Compiler Design Notes

Ayush Raina

February 8, 2025

Control Flow Analysis

Why Control Flow Analysis?

Helps to understand structure of Control Flow Graph (CFG), To detect loops in the CFG, Dominator Information, Dominator Frontier Information required for Single Static Assignment (SSA) form. Interval Information used in Data Flow Analysis, Control Dependence Information used in Parallelization.

Dominators

1. We write $(d \text{ dom } n)$ if every path from initial node n_0 to node n passes through node d. A node dominates itself.

2. Node x strictly dominates node y if x dominates y and $x \neq y$.

3. x is immediate dominator of y if x is closest strict dominator of y.

Algorithm to find Dominators

Let us denote the set of Dominators of node n as $D(n) = OUT(n)$. These sets are very helpful to construct the dominator tree quickly.

- Start with n_0 such that $D(n_0) = \{n_0\}$.

- for each $n \in N - \{n_0\}$ do the following:
    - First set $D(n) = N$.
    - Take intersection of all the outsets of the predecessors of n denoted by $IN(n) = \bigcap_{p \in pred(n)} OUT(p)$.
    - $D(n) = IN(n) \cup \{n\}$.

- Repeat until $D(n)$ stabilizes for all n.

Back Edges and Natural Loops

Back Edges: Edges whose head dominates the tail are called back edges. To find the back edges, one easy way to look if opposing edges according to dominator tree is present in the graph.

Natural Loops: Given a back edge (n, d), the natural loop of the edge is $d \cup$ all the nodes that reach n without going through d. In this case d is the header of the loop and dominates all the nodes in the loop.

Algorithm for finding the natural loop of a back edge

Let the back edge be (n, d). Then initialize a empty stack. Push n into the stack. Initialize a vector with d in it. Then until the stack is empty, pop the top element and push its predecessors into the stack. If the predecessor is not in the vector, add it to the vector. Continue until the stack is empty. The vector now contains the natural loop.

Depth First Numbering of Nodes in a CFG

We start a $DFS(n)$ from initial node n_0 where n is total number of nodes. We mark a node to n when it is last visit to that node. Last visit means that all the children of that node are visited and now this node will not be visited again. Then we decrement n and continue the process. This way we can number the nodes in a CFG.
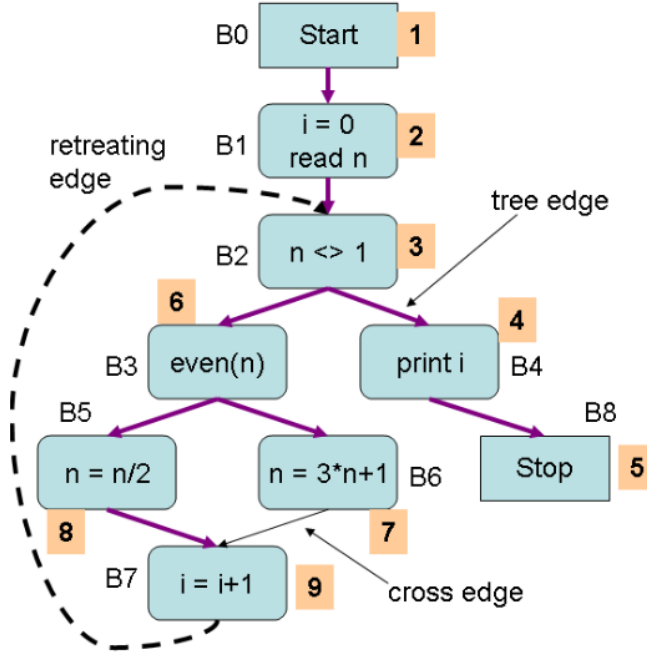
Figure 1: DFS Numbering of Nodes

Reducibility

A flow graph $\mathcal{G}$ is reducible iff every back it can be partitioned into 2 disjoint sets of forward and back edges such that:

- Forward edges form a DAG in which every node is reachable from the initial node.

- All the retreating edges are back edges.

- In an irreducible flow graph, some retreating edges will not be back edges, hence graph of forward edges will contain a cycle.
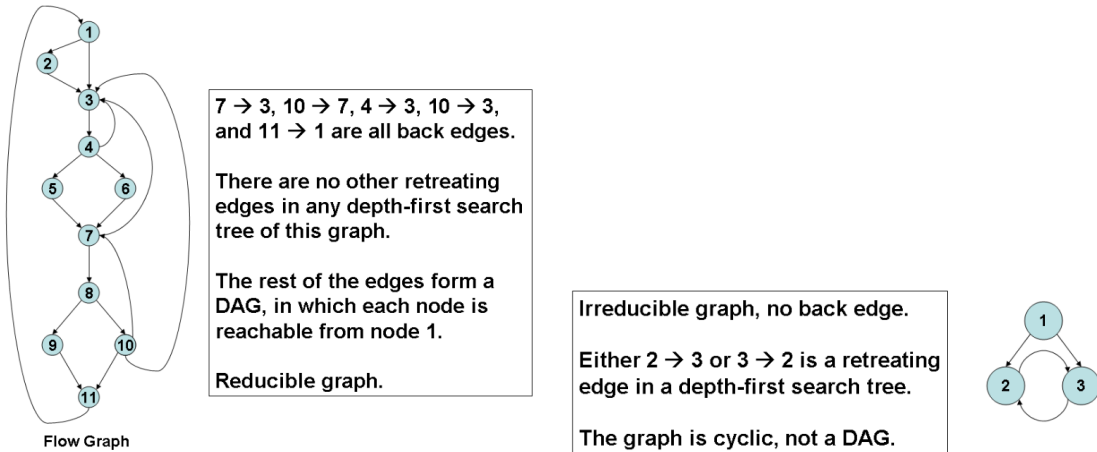

Figure 2: Reducible and Irreducible CFG

Inner Loops

Unless two loops have same header $n \longrightarrow d$ (here d is header), they are either disjoint or one is nested inside the other. If natural loop set of one header is subset of the other then there is nesting. Similarly two loops are disjoint if their natural loop sets are disjoint and two loops are identical if their natural loop sets are identical.

If two loops share a header, neither of these may hold and in such a case loops are combined and transformed.
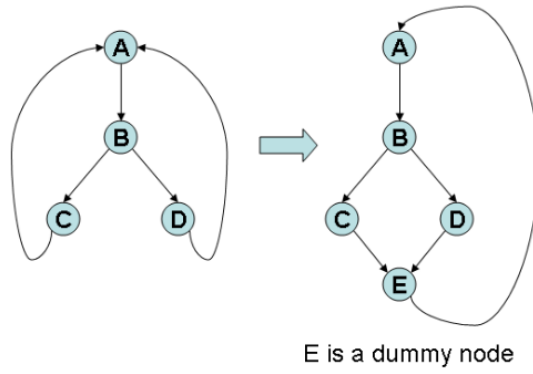
E is a dummy node

Figure 3: Transformation in case of shared header
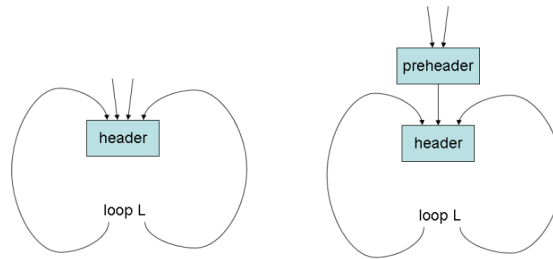
Pre Header

Figure 4: Pre Header

Depth of a CFG and Convergence of DFA Problem

To be continued.

Intervals

Intervals have a header node which dominates all the other nodes in that interval. Given a flow graph $\mathcal{G}$ with initial node n_0. The interval with header n denoted by $I(n)$ is the set defined as:

1. $n \in I(n)$.
2. If $\exists\, m$ such that $Pred(m) \subset I(n)$ then $m \in I(n)$.
3. Nothing else is in $I(n)$.

Constructing $I(n)$:
1. Start with $I(n) = \{n\}$.
2. While there exists a node m such that $Pred(m) \subset I(n)$, do $I(n) = I(n) \cup \{m\}$.

Paritioning a flow graph into disjoint intervals

1. Mark all the nodes as unvisited.
2. Construct $I(n_0)$ and mark all the nodes in $I(n_0)$ as visited.
3. while there is an unvisited node m with atleast one predecessor p which is selected, construct $I(m)$ and mark all the nodes in $I(m)$ as visited and repeat the process.

I(1) = {1,2}; I(3) = {3}
I(4) = {4,5,6}; I(7) = {7,8,9,10,11}
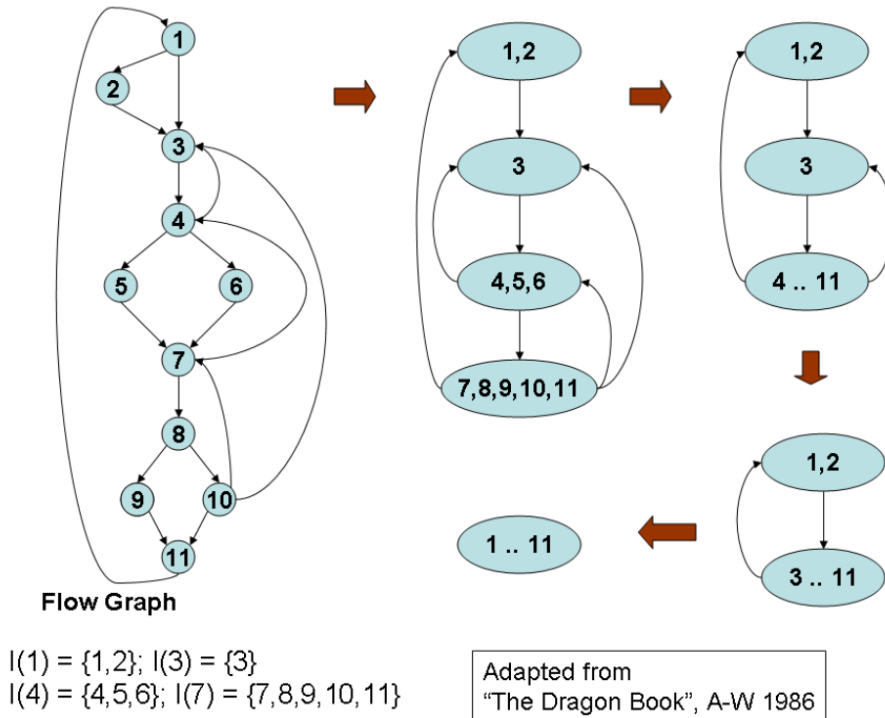
Adapted from
"The Dragon Book", A-W 1986

Figure 5: Intervals

In 1st step we got disjoint intervals. Now in the above figure we repeat same procedure further on the interval graph until we reach single node.

Interval Graphs

1. Interval graph is denoted by $I(\mathcal{G})$. Intervals correspond to nodes and interval containing n_0 is the initial node of $I(\mathcal{G})$.
2. If there is an edge from a node in $I(m)$ to the header of interval $I(n)$ then there is an edge from $I(m)$ to $I(n)$ in $I(\mathcal{G})$.
3. We make intervals in interval graph until we reach a limit flow graph which cannot be reduced further.
4. A flow graph is reducible iff its limit flow graph is a single node.

Node Splitting (Extra)

If we reach limit flow graph which is not a single node, we can still proceed further if we split one or more nodes.
1. If a node has k predecessors then we may replace n by k nodes $n_1, n_2, \ldots, n_k$ such that i^{th} predecessor of n becomes predecessor of n_i and all the successors of n become successors of $n_i \forall i \in [1, k]$.
2. After splitting the nodes, we continue reduction process and do node splitting wherever required to reach a single node limit flow graph, however success is not guaranteed.
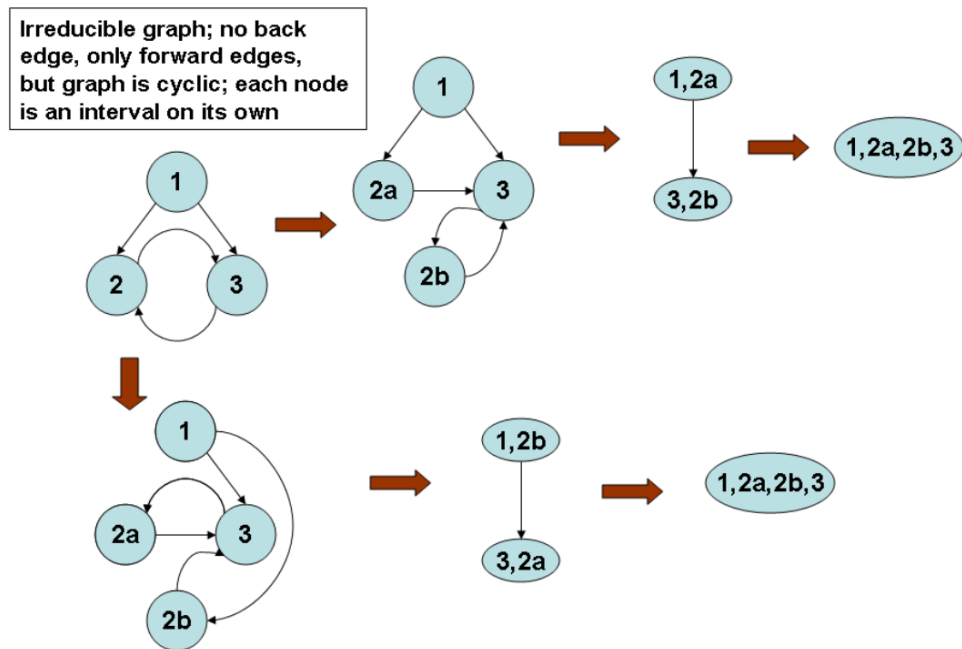
Figure 6: Node Splitting

T1-T2 Transformations and Graph Reduction

T1 Transformation: If n is a node with a loop i.e $(n \to n)$ edge exists then delete that edge.
T2 Transformation: If there is a node n, not the initial node with a unique predecessor m then combine n into m.

By Applying T1,T2 transformations in any order we may reach limit flow graph and node splitting may be required wherever necessary in order to reach a single node limit flow graph.
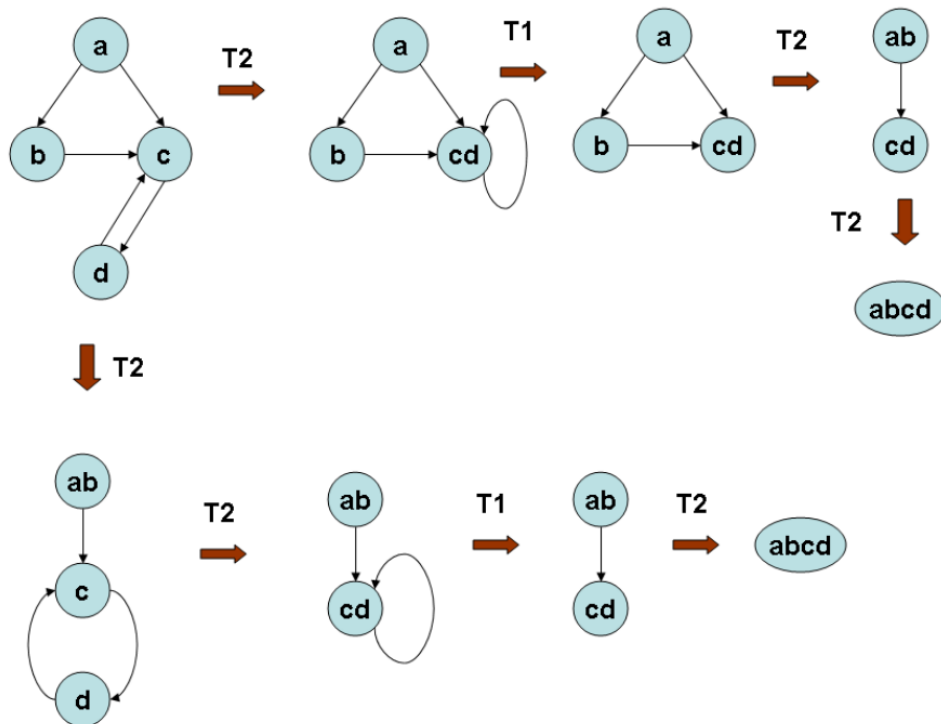

Figure 7: T1-T2 Transformations

One more example of a bigger CFG can be found in slides available on course webpage.

Regions

A set of nodes N that includes a header and it dominates all other nodes in N is called region and all these edges between nodes in N are in the region except (possibly) the edges that enter the header.

All intervals are regions but all regions are not intervals. A region may have multiple exits but an interval has only one exit. As we reduce a flow graph G using $T1 - T2$ transformations, at each step the node obtained is a region.
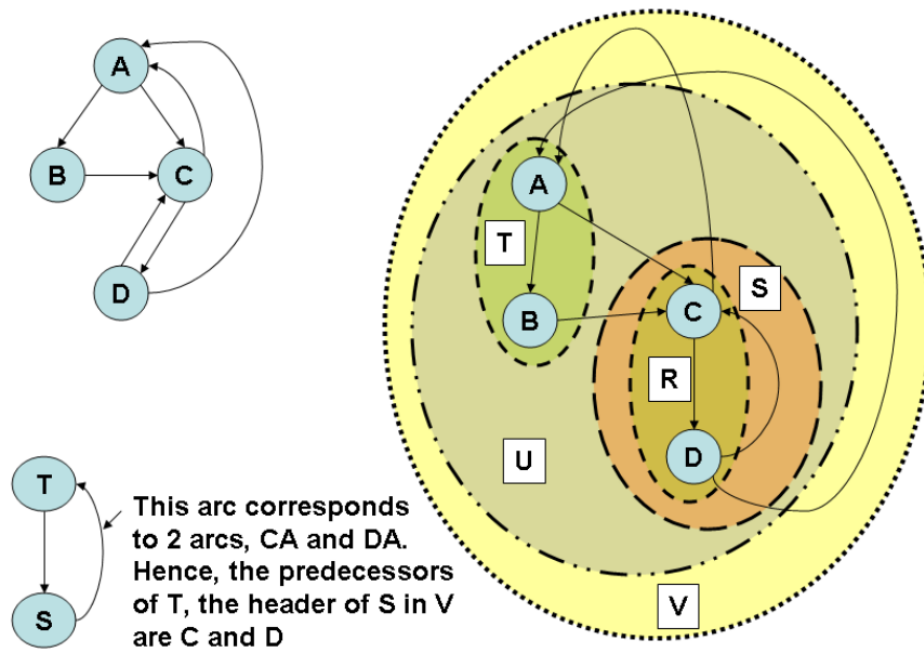

This arc corresponds to 2 arcs, CA and DA. Hence, the predecessors of T, the header of S in V are C and D

Figure 8: R,S,T,U,V are different regions marked

End of Control Flow Analysis

Data Flow Analysis

These are the techniques that derive information about flow of data along execution paths.

An execution path from path p_1 to p_n is sequence of points such that for each $i \in [1, n-1]$ either p_i is preceeding some statement s and p_{i+1} is immediately following the same statement s or p_i is end of some block and p_{i+1} is start of successor block.

In general there can be infinite number of paths from a program and there is no bound on length of a path.

Uses of DFA

Common uses are program debugging and program optimizations.

Program debugging: may include what are the definitions of variables reaching a particular point ? These are reaching definitions.

Program Optimization: It includes optimizations like common subexpression elimination.

Data Flow Analysis Schema

Let us first see some definitions:

1. **Data flow value:** for a program point represents an abstraction of all possible program states that can be observed from that particular point in the program. The set of all possible data flow values is called Domain of the application under consideration.

2. **IN(s) and OUT(s)**: represents the data flow values at beginning and end of each statement s.

3. **Data Flow Problem:** is to find a solution to set of constraints on $IN(s)$ and $OUT(s)$ for all statements s. These constraints are of two types: constraints based on semantics of statements such as transfer functions and constraints based on control flow such as if-else conditions.

4. **Safe Estimates:** A decision or estimate is safe if it never leads to a change in what is being computed in the program (after the change).

A DFA schema consists of:
1. Control Flow Graph (CFG)
2. Direction of Data Flow (Forward or Backward)
3. Set of Data Flow Values (Domain)
4. Confluence Operator
5. Transfer Functions for each block

Reaching Definitions

Purpose of reaching definitions is to perform optimizations by computing loop invariant. A definition d reaches a point p if there is a path from the point immediately following d to p such that d is not killed on that path. We say a definition of a variable is killed if between 2 points along the path, there is re-assignment to the variable.

The data flow equations(constraints) are as follows: Initialize $IN(B) = \phi$ for all blocks B. If some definitions reach B_1 (entry block) then $IN(B_1)$ is initialized with those definitions. Then we have:

1. $IN(B) = \bigcup_{p \in pred(B)} OUT(p)$
2. $OUT(B) = GEN(B) \cup (IN(B) - KILL(B))$

This is a forward DFA problem since $OUT(B)$ is computed in terms of $IN(B)$ and Confluence operator is $\cup$. $GEN(B)$ is the set of definitions which are visible immediately after the block B and $KILL(B)$ is the union of definitions in all basic blocks which are killed by individual definitions in B. Here is an example of reaching definitions:
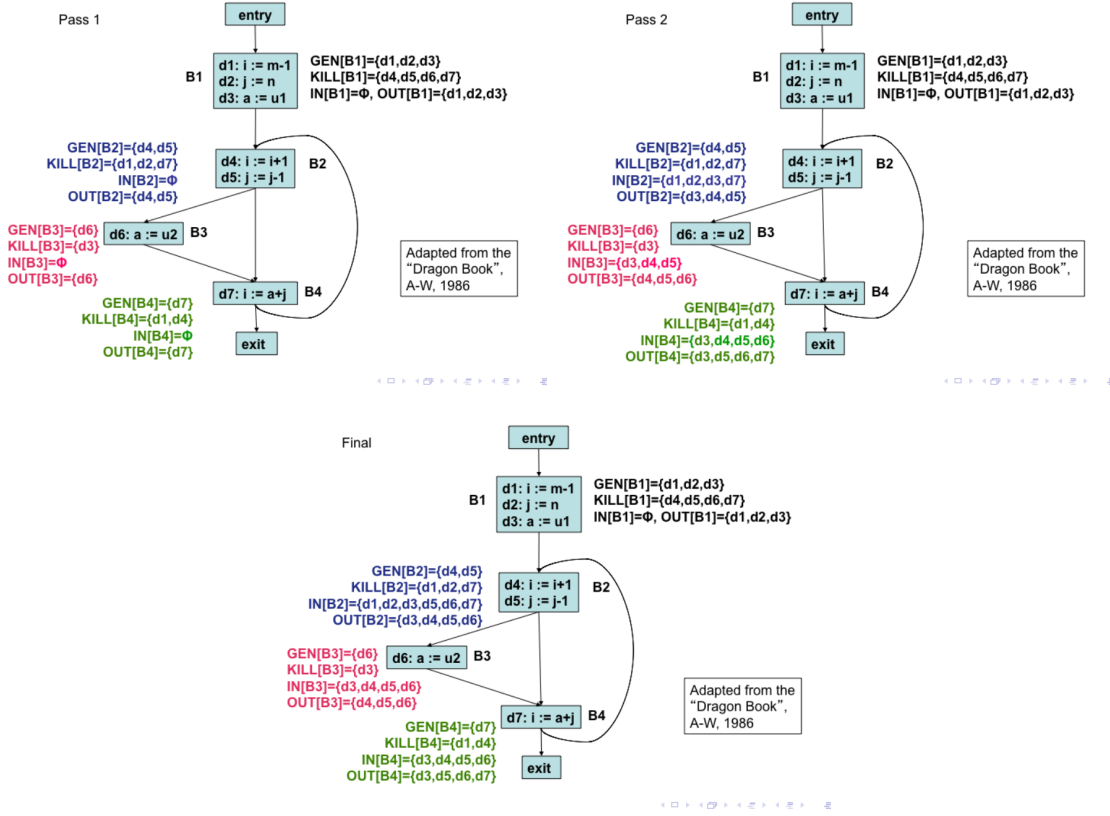
Figure 9: Reaching Definitions

In Pass 1, we initialize $IN(B) = \phi$ for all blocks and compute $OUT(B)$ using GEN, $KILL$ and IN values. Now we repeat the process until IN and OUT values stabilize which in this case takes 3 iterations.

Iterative Algorithm for computing Reaching Definitions

1. for each block B initialize $IN(B) = \phi$ and $OUT(B) = GEN(B)$.
2. while there is a block B such that $OUT(B)$ changes, do the following for each block B:

- $IN(B) = \bigcup_{p \in pred(B)} OUT(p)$

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

GEN, KILL, IN, OUT are all represented as bit vectors with one bit for each definition in the flow graph which is shown in following figure.
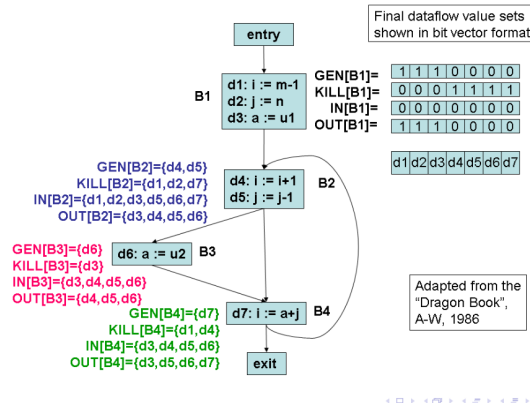

Figure 10: Bit Vector Representation

Use-Definition Chains (u-d chains)

Convenient way to store reaching definitions once they are computed. A u-d chain is a **list of a use of a variable** and all the definitions that reach that use.
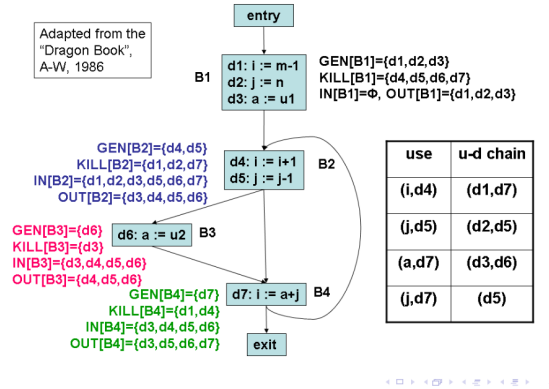

Figure 11: Use-Definition Chains

Construction of ud chains

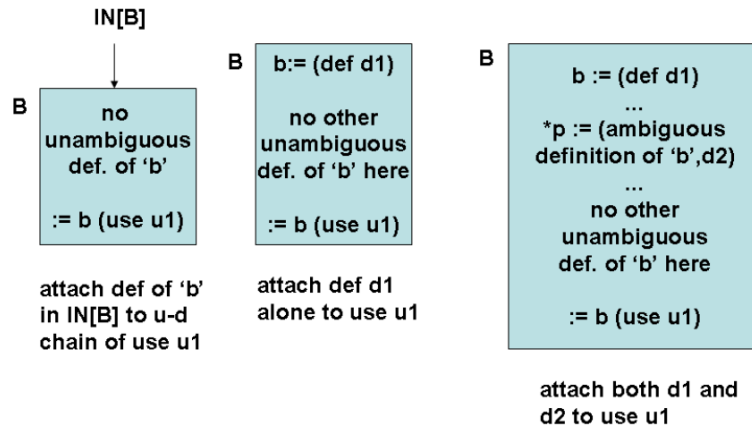We need to consider 3 cases while constructing use-def chains which are as follows:

Figure 12: Construction of ud chains

Available Expression Computation

This computation is necessary to handle common subexpression elimination optimization. This is again a forward flow problem with $\cap$ as Confluence operator.

An Expression E_x is said to be available at a point p if every path (may contain cycle) from initial node which reaches p evaluates E_x somewhere along the path before reaching p. There can be multiple evaluations of E_x but after last such evaluation there are no assignments to any of the variables in E_x.
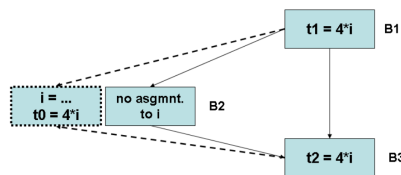

Figure 13: Available Expressions

In the figure above we can see that if we take path B_2 right then first evaluation to $4 * i$ is in B_1 and it is available at B_2. If we take path B_2 left then first evaluation to $4 * i$ then there is reassigment to i and $4 * i$ is again computed,

hence it is available at B_2 in this case as well. In this case last evaluation of the expression is in B_2 and after the last evaluation there was no further re assignment to i.

A block kills E_x if it assigns to any of the variables in E_x. For example consider $E(x,y) = x + y$. A block kills $E(x,y)$ if it assigns to x or y. A block generates E_x if it computes E_x and does not kill it.

Data Flow Equations for Available Expressions

The data flow equations are as follows:
1. $IN(B) = \bigcap_{p \in pred(B)} OUT(p)$ where B is not the entry block.
2. $OUT(B) = \text{e\_gen}(B) \cup (IN(B) - \text{e\_kill}(B))$
3. $IN(B_1) = \phi$ where B_1 is the entry block.
4. $IN(B) = U$, for $B \neq B_1$ (Initialization)

$IN(B_1)$ is always ϕ, U is the set of all expressions

Computing e_gen and e_kill

Computing e_gen[p]
1. A = A $\cup$ {y+z}
2. A = A − {all expressions involving x}
3. e_gen[p] = A

e_gen[q] = A   q .
     x = y + z
     p .

e_kill[q] = A   q .
     x = y + z
     p .

Computing e_kill[p]
1. A = A − {y+z}
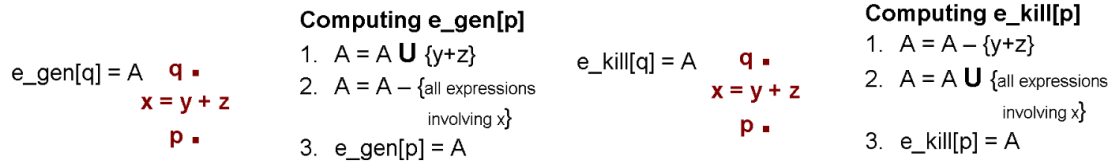2. A = A $\cup$ {all expressions involving x}
3. e_kill[p] = A

Figure 14: Computing e_gen and e_kill

In the left figure we did $A = A - \{$all expressions involving x$\}$ because of re assignment to x. In the right figure we did $A = A - \{y + z\}$ because in this case A is set of killed expressions and $\{y + z\}$ is generated and due to re assignment to x, all expressions involving x are added to A.