s))$  A function  $f_s$  of  $\text{In}(s)$
- Putting them together:  $\text{temp} := f_s(\bigcap_{s' \in \text{pred}(s)} \text{Out}(s'))$

# Termination

- We know the algorithm terminates
- In each iteration, either  $W$  gets smaller, or  $\text{Out}(s)$  decreases for some  $s$ 
  - Since function is monotonic
- Lattice has only finite height, so for each  $s$ ,  $\text{Out}(s)$  can decrease only finitely often

```
Out(s) =  $\top$  for all statements  $s$ 
W := { all statements }
repeat {
  Take  $s$  from  $W$ 

  In(s) :=  $\bigcap_{s' \in \text{pred}(s)} \text{Out}(s')$ 
  temp := Gen(s)  $\cup$  (In(s) - Kill(s))
  if (temp  $\neq$  Out(s)) {
    Out(s) := temp
    W := W  $\cup$  succ(s)
  }
} until W =  $\emptyset$ 
```



# Termination

- A **descending chain** in a lattice is a sequence  $x_0 < x_1 < \dots$
- The **height of a lattice** is the length of the longest descending chain in the lattice
- Then, dataflow must terminate in  $O(nk)$  time
  - $n$  = # of statements in program
  - $k$  = height of lattice
  - assumes meet operation and transfer function takes  $O(1)$  time

# Distributive data flow problems

- If  $f$  is monotonic, then we have

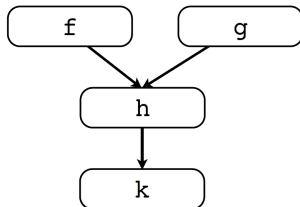
$$f(x \sqcap y) \leq f(x) \sqcap f(y)$$

- If  $f$  is **distributive** then we have

$$f(x \sqcap y) = f(x) \sqcap f(y)$$

# Benefit of distributivity

- Joins lose no information



- $k(h(f(\tau) \sqcap g(\tau)))$   
 $= k(h(f(\tau)) \sqcap h(g(\tau)))$   
 $= k(h(f(\tau))) \sqcap k(h(g(\tau)))$

# Accuracy of data flow analysis

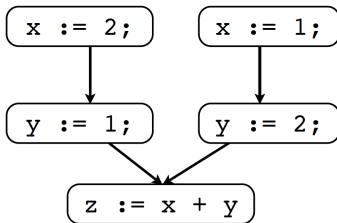
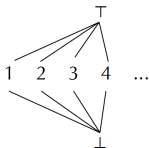
- Ideally we would like to compute the **meet over all paths** (MOP) solution:
  - Let  $f_s$  be the transfer function for statement  $s$
  - If  $p$  is a path  $s_1, \dots, s_n$ , let  $f_p = f_{s_n} \dots f_{s_1}$
  - Let  $\text{paths}(s)$  be the set of paths from the entry to  $s$
- $\text{MOP}(s) = \prod_{p \in \text{paths}(s)} f_p(\top)$
- If the transfer functions are distributive, then solving using the data flow equations in the standard way produces the MOP solution

# What problems are distributive?

- Analyses of *how* the program computes
  - E.g.,
    - Live variables
    - Available expressions
    - Reaching definitions
    - Very busy expressions
- All Gen/Kill problems are distributive

# Non-distributive example

- Constant propagation



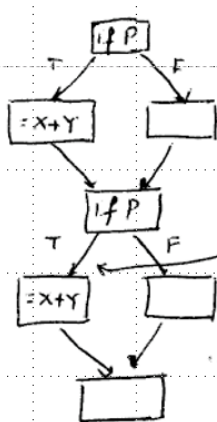
- In general, analysis of *what* the program computes is not distributive
- Thm: MOP for  $\text{In}(s)$  will always be  $\sqsubseteq$  iterative dataflow solution

# Dataflow analysis precision - conservative analysis

- ▶ For compiler optimizations, the dataflow facts we compute should definitely be true (not simply possibly true).
- ▶ However, we may miss optimization opportunities (not able to compute a complete and correct solution): two main reasons
  - ▶ Control Flow
  - ▶ Pointers & Aliasing

# Dataflow analysis precision- control flow

We assume that all paths are executable; however, some may be infeasible.



$X+Y$  is always available if we exclude infeasible paths.



# Dataflow analysis precision- pointer analysis

we may not know what a pointer points to:

1.  $X = 5$

2.  $*p = \dots$  //  $p$  may or may not point to  $X$

3.  $\dots = X$

**Constant propagation:** assume  $p$  does point to  $X$  (i.e., in statement 3,  $X$  cannot be replaced by 5).

**Dead Code Elimination:** assume  $p$  does not point to  $X$  (i.e., statement 1 cannot be deleted).

# Dataflow analysis implementation

# Dataflow analysis implementation

- ▶ Does a definition reach a point ? T or F, each variable definition is a bit, each program point has a bitvector
- ▶ Is an expression available/very busy ? T or F, each expression is a bit, each program point has a bitvector
- ▶ Is a variable live ? T or F, each variable is a bit, each program point has a bitvector

Intersection and union operations in dataflow equations can be implemented using bitwise *and* & *or* operations.

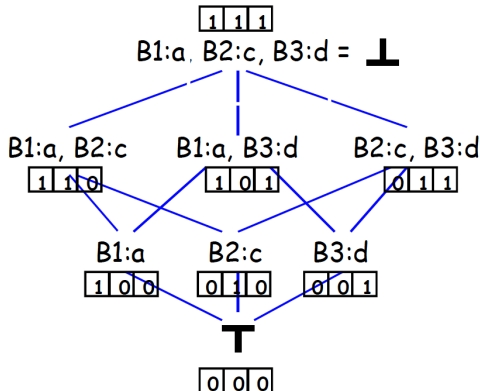
# Practical implementation

- Data flow facts are assertions that are true or false at a program point
- Can represent set of facts as bit vector
  - Fact  $i$  represented by bit  $i$
  - Intersection=bitwise and, union=bitwise or, etc
- “Only” a constant factor speedup
  - But very useful in practice

## Implementation Issues: Bit-vector Problems

The set of facts (universe of facts) can often be expressed as finite subsets of a finite base set. Such sets can be represented as bit-vectors.

Example: RD - { B1: a:=2, B2: c := d+2, B3: d := a-5 }



## Implementation Issues: Bit-vector Problems

Meet operation is either bit-wise logical **AND** or bit-wise logical **OR** .

GEN and KILL sets can be expressed as single bit-vectors.

Bit-wise logical **-** is bit-wise negation followed by bit-wise **AND** .

Implementation steps:

1. - bit-vector construction/interpretation
2. - bit-vector CFG initialization (RD, GEN, and KILL vectors)
3. - bit-vector CFG propagation
4. - information post-processing (e.g.: DU/UD chains)

# Dataflow analysis for bug detection

basic dataflow algorithms: what are the dataflow facts?

- ▶ How to detect uninitialized variables?
- ▶ How to detect memory leaks?

make the analysis more precise and less false positives

- ▶ How to detect infeasible paths
- ▶ How to make it more precise by considering pointer aliasing information
- ▶ How to track inter-procedural bugs?
- ▶ ...

# Static analysis for bug detection

- ▶ construct a cfg
- ▶ map to a dataflow problem
- ▶ dataflow analysis
- ▶ extend interprocedurally
- ▶ further improve precision: add pointer analysis and infeasible paths detection



# Further Reading

Lattice Theory by Patrick Cousot

Data flow analysis in Principles of Program Analysis