CS6013 - Modern Compilers: Theory and Practise

Data flow analysis

V. Krishna Nandivada

IIT Madras

Data Flow Analysis

Why:

- Provide information about a program manipulates its data.
- Study functions behavior.
- To help build control flow information.
- Program understanding (a function sorts an array!).
- Generating a model of the original program and verify the model.
- The DFA should give information about that program that does not misrepresent what the procedure being analyzed does.
- Program validation.

Reaching Definitions

A particular definition of a variable is said to reach a given point if

- there is an execution path from the definition to that point
- the variable might may have the value assigned by the definition.

In general undecidable.

- The analysis must be conservative – the analysis should not tell us that a particular definition does not reach a particular use, if it may reach.
- A 'may' conservative analysis gives us a larger set of reaching definitions than it might, if it could produce the minimal result.

To make maximum benefit from our analysis, we want the analysis to be conservative, but as aggressive as possible.

Different types of analysis

- Intra procedural analysis.
- Whole program (inter-procedural) analysis.
- Generate intra procedural analysis and extend it to whole program.

We will study an iterative mechanism to perform such analyses.

Iterative Dataflow Analysis

- Build a collection of data flow equations – specifying which data may flow to which variable.
- Solve it iteratively.
- Start from a conservative set of initial values – and continuously improve the precision.
  Disadvantage: We may be handling large data sets.
- Start from an aggressive set of initial values – and continuously improve the precision.
  Advantage: Datasets are small to start with.
- Choice – depends on the problem at hand.

Example program

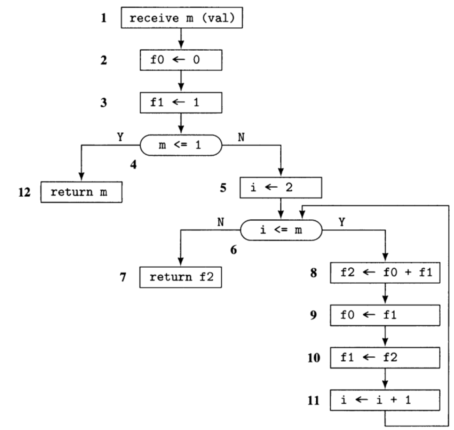
```
1   int g(int m, int i);

2   int f(n)
3       int n;
4   {   int i = 0, j;
5       if (n == 1) i = 2;
6       while (n > 0) {
7           j = i + 1;
8           n = g(n,i);
9       }
10      return j;
11  }
```

- Does def of `i` in line 4 reach the uses in line 7 and 8?
- Does def of `j` in line 7 reach the use in line 10?

Definitions

- GEN : GEN(b) returns the set of definitions generated in the basic block b; assigned values in the block and not subsequently killed in it.
- KILL : KILL(b) returns the set of definitions killed in the basic block b.
- IN : IN(b) returns the set of definitions reaching the basic block b.
- OUT : OUT(b) returns the set of definitions going out of basic block b.
- PRSV : Negation of KILL

Representation and Initialization



Bit Position	Definition	Basic Block
1	m in node 1	B1
2	f0 in node 2	
3	f1 in node 3	
4	i in node 5	B3
5	f2 in node 8	B6
6	f0 in node 9	
7	f1 in node 10	
8	i in node 11	

	Set rep	Bit vector
GEN(B1)	$= \{1, 2, 3\}$	$\langle 11100000 \rangle$
GEN(B3)	$= \{4\}$	$\langle 00010000 \rangle$
GEN(B6)	$= \{5, 6, 7, 8\}$	$\langle 00001111 \rangle$
GEN(.)	$= \{\}$	$\langle 00000000 \rangle$

Populating PRSV, OUT and IN

	Set rep	Bit vector
$PRSV$(B1)	= $\{4, 5, 8\}$	$\langle 00011001 \rangle$
$PRSV$(B3)	= $\{1, 2, 3, 5, 6, 7\}$	$\langle 11101110 \rangle$
$PRSV$(B6)	= $\{1\}$	$\langle 10000000 \rangle$
$PRSV(.)$	= $\{1, 2, 3, 4, 5, 6, 7, 8\}$	$\langle 11111111 \rangle$

Dataflow equations

A definition may reach the end of a basic block i:

$$OUT(i) = GEN(i) \cup (IN(i) \cap PRSV(i))$$

or with bit vectors:

$$OUT(i) = GEN(i) \vee (IN(i) \wedge PRSV(i))$$

A definition may reach the beginning of a basicblock i:

$$IN(i) = \bigcup_{j \in Pred(i)} OUT(j)$$

- GEN, $PRSV$ and OUT are created in each basic block.
- $OUT(i)$ = $\{\}$ // initialization
- But IN needs to be initialized to something safe.
- $IN(entry)$ = $\{\}$

Solving the Dataflow equations: example

Itr 1:

$$\begin{array}{llll}
OUT(entry) & = \langle 00000000 \rangle & IN(entry) & = \langle 00000000 \rangle \\
OUT(B1) & = \langle 11100000 \rangle & IN(B1) & = \langle 00000000 \rangle \\
OUT(B2) & = \langle 11100000 \rangle & IN(B2) & = \langle 11100000 \rangle \\
OUT(B3) & = \langle 11110000 \rangle & IN(B3) & = \langle 11100000 \rangle \\
OUT(B4) & = \langle 11110000 \rangle & IN(B4) & = \langle 11110000 \rangle \\
OUT(B5) & = \langle 11110000 \rangle & IN(B5) & = \langle 11110000 \rangle \\
OUT(B6) & = \langle 00001111 \rangle & IN(B6) & = \langle 11110000 \rangle \\
OUT(entry) & = \langle 11110000 \rangle & IN(exit) & = \langle 11110000 \rangle
\end{array}$$

Itr 2:

$$\begin{array}{llll}
OUT(entry) & = \langle 00000000 \rangle & IN(entry) & = \langle 00000000 \rangle \\
OUT(B1) & = \langle 11100000 \rangle & IN(B1) & = \langle 00000000 \rangle \\
OUT(B2) & = \langle 11100000 \rangle & IN(B2) & = \langle 11100000 \rangle \\
OUT(B3) & = \langle 11110000 \rangle & IN(B3) & = \langle 11100000 \rangle \\
OUT(B4) & = \langle 11111111 \rangle & IN(B4) & = \langle 11111111 \rangle \\
OUT(B5) & = \langle 11111111 \rangle & IN(B5) & = \langle 11111111 \rangle \\
OUT(B6) & = \langle 10001111 \rangle & IN(B6) & = \langle 11111111 \rangle \\
OUT(entry) & = \langle 11111111 \rangle & IN(exit) & = \langle 11111111 \rangle
\end{array}$$

Dataflow equations: behavior

- We specify the relationship between the data-flow values before and after a block – transfer or flow equations.
  - Forward: $OUT(s) = f(IN(s), \cdots)$
  - Backward: $IN(s) = f(OUT(s), \cdots)$
- The rules never change a 1 to 0. They may only change a 0 to a 1.
- They are monotone.
- Implication – the iteration process will terminate.
- Q: What good is reaching definitions? undefined variables.
- Q: Why do the iterations produce an acceptable solution to the set of equations? – lattices and fixed points.

Lattice

- **What** : Lattice is an algebraic structure
- **Why** : To represent abstract properties of variables, expressions, functions, etc etc.
    - Values
    - Attributes
    - ...
- **Why “abstract?** Exact interpretation (execution) gives exact values, abstract interpretation gives abstract values.

Lattice definition

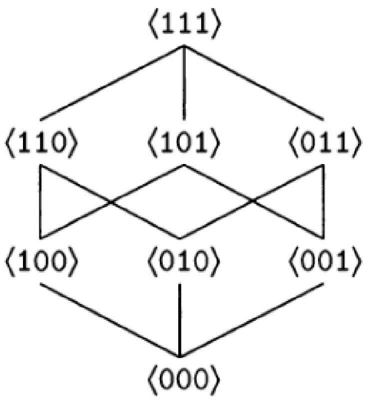
A lattice L consists of a set of values, and two operations called *meet* ($\sqcup$) and *join* ($\sqcap$). Satisfies properties:

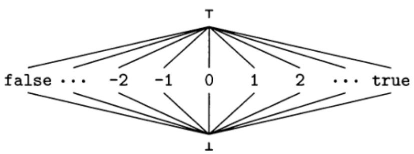
- **closure**: For all $x, y \in L$, $\exists$ a unique z and $w \in L$, such that $x \sqcap y = z$ and $x \sqcup y = w$ – each pair of elements have a unique lub and glb.
- **commutative**: For all $x, y \in L$, $x \sqcap y = y \sqcap x$, and $x \sqcup y = y \sqcup x$.
- **associative**: For all $x, y, z \in L$, $(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$, and $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$
- There exists two special elements of L called bottom ($\perp$), and top ($\top$).
  $\forall x \in L$, $x \sqcap \perp = \perp$ and $x \sqcup \top = \top$.
- **distributive** : (optional). $\forall x, y, z \in L$, $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$, and $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$

Lattice properties

- Meet (and join) induce a partial order ($\sqsubseteq$):
  $\forall x, y \in L$, $x \sqsubseteq y$, iff $x \sqcap y = x$.
- Transitive, antisymmetry and reflexive.

Example Lattices:





Monotones and fixed point

- A function $f : L \rightarrow L$, is a monotone, if for all $x, y \in L$, $x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$.
- Example: bit-vector lattice:
    - $f(x_1 x_2 x_3) = \langle x_1 1 x_2 \rangle$
    - $f(x_1 x_2 x_3) = \langle x_2 x_3 x_1 \rangle$
- A flow function models the effect of a programming language construct. as a mapping from the lattice for that particular analysis to itself.
- We want the flow functions to be monotones. Why?
- A fixed point of a function $f : L \rightarrow L$ is an element $z \in L$, such that $f(z) = z$.
- For a set of data-flow equations, a fixed-point is a solution of the set of equations – cannot generate any further refinement.

Meet Over All Paths solutions

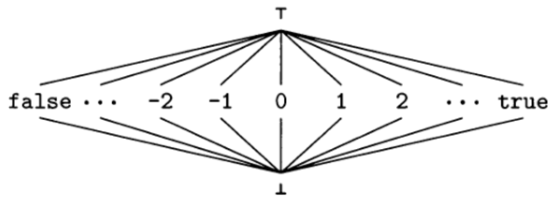
- The value we wish to compute in solving data-flow equations is – meet over all paths (MOP) solution.
- Start with some prescribed information at the entry (or exit depending on forward or backward).
- Repeatedly apply the composition of the appropriate flow functions.
- For each node form the meet of the results.

A worklist based implementation

```
procedure Worklist_Iterate(N,entry,F,dfin,Init)
   N: in set of Node
   entry: in Node
   F: in Node × L ⟶ L
   dfin: out Node ⟶ L
   Init: in L
begin
   B, P: Node
   Worklist: set of Node
   effect, totaleffect: L
   dfin(entry) := Init
   Worklist := N - {entry}
   for each B ∈ N do
      dfin(B) := ⊤
   od
   repeat
      B := ♦Worklist
      Worklist -= {B}
      totaleffect := ⊤
      for each P ∈ Pred(B) do
         effect := F(P,dfin(P))
         totaleffect ⊓= effect
      od
      if dfin(B) ≠ totaleffect then
         dfin(B) := totaleffect
         Worklist ∪= Succ(B)
      fi
   until Worklist = ∅
end    || Worklist_Iterate
```


Example: Constant Propagation

Goal: Discover values that are constants on all possible executions of a program and to propagate these constant values as far forward through the program as possible
Conservative: Can discover only a subset of all the possible constants.
Lattice:



Constant Propagation lattice meet rules

- $\bot$ = Constant value cannot be guaranteed.
- $\top$ = May be a constant, not yet determined.
- $\forall x$
  - $x \sqcap \top = x$
  - $x \sqcap \bot = \bot$
  - $c_1 \sqcap c_1 = c_1$
  - $c_2 \sqcap c_1 = \bot$

Simple constant propagation

- Gary A. Kildall: A Unified Approach to Global Program Optimization - POPL 1973.
- Reif, Lewis: Symbolic evaluation and the global value graph - POPL 1977.
- **Simple constant** Constants that can be proved to be constant provided,
  - no information is assumed about which direction branches will take.
  - Only one value of each variable is maintainedalong each path in the program.

Kildall's algorithm

- Start with an entry node in the program graph.
- Process the entry node, and produce the constant propagation information. Send it to all the immediate successors of the entry node.
- At a merge point, get an intersection of the information.
- If at any successor node, if for any variable the value is "reduced, the process the successor, similar to the processing done for entry node.

Constant propagation - equations

- Let us assume that one basic block per statement.
- Transfer functions set F - a set of transfer functions. $f_s \in F$ is the transfer function for statement s.
- The dataflow values are given by a map: m: $Vars \rightarrow ConstantVal$
- If m is the set of input dataflow values, then $m' = f_s(m)$ gives the output dataflow values.
- Generate equations like before.

Constant propagation: equations (contd)

- Start with the entry node.
- If s is not an assignment statement, then f_s is simply the identity function.
- If s is an assignment statement to variable v, then
- Else, $f_s(m) = m'$, where:
  - For all $v' \neq v$, $m'(v') = m(v')$.
  - If the RHS of the statement is a constant c, then $m'(v) = c$.
  - If the RHS is an expression (say $y op z$),

$$m'(v) = \begin{cases} m(y) op m(z) & \text{if } m(y) \text{ and } m(z) \text{ are constant values} \\ \perp & \text{if either of } m(y) \text{ and } m(z) \text{ is } \perp \\ \top & \text{Otherwise} \end{cases}$$

  - If the RHS is an expression that cannot be evaluated, then $m'(v) = \perp$.
- At a merge point, get a meet of the flow maps.

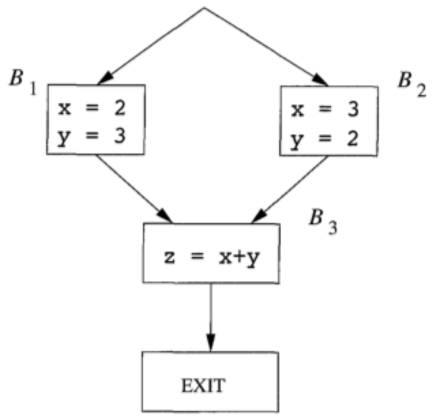
Constant Propagation - example

```
x = 10;
y = 1;
if  (cond) {
   y = y / x;
   x = x - 1;
} else {
   y = 0;
}
print x +  y;
```


Constant Propagation - example

```
x = 10;
y = 1;
while (x > 1) {
    y = x * y;
    x = x - 1;
}
A[x] = y;
```


constant propagation: Non distributive



Say f_1, f_2, and f_3 represent the transfer functions of B_1, B_2 and B_3, respectively.

$f_3(f_1(m_0) \wedge f_2(m_0)) \sqsubseteq$
$f_3(f_1(m_0)) \wedge f_3(f_2(m_0))$

m	$m(x)$	$m(y)$	$m(z)$
m_0	UNDEF	UNDEF	UNDEF
$f_1(m_0)$	2	3	UNDEF
$f_2(m_0)$	3	2	UNDEF
$f_1(m_0) \wedge f_2(m_0)$	NAC	NAC	UNDEF
$f_3(f_1(m_0) \wedge f_2(m_0))$	NAC	NAC	NAC
$f_3(f_1(m_0))$	2	3	5
$f_3(f_2(m_0))$	3	2	5
$f_3(f_1(m_0)) \wedge f_3(f_2(m_0))$	NAC	NAC	5