Introduction

- Code optimization (or code improvement)
  - Eliminating unnecessary instructions in object code
  - Replacing one sequence of instructions by a faster sequence of instructions that does the same thing
- Global optimizations
  - Performed on a flow graph as a whole, involving multiple basic blocks
  - Most global optimizations are based on data-flow analysis
- A compilers knows only how to apply relatively low-level semantic transformations to optimize code
  - High-level optimizations: architectural/algorithmic changes, refactoring, etc.

What is Data-Flow Analysis?

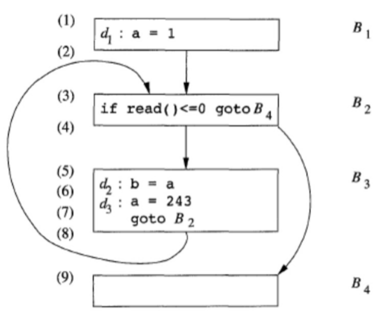
- A body of techniques that derive information about the flow of data along program execution paths
  - **Example 1:** Whether two textually identical expressions evaluate to the same value along any execution path? (Identify common subexpressions)
  - **Example 2:** Whether the result of an assignment is used on subsequent execution paths? (Eliminate dead code)
- Data-flow analysis is the foundation of many optimizations

判断公共子表达式

消除死代码

The Data-Flow Abstraction

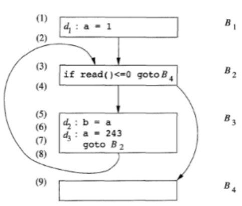
- **Program points**: points before and after each statement
  - Within one block, the program point after a statement is the same as the program point before the next statement (e.g., 6)
  - If there is an edge from block B to block C, then the program point after the last statement of B may be followed immediately by the program point before the first statement of C (e.g., 8 and 3)



程序点在每条statement的前后

若B→C，则B的最后一个程序点后紧接着C的第一个程序点.

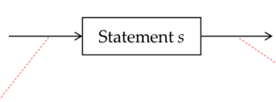
- An **execution path** from point p_1 to point p_n is a sequence of points $p_1, p_2, \ldots, p_n$ such that for each $i = 1, 2, \ldots, n$-1:
  - either p_i is the point immediately preceding a statement and p_{i+1} is the point immediately follow that statement (the "within block" case)
  - or p_i is the end of a block and p_{i+1} is the beginning of a successor block (the "across block" case)



The shortest complete execution path: (1, 2, 3, 4, 9)

The next shortest path executing one iteration of the loop: (1, 2, 3, 4, 5, 6, 7, 8, 3, 4, 9)

- Program execution can be viewed as *a series of transformations of the program state* (the values of all variables)
  - Each execution of a statement transforms an input state to a new output state



Input state (associated with the program point before s)

Output state (associated with the program point after s)

程序执行即为一系列的状态转移

- When analyzing a program, we must consider all the possible execution paths through a flow graph
  - In general, there is an infinite number of possible paths due to the existence of loops and recursions

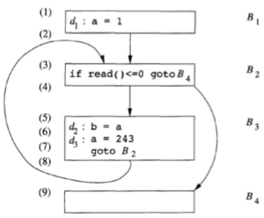
循环、递归会导致无穷多的可能路径.

- Data-flow analyses summarize all the possible program states that can occur at a program point with a finite set of facts
  - Different analyses may choose to abstract out different information (i.e., obtaining approximations)
  - In general, no analysis is necessarily a perfect representation of the state

总结所有可能的状态

Example

到达定值



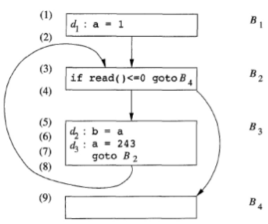
Reaching definitions:

- The first time point (5) is executed, the value of a is 1, i.e., definition d_1 reaches (5) in the first iteration
- In subsequent iterations, d_3 reaches point (5) and the value of a is 243
- So, we may summarize all the program states at (5) by saying that the value of a is one of {1, 243} and defined by one of $\{d_1, d_3\}$

Static approximation.
The exact value depends on the execution path at runtime.

常量折叠

无论什么执行路径,在某个程序点时变量定义唯一.

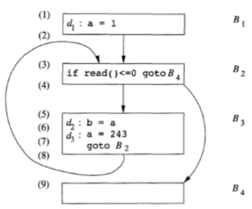


Constant folding:

- Find those definitions that are the unique definition of their variable to reach a given program point, regardless of the execution paths
- For this task, we describe some variables as "constant" or "not a constant"
  - a is not a constant at point (5) and thus cannot be replaced by a constant value

The Data-Flow Analysis Schema

- We associate with every program point a *data-flow value* that represents an abstraction of program states observed for that point*



Reaching definitions:
The data-flow value at (5) is the set of a's definitions that can reach (5), that is, $\{d_1, d_3\}$

Constant folding:
The data-flow value for variable a at (5) is NAC (not a constant)

* In each application of data-flow analysis, the set of possible data-flow values is the domain for the application.

- IN[s]: The data-flow value before the statement s
- OUT[s]: The data-flow value after the statement s
- The *data-flow problem* is to find a solution to a set of constraints on the IN[s]'s and OUT[s]'s for all statements s
  1. Constraints based on the semantics of the statements ("*transfer functions*")
  2. Constraints based on the flow of control

Statement exeuctions may alter data-flow values

Control flows propagate data-flow values

Transfer Functions (传递函数)

- The relationship between the data-flow values before and after each statement is known as a *transfer function*
- In a forward-flow problem (information propagate forward along execution paths), the transfer function f_s takes the data-flow value before the statement s and produces a new data-flow value after s
  - $\mathrm{OUT}[s] = f_s(\mathrm{IN}[s])$
- In a backward-flow problem (information flow backwards up the execution paths), the transfer function f_s takes the data-flow value after the statement s and produces a new data-flow value before s
  - $\mathrm{IN}[s] = f_s(\mathrm{OUT}[s])$

Control-Flow Constraints

1. Within a basic block of statements $s_1, s_2, \ldots, s_n$, the data-flow value out of s_i is the same as that into s_{i+1}
   - $\mathrm{IN}[s_{i+1}] = \mathrm{OUT}[s_i]$, for all $i = 1, 2, \ldots, n-1$
2. Control-flow edges between basic blocks may create more complex constraints
   - For example, in reaching definitions analysis, the set of definitions reaching the leader statement of a block should be <u>the union of the definitions</u> after the last statements of each of the predecessor blocks

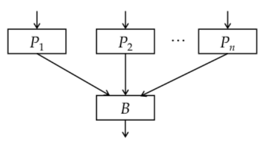
Data-Flow Schema on Basic Blocks (The Intra-Block Case)

- Within a basic block, control flows from the beginning to the end without interruption or branching
- For a block B consisting of statements $s_1, s_2, \ldots, s_n$, we have*
  - $\mathrm{IN}[B] = \mathrm{IN}[s_1]$
  - $\mathrm{OUT}[B] = \mathrm{OUT}[s_n]$
  - $\mathrm{OUT}[B] = f_B(\mathrm{IN}[B])$, where $f_B = f_{s_n} \circ \ldots \circ f_{s_2} \circ f_{s_1}$ (composing statement-level transfer functions)

函数组合

* For backward-flow problem, $\mathrm{IN}[B] = f_B(\mathrm{OUT}[B])$, where $f_B = f_{s_1} \circ f_{s_2} \circ \ldots \circ f_{s_n}$

Data-Flow Schema on Basic Blocks (The Inter-Block Case)



Forward-flow problem:

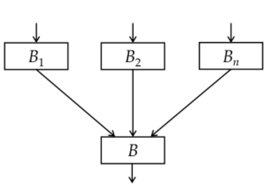
$$\mathrm{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \mathrm{OUT}[P]$$

Backward-flow problem:

$$\mathrm{OUT}[B] = \bigcup_{S \text{ a successor of } B} \mathrm{IN}[S]$$

$\cup$ is a generic *meet operator* depending on specific problems.

Example (Constant Folding Problem)



If:

$\mathrm{OUT}[B_1]$: x: 3; y: 4; z: NAC

$\mathrm{OUT}[B_2]$: x: 3; y: 5; z: 7

$\mathrm{OUT}[B_3]$: x: 3; y: 4; z: 7

then:

$\mathrm{IN}[B]$: x: 3; y: NAC; z: NAC

Solutions to Data-Flow Equations

- Data-flow equations usually do not have a unique solution
  - All data-flow schemas compute approximations to the ground truth
- Our goal is to find the most "precise" solution that satisfies both control-flow and transfer constraints
  - Being precise enables valid code improvements (in general, a higher precision leads to a better improvement)
  - Satisfying constraints guarantees the safety of transformations (does not change program semantics)

没有唯一解

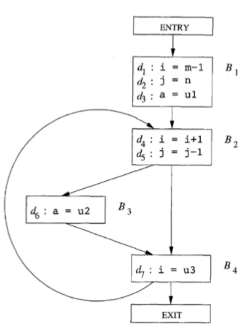
只能找到最精确的角

Reaching Definitions

- A definition d of some variable x *reaches* a point p if there is a path from the program point after d to p, such that d is not "killed" along that path
  - d is *killed* if there is any other definition of x along the path
  - Intuitively, if d reaches the point p, then d might be the last definition of x

也即之后不再有定义

Example



Which definitions reach the block B_2?

- d_1, d_2, d_3 reach B_2
- d_5 reaches B_2 since there is no other definition to j in the loop
- d_4 does not reach B_2 since i is always redefined by d_7 (i.e., d_7 reaches B_2)
- d_6 reaches B_2

Reaching Definitions

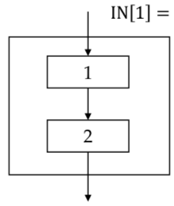
- For reaching definitions analysis, we allow inaccuracies. However, the decisions should be *"safe"*.
  - It is ok that some inferred reaching defs cannot actually reach a point
  - However, those defs that can reach a point should be identified
  - In other words, over-approximations are acceptable
- ***Reason of inaccuracies:*** In general, to decide whether each path in a flow graph can be taken is an undecidable problem*
  - We often simply assume that every path in the flow graph can be followed in some execution of the program

* This is the well-known *path feasibility problem*.

Transfer Equations

- Consider a statement `d: u = v + w`
  - It *generates* a definition d of variable u and *kills* all other definitions of u
- Transfer function of a statement is in *gen-kill* form:
  - $f_d(x) = gen_d \cup (x - kill_d)$
    - x is the data-flow value (reaching definitions) before the statement
    - gen_d is the set of generated definitions, i.e., $\{d\}$
    - $kill_d$ is the set of killed definitions, i.e., all other definitions of u

- Transfer function of a basic block is also in *gen-kill* form
- Consider a block of only two statements:



$\text{IN}[1] = x$

$\text{OUT}[1] = f_1(x) = gen_1 \cup (x - kill_1)$

$\text{OUT}[2] = f_2(\text{IN}[2]) = f_2(\text{OUT}[1])$

$= gen_2 \cup (\text{OUT}[1] - kill_2)$

$= gen_2 \cup ((gen_1 \cup (x - kill_1)) - kill_2)$

$= gen_2 \cup (gen_1 - kill_2) \cup (x - (kill_1 \cup kill_2))$

2中新生成　1中新生成减去2中杀掉　进入1之前的减去1与2中杀掉的

- Generalize to a block B with n statements, we have:

$$f_B(x) = gen_B \cup (x - kill_B)$$

where

$$kill_B = kill_1 \cup kill_2 \cup \cdots \cup kill_n$$

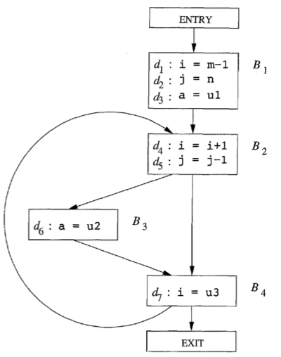
and

$$gen_B = gen_n \cup (gen_{n-1} - kill_n) \cup (gen_{n-2} - kill_{n-1} - kill_n) \cup \ldots \cup (gen_1 - kill_2 - kill_3 - \cdots - kill_n)$$

> ***$kill_B$***
> The *kill set* is the union of all the defs killed by the individual statements

> ***gen_B***
> The *gen set* contains all the defs inside the block that are *downward exposed* (visible immediately after the block)

Example



$gen_{B_1} = \{d_1, d_2, d_3\}$
$kill_{B_1} = \{d_4, d_5, d_6, d_7\}$

$gen_{B_2} = \{d_4, d_5\}$
$kill_{B_2} = \{d_1, d_2, d_7\}$

$gen_{B_3} = \{d_6\}$
$kill_{B_3} = \{d_3\}$

$gen_{B_4} = \{d_7\}$
$kill_{B_4} = \{d_1, d_4\}$

Control-Flow Equations

$$\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P]$$

The *meet operator*

- **Rationale:**
  - A definition reaches a program point as long as there exists at least one path along which the definition reaches. This explains why $\text{OUT}[P] \subseteq \text{IN}[B]$.
  - Since a definition cannot reach a point unless there is a path along which it reaches, $\text{IN}[B]$ needs to be no larger than the union of the reaching definitions of all the predecessor blocks

Problem Formulation

- The reaching definitions problem is defined by the following equations:
  - For the ENTRY block: $\text{OUT}[\text{ENTRY}] = \emptyset$
    - Since no definitions reach the beginning of a program
  - For all basic blocks B other than ENTRY:
    - $\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B)$
    - $\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P]$

The Iterative Algorithm

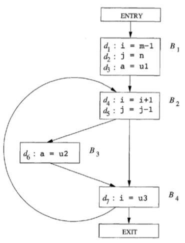
- **Input:** A flow graph with $kill_B$ and gen_B computed for each block B
- **Output:** IN[B] and OUT[B] for each block B

$\text{OUT}[\text{ENTRY}] = \emptyset$;
for (each basic block B other than ENTRY) $\text{OUT}[B] = \emptyset$;
while (changes to any OUT occur)
  for (each basic block B other than ENTRY) {
    $\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P]$;
    $\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B)$;
  }

Example

Example

$\text{OUT}[\text{ENTRY}] = \emptyset$;
for (each basic block B other than ENTRY) $\text{OUT}[B] = \emptyset$;
while (changes to any OUT occur)
  for (each basic block B other than ENTRY) {
    $\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P]$;
    $\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B)$;
  }



$gen_{B_1} = \{ d_1, d_2, d_3 \}$
$kill_{B_1} = \{ d_4, d_5, d_6, d_7 \}$

$gen_{B_2} = \{ d_4, d_5 \}$
$kill_{B_2} = \{ d_1, d_2, d_7 \}$

$gen_{B_3} = \{ d_6 \}$
$kill_{B_3} = \{ d_3 \}$

$gen_{B_4} = \{ d_7 \}$
$kill_{B_4} = \{ d_1, d_4 \}$

Initialization:

OUT[ENTRY] = ∅

OUT[B_1] = ∅

OUT[B_2] = ∅

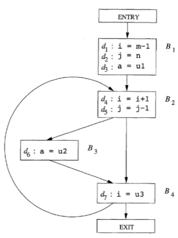
OUT[B_3] = ∅

OUT[B_4] = ∅

OUT[EXIT] = ∅

Example

$\text{OUT}[\text{ENTRY}] = \emptyset$;
for (each basic block B other than ENTRY) $\text{OUT}[B] = \emptyset$;
while (changes to any OUT occur)
  for (each basic block B other than ENTRY) {
    $\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P]$;
    $\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B)$;
  }



$gen_{B_1} = \{ d_1, d_2, d_3 \}$
$kill_{B_1} = \{ d_4, d_5, d_6, d_7 \}$

$gen_{B_2} = \{ d_4, d_5 \}$
$kill_{B_2} = \{ d_1, d_2, d_7 \}$

$gen_{B_3} = \{ d_6 \}$
$kill_{B_3} = \{ d_3 \}$

$gen_{B_4} = \{ d_7 \}$
$kill_{B_4} = \{ d_1, d_4 \}$

Iteration #1:

OUT[ENTRY] = ∅

IN[B_1] = ∅
OUT[B_1] = $\{d_1, d_2, d_3\}$

IN[B_2] = $\{d_1, d_2, d_3\}$
OUT[B_2] = $\{d_3, d_4, d_5\}$

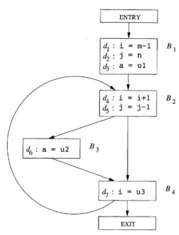
IN[B_3] = $\{d_3, d_4, d_5\}$
OUT[B_3] = $\{d_4, d_5, d_6\}$

IN[B_4] = $\{d_3, d_4, d_5, d_6\}$
OUT[B_4] = $\{d_3, d_5, d_6, d_7\}$

IN[EXIT] = $\{d_3, d_5, d_6, d_7\}$
OUT[EXIT] = $\{d_3, d_5, d_6, d_7\}$

Example

$\text{OUT}[\text{ENTRY}] = \emptyset$;
for (each basic block B other than ENTRY) $\text{OUT}[B] = \emptyset$;
while (changes to any OUT occur)
  for (each basic block B other than ENTRY) {
    $\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P]$;
    $\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B)$;
  }



$gen_{B_1} = \{ d_1, d_2, d_3 \}$
$kill_{B_1} = \{ d_4, d_5, d_6, d_7 \}$

$gen_{B_2} = \{ d_4, d_5 \}$
$kill_{B_2} = \{ d_1, d_2, d_7 \}$

$gen_{B_3} = \{ d_6 \}$
$kill_{B_3} = \{ d_3 \}$

$gen_{B_4} = \{ d_7 \}$
$kill_{B_4} = \{ d_1, d_4 \}$

Iteration #2:

OUT[ENTRY] = ∅

IN[B_1] = ∅
OUT[B_1] = $\{d_1, d_2, d_3\}$

IN[B_2] = $\{d_1, d_2, d_3, d_5, d_6, d_7\}$
OUT[B_2] = $\{d_3, d_4, d_5, d_6\}$

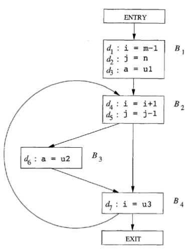
IN[B_3] = $\{d_3, d_4, d_5, d_6\}$
OUT[B_3] = $\{d_4, d_5, d_6\}$

IN[B_4] = $\{d_3, d_4, d_5, d_6\}$
OUT[B_4] = $\{d_3, d_5, d_6, d_7\}$

IN[EXIT] = $\{d_3, d_5, d_6, d_7\}$
OUT[EXIT] = $\{d_3, d_5, d_6, d_7\}$

Example

$\text{OUT}[\text{ENTRY}] = \emptyset$;
for (each basic block B other than ENTRY) $\text{OUT}[B] = \emptyset$;
while (changes to any OUT occur)
  for (each basic block B other than ENTRY) {
    $\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P]$;
    $\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B)$;
  }



$gen_{B_1} = \{ d_1, d_2, d_3 \}$
$kill_{B_1} = \{ d_4, d_5, d_6, d_7 \}$

$gen_{B_2} = \{ d_4, d_5 \}$
$kill_{B_2} = \{ d_1, d_2, d_7 \}$

$gen_{B_3} = \{ d_6 \}$
$kill_{B_3} = \{ d_3 \}$

$gen_{B_4} = \{ d_7 \}$
$kill_{B_4} = \{ d_1, d_4 \}$

Iteration #3: fixed point!

OUT[ENTRY] = ∅

IN[B_1] = ∅
OUT[B_1] = $\{d_1, d_2, d_3\}$

IN[B_2] = $\{d_1, d_2, d_3, d_5, d_6, d_7\}$
OUT[B_2] = $\{d_3, d_4, d_5, d_6\}$

IN[B_3] = $\{d_3, d_4, d_5, d_6\}$
OUT[B_3] = $\{d_4, d_5, d_6\}$

IN[B_4] = $\{d_3, d_4, d_5, d_6\}$
OUT[B_4] = $\{d_3, d_5, d_6, d_7\}$

IN[EXIT] = $\{d_3, d_5, d_6, d_7\}$
OUT[EXIT] = $\{d_3, d_5, d_6, d_7\}$