Dataflow Analysis

Wei Le

February 15, 2021

What is dataflow analysis?

Intuitively, what are the data usage patterns in a program?

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

1. *dataflow analysis*: global analysis of variables and expressions relationships
    - how the variables are defined and used?
    - is this expression already "available"? at this program point?
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

Three classical dataflow analysis problems

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

See ppt slides for example details

Generalizing dataflow analysis

Defining a dataflow problem:

- What is the problem? (what are the properties/dataflow facts to compute at each program point?)
- 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

Generalizing Dataflow Analysis

- **Goal**: solving dataflow equations, determine dataflow facts/program point for the whole program
- **Framework**: a set of data propagation algorithms for a set of dataflow problems
- **Key of the algorithm**:
  - datafow 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
- **Data structure for efficiency**: bit vector to represent the binary information

Representing Dataflow Facts

- Does a definition reach a point ? T or F, each 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 can be implemented using bitwise *and* & *or* operations.

Dataflow Algorithms – Solving Dataflow Equations

Solving dataflow equations for all program points: performing dataflow analysis for the entire program, propagating dataflow until fix points are reached (stabilize via loops)

Iterative approach

- initialize sets
- iterate over the sets till they stabilize

Example: Forward problem (Available Expressions)

```
IN[B_0] = ∅ ;  OUT[B_0] = GEN[B_0]
For i=1 to N do  OUT(B_i) = {all expressions} - KILL[B_i]
change = true
while change do
    change = false
    for each block B ≠ B_0 do
        OLDOUT = OUT[B]
        IN[B] = $\bigcap_{p \in pred(B)}$ OUT[P]
        OUT[B] = GEN[B] ∪ (IN[B] - KILL[B])
        IF OUT[B] ≠ OLDOUT then change = true
    endfor
endwhile
```

$\cap$ - start with largest estimate + iteratively shrink the solution till it stabalizes.

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
endwhile
```

Very Busy Expressions

Very Busy Expressions is an interesting variant of available expression analysis. An expression is very busy at a point if it is guaranteed that the expression will be computed at some time in the future.Thus starting at the point in question,the expression must be reached before its value changes. It is a backward and must dataflow problem.

Alternative Approach- Worklist Algorithm

Example - backward problem ($\overset{\text{very}}{\text{busy}}$ expressions)

```
for i=1 to N do  IN[$B_i$] = {All expressions} − KILL[$B_i$]
OUT[$B_{exit}$] = $\emptyset$
Worklist ← All blocks
while Worklist ≠ $\emptyset$ do
        get B from worklist
        OLDIN = IN[B]
        OUT[B] = $\bigcap_{S \in succ(B)}$ IN[S]
        IN[B] = GEN(B) ∪ (OUT[B] - KILL[B])
        If OLDIN ≠ IN[B] then
                Add Pred(B) to worklist
endwhile
```

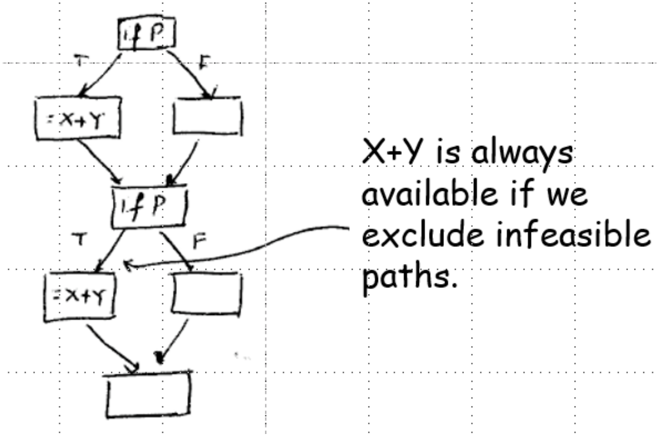
$\cap$ - start with largest solution + keep iterating till it stops shrinking.

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

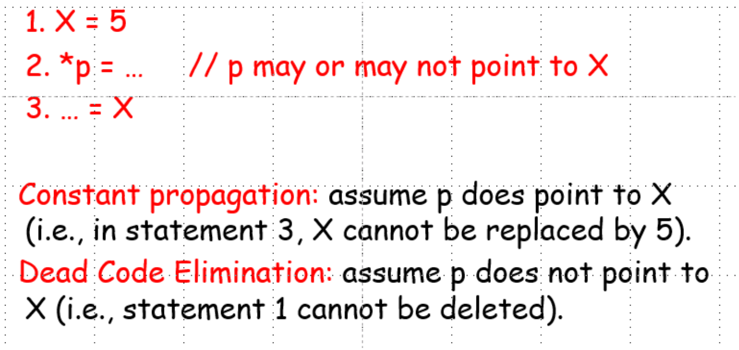
Conservative Analysis - Control Flow

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



Conservative Analysis - Pointer Analysis

we may not know what a pointer points to:



```
1. X = 5
2. *p = ...     // p may or may not point to X
3. ... = 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).

Formalize Dataflow Problems (optional)

- *Lattice (L)*: the set of elements plus the order of these elements, it has a upper bound and a lower bound
- Operators on lattice:
  - join operator – computing the least upper bound $\sqcup(\{1\},\{2\},\{3\}) = \{1,2,3\}$
  - meet operator – computing the greatest lower bound, $\sqcap(\{1\},\{1,2\}) = \{1\}$

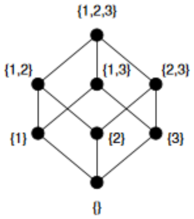


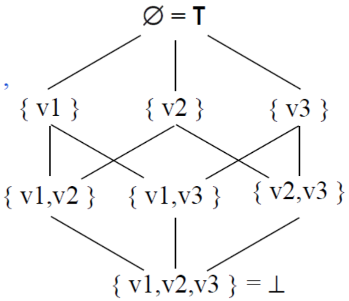
Figure 1: A Hasse diagram.

Formalize Dataflow Problems [Kildall:1973] [Kam:1976]

- 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" in dataflow: element a is a conservative approximation of b, $a < b$
- *flow function* for dataflow problem: how each node affects the dataflow fact $F : L \rightarrow L$
- *merge function* for dataflow problem: reduce to meet or join operators on lattice

Example: Liveness analysis with 3 variables
$S = \{v1, v2, v3\}$

- V: $2^S = \{\{v1,v2,v3\}, \{v1,v2\},\{v1,v3\},\{v2,v3\}, \{v1\},\{v2\},\{v3\}, \varnothing\}$
- Meet (^): $\cup$
  - $\leq$: $\supseteq$
  - Top(T): $\varnothing$
  - Bottom ($\perp$): υ
- F: $\{f_n(X) = Gen_n \cup (X - Kill_n), \forall n\}$



Dataflow analysis for bug detection

Final project examples: implement a static bug detector

basic dataflow algorithms:

- 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: both manually and with tools

- construct a cfg
- map to a dataflow problem
- dataflow analysis
- extend interprocedurally
- add pointer analysis

Static analysis

static analysis is a subject (including tools and theory) of abstraction:

- how do we merge information?
  - where the pointer points to (pointer analysis)
  - which calling context the call site bounds to (call graph)
  - which object the call site bounds to (call graph)
  - which branches the information come from? (merge problems in dataflow analysis)
- another type of approximation: allow mistakes?
  - flow insensitive analysis: ignore the order of the statements
  - context insensitive: ignore the calling context
  - path-insensitive: do not distinguish the paths
- does this approximation (e.g., merge) sound?
  - compilation optimization needs to be sound
  - conservative analysis is sound

Further Reading

Lattice Theory by Patrick Cousot
Data flow analysis in Principles of Program Analysis