Data-Flow Analysis

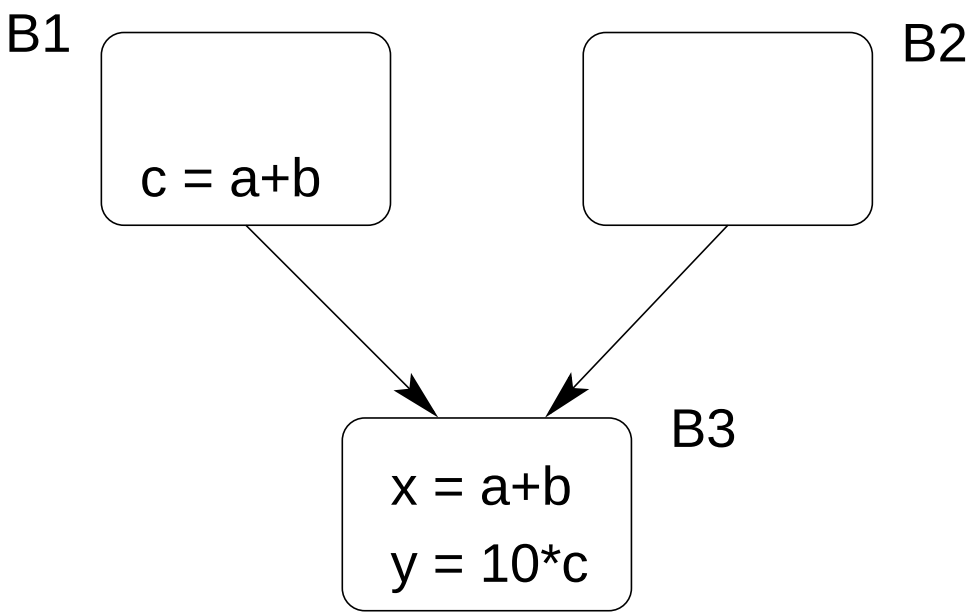
Data-Flow Analysis

- A collection of methods that extract information about the flow of data along the paths of execution of a program.
- As an example two identical (textually) subexpression may or may not evaluate to the same value on different execution paths.

Data-Flow Analysis

- If the definition of a variable is not used subsequently, it may be eliminated as a dead code. So it is necessary to detect whether a variable is live at some point.
- There are other important questions e.g. loop-invariant detection, identification of induction variables etc.

Same, Different or Dead



Why?

- The expression `a+b` in B_1 and B_3 may not evaluate to the same value[a].
- The statements `x = a+b` in B_3 is a dead code if `x` is not used any where down the path of execution of the program.

[a] Note the advantage of SSA in this connection.

Execution States and State Transformation

- A state of a program consists of values in different variables and the content of all activation records in the stack.
- A statement changes the program state.

State before: $\cdots$ `i:5` $\cdots$

```
i = i + 1
```

State after: $\cdots$ `i:6` $\cdots$

- Each state on the control path is a program point (p).

Data-Flow Analysis

- It is necessary to considers all possible sequences of states and their transformations through different paths in the control flow graph of the program.
- Different information is extracted depending on the type of data-flow analysis.

Data-Flow Analysis

- Some analysis are along the direction of the control-flow known as a forward analysis.
- Some are in the reverse direction of control-flow, known as a backward analysis.
- Some analysis are on the existence of a property on a path, some are on all possible paths.

State Sequence and Program Point

- Within a basic block, the state after a statement i is the state before the statement $i + 1$[a]
- In a forward flow problem, the state before the first statement of a basic block B, is a combination of the states after the last statements of its predecessor blocks.

[a]The computation of states may be in the forward or in the backward directions.

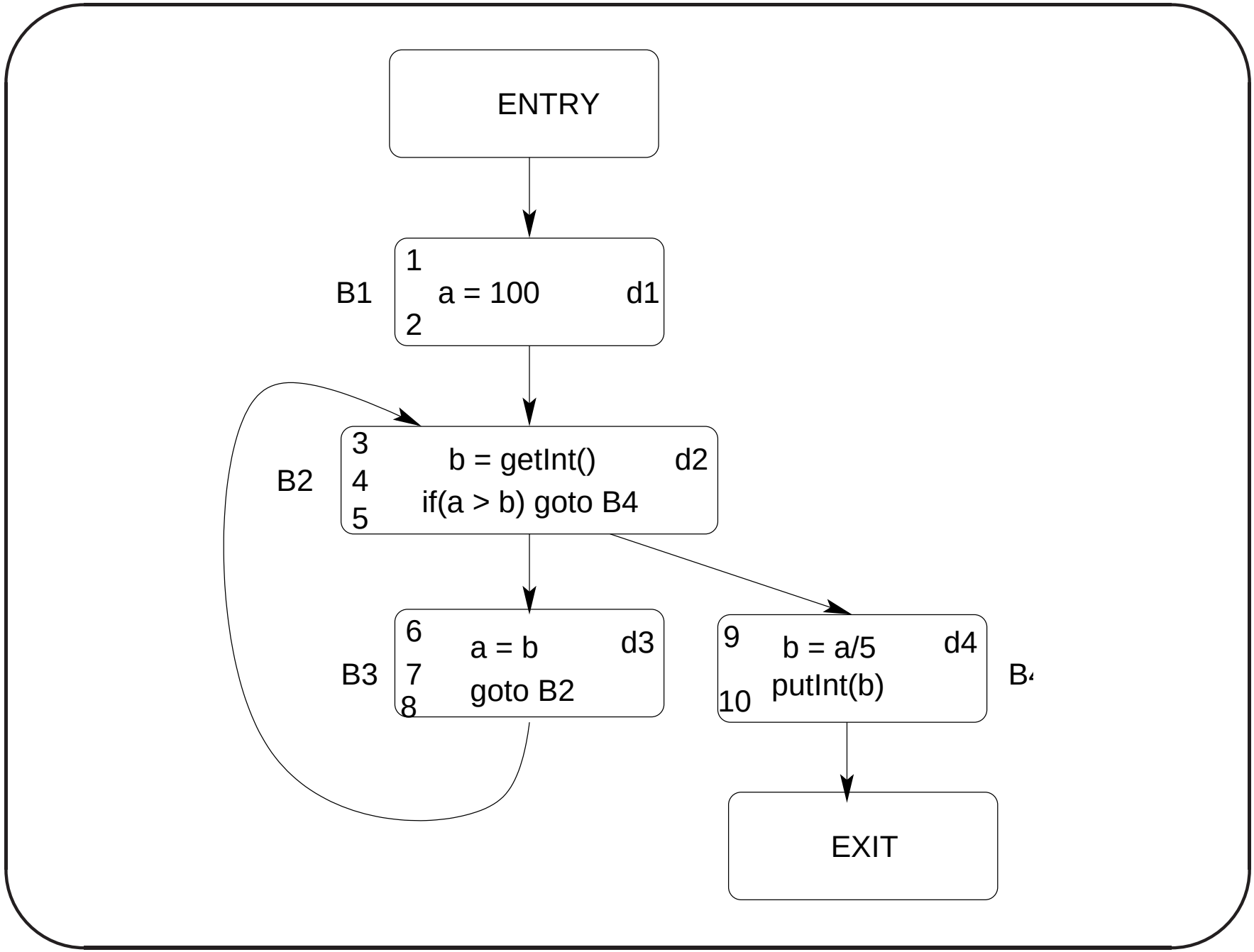
State Sequence and Program Point

- In a backward flow problem, the state after the last statement of a basic block B, is some combination of the states before the first statements of its successor blocks.
- In forward flow the block B transforms its input state to output state. In backward flow it is the reverse.

Execution Path

- A sequence of states or program points satisfying the previous conditions is an execution path.
- There is no upper bound on the number of execution paths.
- A data-flow analysis abstracts out a required finite set of facts from the unbounded number of execution sequences.

Flow Graph and Execution Paths


ENTRY
B1
1
a = 100
d1
2
B2
3
b = getInt()
d2
4
if(a > b) goto B4
5
B3
6
a = b
d3
7
goto B2
8
9
b = a/5
d4
10
putInt(b)
EXIT

Data-Flow Values

- A data-flow value is associated to every program point of the execution sequence.
- Such a value is an abstraction of the set of program states.
- The domain of the data-flow values depends on the goal of the analysis.

Data-Flow Values

- As an example, in reaching definition analysis, we wish to know the subsets of definitions that reach a program point p.
- In a live-variable analysis given a variable x and a program point p, we wish to know whether the value of x will be used in some path starting at p.

Data-Flow Values

- Let the data-flow values before and after a statement s be $IN[s]$ and $OUT[s]$ respectively.
- A data-flow problem is to find a solution of $IN[s]$ and $OUT[s]$ for all states s, under a set of constraints.

Constraints on Data-Flow Values

- There are two types of constraints.
- One is based on the semantics of statements. It is called a transfer function f_s of the statement s.
- The other one depends on the control-flow within basic blocks and across basic block boundaries.

Transfer Functions

- There are two types of transfer functions.
- If the information flows forward along the execution path, then for a statement s and a forward transfer function f_s, $OUT[s] = f_s(IN[s])$.
- If the information flows backward, then for a statement s and a backward transfer function f_s, $IN[s] = f_s(OUT[s])$.

Control-Flow Constraints

- Let the sequence of statements within a basic block (B) be $s_1, s_2, \cdots, s_n$.
- $IN[s_{i+1}] = OUT[s_i], \quad 1 \leq i < n-1$ for a forward flow.

Control-Flow Constraints

- Data-flow within a block is easy to evaluate. It is more complicated at the block boundary due to different nature of information flow.
- Given a block B we define $IN[B]$ and $OUT[B]$ as the data-flow values immediately before and after the block B.

Control-Flow Constraints

- $IN[B] = IN[s_1]$, $OUT[B] = OUT[s_n]$.
- For a forward flow $OUT[B] = f_B(IN[B])$, where $f_B = f_{s_n} \circ \cdots \circ f_{s_2} \circ f_{s_1}$.
- For a backward flow $IN[B] = f_B(OUT[B])$, where $f_B = f_1 \circ f_2 \circ \cdots \circ f_{n-1} \circ f_n$.
- At the beginning and at the end of the block relations are as follows:

Control-Flow Constraints

- For a forward-flow:

$$IN[B] = \bigvee_{C \in \text{pred}(B)} OUT[C],$$

$$OUT[B] = f_B(IN[B]).$$

- For backward-data flow:

$$IN[B] = f_B(OUT[B]),$$

$$OUT[B] = \bigvee_{C \in \text{succ}(B)} IN[C]$$

Note

- The meet operator, '$\bigvee$', depends on the type of analysis. It may be a '$\bigcup$' if the claim is existential e.g. there exists a path from the definition d to the program point p.
- It may be an '$\bigcap$' if the claim is universal e.g. an expression `a*b` is available at a program point p if it is evaluated on every path from ENTRY $\rightarrow p$ without redefining `a` and `b` in between.

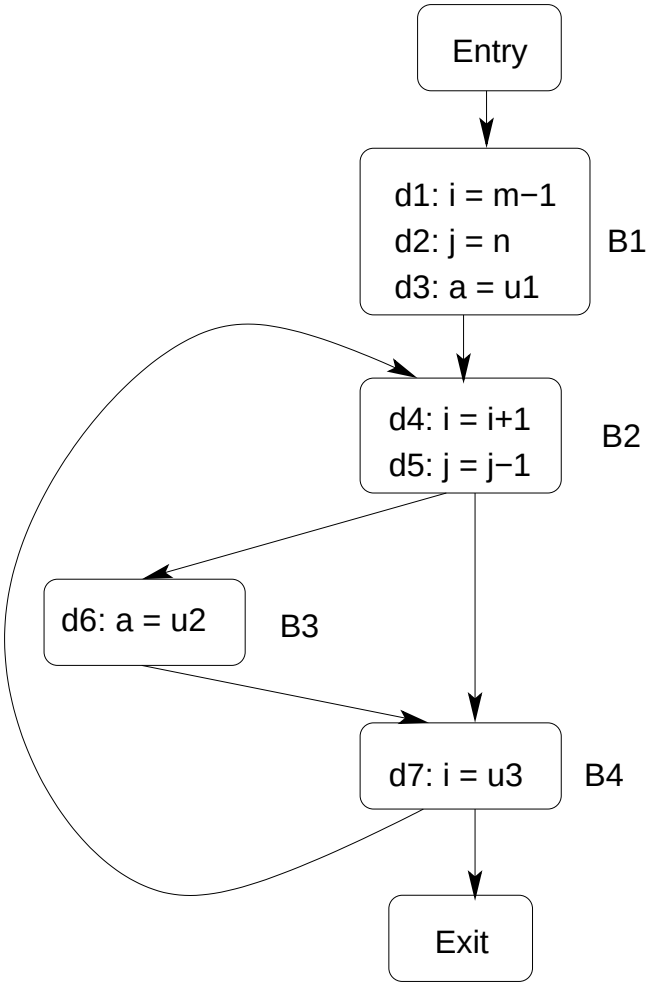
Reaching Definition

- Let d : `x = 10` be a definition of `x` and p be a program point just before `a = 2*x`.
- If d is the only definition of `x` that reaches p, then `x` is a constant at `a = 2*x`.
- If no definition of `x` reaches p, then it is undefined at p.

Reaching Definition

- A definition d reaches a point p, if there is a path from $d \rightarrow p$ on which d is not killed.
- For simplicity we assume that there is no alias of `x`.

An Example


Entry
d1: i = m−1
d2: j = n
d3: a = u1
B1
d4: i = i+1
d5: j = j−1
B2
d6: a = u2
B3
d7: i = u3
B4
Exit

Reaching Definitions: B_2

- d_1, d_2, d_3 reaches B_2.
- d_7 reach the beginning of B_2. It kills d_4, prevents it to reach B_2.
- d_5 reaches B_2.
- d_6 also reaches B_2.

Transfer Functions

- We start with a single statement:
  `d: u = v + w`, where '+' may be taken as a generic binary operator.
- Let x be the set of definitions that reaches the statement.
- The definition d kills all other definitions of `u` in the program. Killed definitions of `u` forms the set kill_d.

Transfer Functions

- It generates the new definition d of `u`, $\text{gen}_d = \{d\}$.
- The transfer function of `d: u = v + w`, f_d, computes the the definitions that reach after it: $f_d(x) = \text{gen}_d \cup (x \setminus \text{kill}_d)$.

Function Composition

If $f_1(x) = \text{gen}_1 \cup (x \setminus \text{kill}_1)$ and
$f_2(x) = \text{gen}_2 \cup (x \setminus \text{kill}_2)$, then

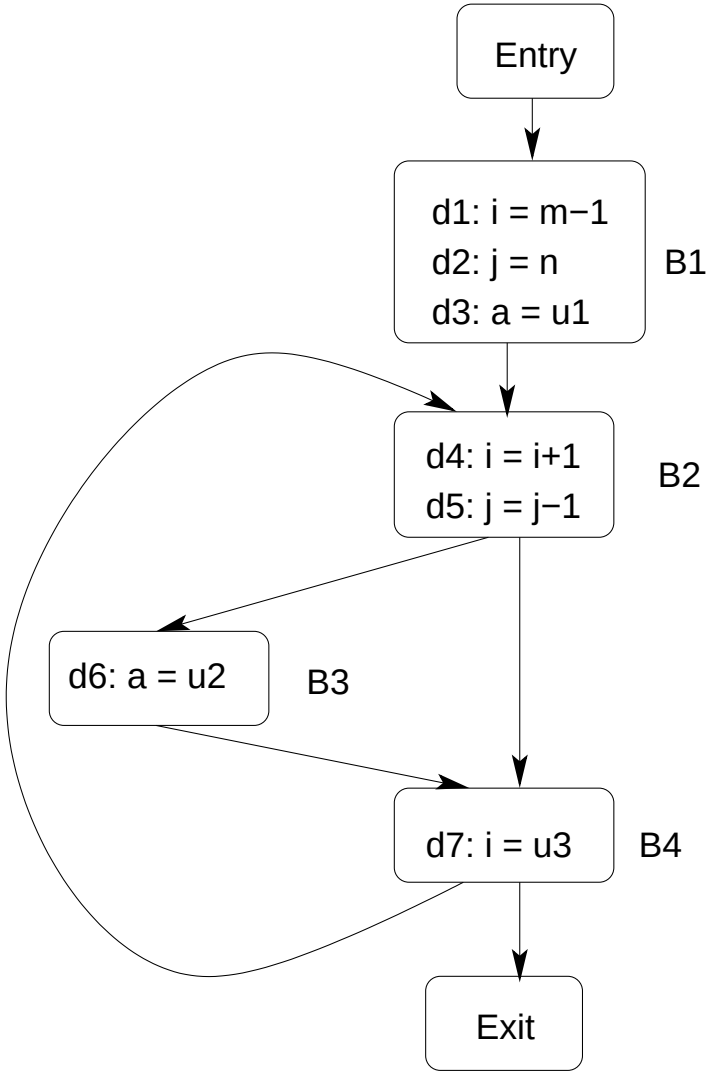
$$\begin{aligned} f_2(f_1(x)) &= \text{gen}_2 \cup ((\text{gen}_1 \cup x \setminus \text{kill}_1) \setminus \text{kill}_2) \\ &= \text{gen}_2 \cup (\text{gen}_1 \setminus \text{kill}_2) \cup ((x \setminus \text{kill}_1) \setminus \text{kill}_2) \end{aligned}$$

Function Composition

If a block B has n statements with transfer functions $f_i(x) = \text{gen}_i \cup (x \setminus \text{kill}_i)$, then the transfer function for the block can be written as $f_B(x) = \text{gen}_B \cup (x \setminus \text{kill}_B)$, where

$$\begin{aligned} \text{gen}_B &= \text{gen}_n \cup (\text{gen}_{n-1} \setminus \text{kill}_n) \cup \\ &\quad (\text{gen}_{n-2} \setminus \text{kill}_{n-1} \setminus \text{kill}_n) \cup \cdots \cup \\ &\quad (\text{gen}_1 \setminus \text{kill}_2 \setminus \text{kill}_3 \setminus \cdots \setminus \text{kill}_n) \\ \text{kill}_B &= \text{kill}_1 \cup \text{kill}_2 \cup \cdots \cup \text{kill}_n \end{aligned}$$

Reaching Definitions: an Example



gen(B1): {d1, d2, d3}

kill(B1): {d4, d5, d6, d7}

gen(B2): {d4, d5}

kill(B2): {d1, d2, d7}

gen(B3): {d6}

kill(B3): {d3}

gen(B4): {d7}

kill(B4): {d1, d4}

Control Flow Constraints

- A definition reaches at the first statement of block B only through some path of its predecessor blocks.
- So the meet operator is union over all predecessors blocks of B.

$$IN[B] = \bigcup_{P \in pred(B)} OUT[P].$$

Reaching Definition: Iterative Algorithm

- Every control-flow graph (CFG) has two empty basic blocks, ENTRY and EXIT.
- The ENTRY block is the starting point of the graph.
- The EXIT block is where all exit out of the graph go.
- The $OUT[ENTRY] = \emptyset$ for the reaching definition problem.

Reaching Definition: Iterative Algorithm

- For all basic blocks B other than ENTRY,

$$OUT[B] = \text{gen}_B \cup (IN[B] \setminus \text{kill}_B)$$

$$IN[B] = \bigcup_{P \in \text{pred}(B)} OUT[P].$$

Reaching Definition: Iterative Algorithm

- These equations can be solved by the following algorithm.
- The output of the algorithm is the least fixed-point of the equations.
- The solutions of $IN[B]$ and $OUT[B]$ produced by the algorithm is contained (subset) in any other solutions of the equations.

Algorithm

1 $OUT[ENTRY] = \emptyset$

2 `for` each basic block $B \neq ENTRY$, $OUT[B] = \emptyset$

3 `while` ($OUT[B]$ for any B changes) `do`

4 `for` (each $B \neq$ ENTRY)

5 $IN[B] = \bigcup_{P \in \text{pred}(B)} OUT[P]$

6 $OUT[B] = \text{gen}_B \cup (IN[B] \setminus \text{kill}_B)$

Note

- Subset of definitions are represented as bit-vectors: the i^{th} bit of $b_1 b_2 \cdots b_n$ corresponds to the i^{th} definition d_i.
- The set union is a bitwise OR operation.
- The set difference $A \setminus B = A \cap \overline{B}$, is bitwise AND of A with the complement of B.

The Example

- There are seven definitions in our example, $\{d_1, d_2, \cdots, d_7\}$.
- At the beginning and at the end of each block we associate a 7-bit vector, $b_1 b_2 \cdots b_7$.
- The bit $b_i,\ i = 1, \cdots, 7$ of a vector V_p of a program point p is '1', if the definition d_i reaches p.
- Otherwise $b_i = 0$.

The Example

- The vector of OUT[ENTRY] is always zero, no definition reaches that point.
- Initially all OUT[B] vectors are zero.
- We iterate through the data till we reach the fixed-point.

The Example

Block(B)	OUT$[B]_0$	IN[B]$_1$	OUT[B]$_1$
B_1	0000 000	0000 000	1110 000
B_2	0000 000	1110 000	0011 100
B_3	0000 000	0011 100	0001 110
B_4	0000 000	0011 110	0010 111
EXIT	0000 000	0010 111	0010 111

An Example

Block(B)	OUT$[B]_1$	IN[B]$_2$	OUT[B]$_2$
B_1	1110 000	0000 000	1110 000
B_2	0011 100	1110 111	0011 110
B_3	0001 110	0011 110	0001 110
B_4	0010 111	0011 110	0010 111
EXIT	0010 111	0010 111	0010 111

An Example

Block(B)	OUT$[B]_2$	IN[B]$_3$	OUT[B]$_3$
B_1	1110 000	0000 000	1110 000
B_2	0011 110	1110 111	0011 110
B_3	0001 110	0011 110	0001 110
B_4	0010 111	0011 110	0010 111
EXIT	0010 111	0010 111	0010 111

The algorithm terminates after the 3^{rd} pass.

Reaching Definitions: an Example

- No definition reaches at the beginning of B_1.
- $d_1, d_2, d_3, d_5, d_6, d_7$ reaches at the beginning of B_2.
- d_3, d_4, d_5, d_6 reaches at the beginning of B_3.
- d_3, d_4, d_5, d_6 reaches at the beginning of B_4.
- d_3, d_5, d_6, d_7 reaches at the beginning of EXIT.

Live Variable Analysis

- Given a program point p and a variable `x`, we wish to know whether the value of `x` at p will be used in some execution path starting at p.
- If the value of `x` is used, then `x` is live at p, otherwise, `x` is dead at p.
- This requires an analysis in the direction opposite to the flow of control.

Register Allocation within Basic Block

- If a value computed in a register is dead at the end of the block, it need not be stored in the memory.
- When all registers are in use, and there is a demand for another register, a register containing dead value can be reused.

Definition

In a basic block B.

- A variable v is a member of the set def_B if it is defined in B before any use of it in B.
- A variable u is a member of the set use_B if it is used in B before any definition of it in B.
- The variables in def_B are dead and the variables in use_B are live at the beginning of B.

Definition

Following equations relate the live variables $IN[B]$ and $OUT[B]$.

$$IN[B] = \text{use}_B \cup (OUT[B] \setminus \text{def}_B).$$

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

Note

- Both reaching definition and live variable analysis has union as the meet operator.
- The information is propagated through paths, and we want to know whether any path has the desired property.
- If a variable is used by any successor S of B, then it must be live at the end of B.

Note

- In a backward-flow problem, we start from the EXIT block. Nothing is live at exit. So we initialize IN[EXIT] $= \emptyset$.
- The roles of $IN[B]$ and $OUT[B]$ are reversed.

Algorithm

1 $IN[EXIT] = \emptyset$

2 `for` each basic block $B \neq EXIT$, $IN[B] = \emptyset$

3 `while` ($IN[B]$ for any B changes) `do`

4 `for` (each $B \neq$ EXIT)

5 $OUT[B] = \bigcup_{S \in \text{succ}(B)} IN[S]$

6 $IN[B] = \text{use}_B \cup (OUT[B] \setminus \text{def}_B)$

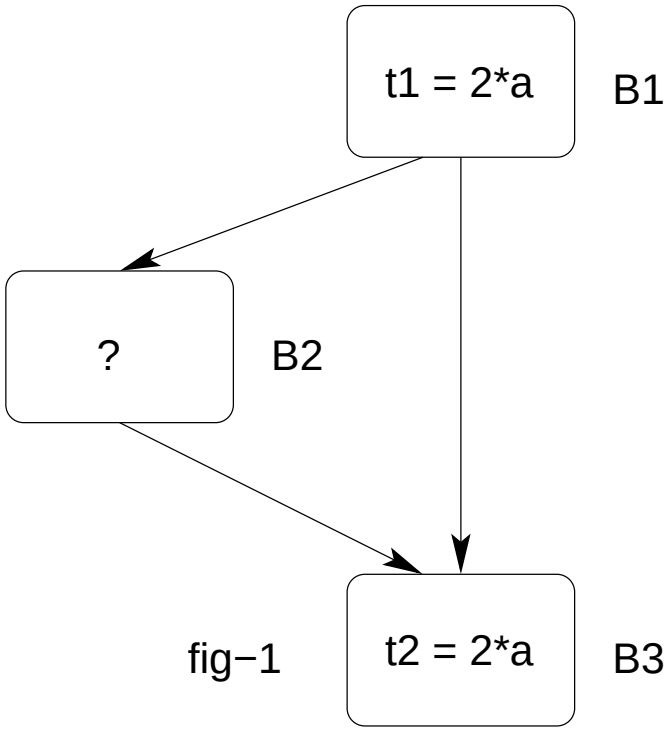
Available Expression

An expression `a*b` is available at a program point p if every path from the ENTRY to p evaluates `a*b`. And in between an evaluation and the point p, there is no modification of `a` or `b`.

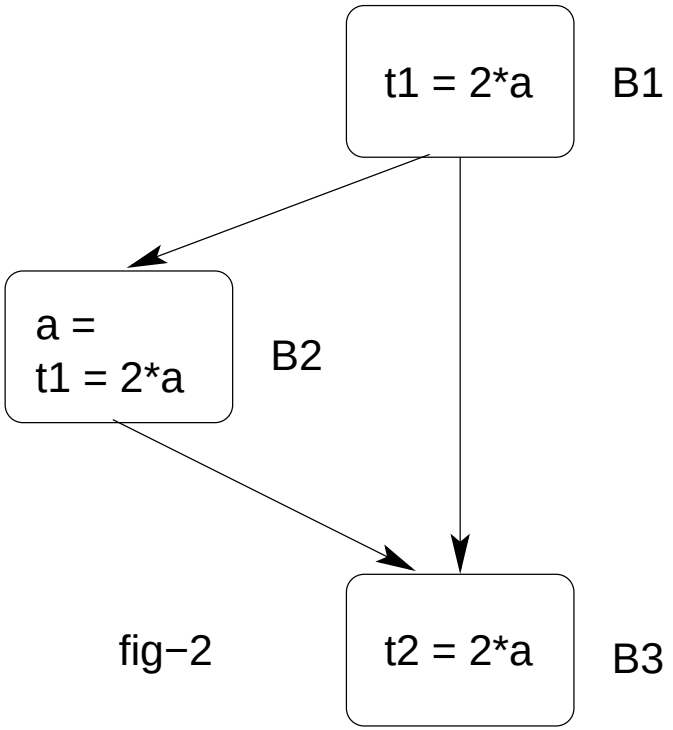
Available Expression

- A block kills an expression `a*b` if it modifies `a` or `b` but subsequently does not recompute `a*b`.
- A block generates an expression `a*b` if it evaluates `a*b`, and subsequently does not modifies `a` or `b`.

Global Common Subexpressions



fig−1



fig−2

Available Expression

- `2*a` is a common subexpression in B_3 if `2*a` is available at the beginning of the block.
- It will be available if `a` is not assigned a new value in B_2 or `2*a` is recomputed after `a` is assigned a value.

Available Expression: Computation in a Block

- At the beginning the set of common expressions is empty.
- Let S be the set of available expressions at a point p and q is the next point after the statement `a = b+c`.
- The set of common expressions at q is computed as follows.

Available Expression: Computation in a Block

- Add {b+c} to S.
- Remove all expressions involving a from S.
- Add and Delete must be done in the given order as b or c may be same as a.
- Killed expressions are of the form b+c where b or c are defined in the block, but b+c is not computed.

An Example

Statement	Available Expression	Reason
	$\emptyset$	
`a = b + c`		
	`b + c`	
`b = a - d`		kills `b + c`
	`a - d`	
`c = b + c`		
	`a - d`	
`d = a - d`		kills `a - d`
	$\emptyset$	

Available Expression: Data

- U is the set of expressions appearing in the right side of any 3-address code of a function.
- $IN[B]$ is a subset of U available at the beginning of a basic block B.
- $OUT[B]$ is the subset of U available after the last statement of B.
- eGen_B and eKill_B are the expressions generated and killed in B.

Data-Flow Equations

- No available expression at ENTRY:
  OUT[ENTRY] $= \emptyset$.
- For every basic block other than ENTRY,

$$IN[B] = \bigcap_{P \in \text{pred}(B)} OUT[P].$$

Data-Flow Equations

- An expression is available at the beginning of B if it is available on every path to B. So the meet operator is an intersection ('$\bigcap$').
- $$OUT[B] = \text{eGen}_B \cup (IN[B] \setminus \text{eKill}_B)\,.$$

Note

- The behavior of the equations of available expression and reaching definitions are different due to different meet operator.
- The equations of reaching definitions computes the least fixed point, but the equations of available expression computes the greatest fixed point.

Note

- An empty set of available expressions as the initial values of $OUT[B]$ for every block B does not give us anything. The intersection with the empty set is an empty set.
- We need to find the largest set of available expressions at the beginning of each block

Note

- Starting from an empty set, the iterative algorithm computes the set of definitions d that have paths to the beginning of the block B.
- On the contrary, available expression starts with the assumption that all all expressions are available at the beginning of the block B, unless killed on any path of one of its predecessor.

Algorithm

1 $\mathrm{OUT[ENTRY]} = \emptyset$

2 `for` each basic block B, $B \neq \mathrm{ENTRY}$, $OUT[B] = U$

3 `while` ($OUT[B]$ for any B changes) `do`

4 `for` (each $B \neq \mathrm{ENTRY}$)

5 $IN[B] = \bigcap_{P \in \mathrm{pred}(B)} OUT[B]$

6 $OUT[B] = \mathrm{eGen}_B \cup (IN[B] \setminus \mathrm{eKill}_B)$