Data-Flow Analysis

The algorithm to compute Dominators is an example of an Iterative Data-Flow Analysis Algorithm.
- Initialize all the nodes to a given value;
- Visit nodes in some order;
- Calculate the node's value;
- Repeat until no value changes (fixed-point computation).

See examples in the slides 46 – 54

Data-Flow Analysis

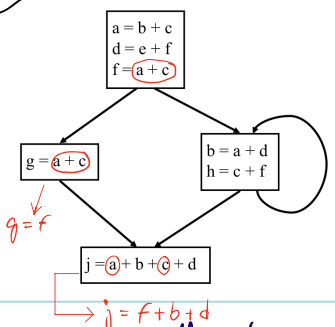
A collection of techniques for compile-time reasoning about the runtime flow of values in a program

- Local Analysis
  - Analyze the "effect" of each Instruction in each Basic Block
  - Compose "effects" of instructions to derive information from beginning of basic block to each instruction
- Data-Flow Analysis
  - Iteratively propagate basic block information over the control-flow graph until no changes
  - Calculate the final value(s) at the beginning/end of the Basic Block
- Local Propagation
  - Propagate the information from the beginning/end of the Basic Block to each instruction

Available Expression

- An Expression is *Available* at point *p* if and only if
  - All paths of execution reaching the current point *p* pass through the point where the expression was defined (the definition point)
  - No variable used in the expression was modified between the definition point and the current point *p*
- In other words: Expression is still *current* at p
- Why is this a Data-Flow Problem?
  - We have to "know" a property about the program's execution that depends on the control-flow of the program!
  - All-Paths or At-Least-One-Path Issue.

Use of Available Expressions



Gen Set

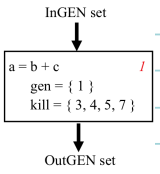
- If a Basic Block (or instruction) defines the expression then the expression number is in the Gen Set for that Basic Block (or instruction)

Kill Set

- If a Basic Block (or instruction) (re)defines a variable in the expression then that expression number is in the Kill Set for that Basic Block (or instruction)
- Expression is thus not valid after that Basic Block (or instruction)

Aggregate Gen Set

- An expression in the Gen Set in the current instruction should be in the OutGEN Set
- Any expression in the InGEN Set that is not killed should be in the OutGEN Set



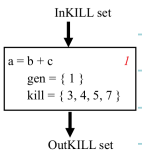
OutGEN = gen ∪ (InGEN - kill)

Aggregate Kill Set

- An expression in the Kill Set in the current instruction should be in the OutKILL set
- Note that different instruction define differently numbered expressions

⇒ expressions are unique

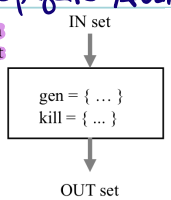
- Any expression in the InKILL set should be in OutKILL



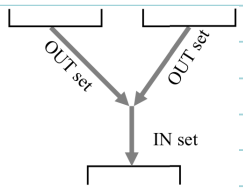
OutKILL = kill ∪ InKILL

Propagate Available Expression Set

- If the expression is generated (in the Gen set) then it is available at the end
  - should be in the OUT set
- Any expression available at the input (in the IN set) and not killed should be available at the end



- Expression is available only if it is available in *All Input Paths*



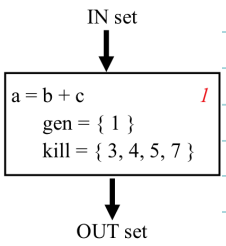
IN = ⋂ OUT

OUT = gen ∪ (IN - kill)

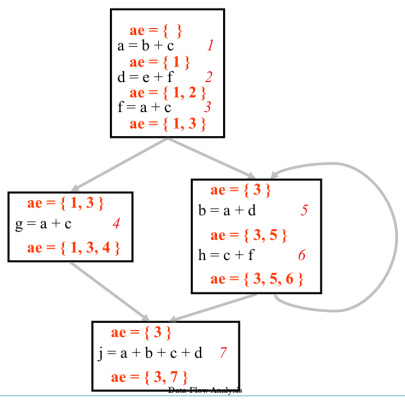
OUT = gen ∪ (IN - kill)

Propagate within the Basic Block

- Start with the IN set of available expressions
- Linearly Propagate Information down the basic block
  - same as a data-flow step
  - single pass since no back edges



OUT = gen ∪ (IN - kill)



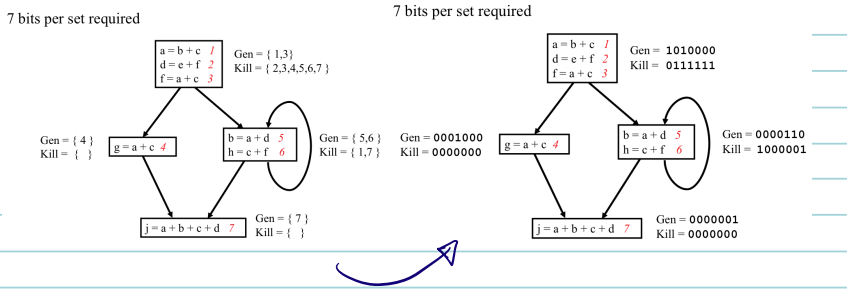
Algorithm for Available Expression

- Assign a Number to each Expression in the Program
- Calculate Gen and Kill Sets for each Instruction
- Derive **aggregate** Gen/Kill Sets for each Basic Block
- Initialize Available Expression Sets at each Basic Block to be the entire Set (Universe)
- Iteratively propagate Available Expression Sets over the CFG until it reaches a Solution
- Propagate Solution within the Basic Block

Entire algorithm with an example between the slides 64 and 140.

Pratical Issues: Bit Sets

- Assign a bit to each element of the set
  - Union ⇒ bit OR
  - Intersection ⇒ bit AND
  - Subtraction ⇒ bit NEGATE and AND
- Fast implementation
  - 32 elements packed to each word
  - AND and OR are single instructions



First Summary

- Overview of Control-Flow Analysis
- Available Expressions Data-Flow Analysis Problem
- Algorithm for Computing Available Expressions
- Practical Issues: Bit Sets

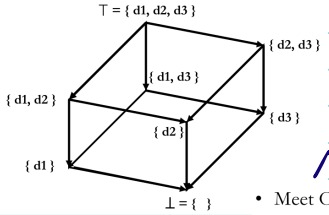
Iterative Data-Flow Formulation

The Basics

- Semi-Lattice/Lattice
  - Abstract quantities V over which the analysis will operate
  - Two operations meet ($\wedge$) and join ($\vee$) over values of V
  - A top value ($\top$) and a bottom value ($\bot$)
  - Example: Sets of Available Expressions and Intersection
- Transfer Functions
  - How each instruction (statement) and control-flow construct affects the abstract quantities V
    Example: the OUT equation for each statement
- Merging of Control-Flow Paths
  - Combining Operator of Data-Flow "meet" Operation
  - Typically Union or Intersection
  - *not the same as the lattice "meet" or "join"...*

Lattice

- A (True) Lattice L consists of
  - A Set of Values V
  - Two operations meet ($\wedge$) and join ($\vee$)
  - A top value ($\top$) and a bottom value ($\bot$)



Meet and Join Operators

- Meet Operation: Greatest Lower Bound - GLB({x,y})
  - Example: Set Intersection
  - Follow the lines in L downwards from the two elements in the lattice until they meet at a single unique element of lattice.
- Join Operation: Lowest Upper Bound - LUB({x,y})
  - Example: Set Union
  - There is a unique element in the lattice from where there is a downwards path (with no shared segment) to both elements (same as a reverse path to a lowest common ancestor in the lattice).

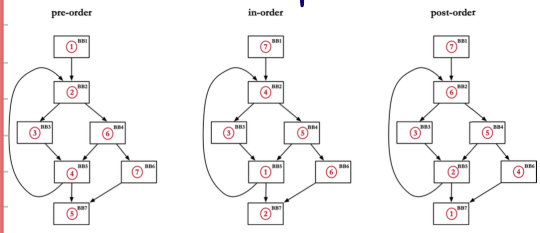
Iterative Algorithm for Data-Flow

INPUT: A data-flow framework with the following components:

1. A control-flow graph (CFG), with specially labeled *Entry* and *Exit* nodes,
2. A direction of the data-flow D,
3. A set of values V,
4. A meet operator Λ,
5. A set of functions F, where $f_B \in F$ is the transfer function for block B, and
6. A constant value v_{entry} or $v_{exit} \in V$, representing the boundary condition for forward and backward frameworks, respectively.

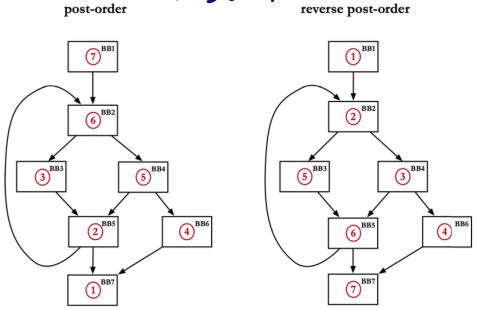
OUTPUT: Values in V for IN[B] and OUT[B] for each block B in the data-flow graph.

Order and Speed



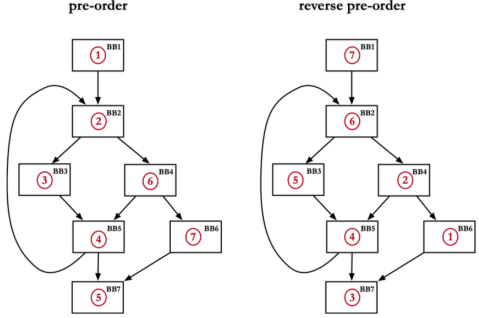
- Which Order for Node's Computation is Best?
  - Forward Problems vs Backwards Problems

For Forward Problems



- Reverse Post-Order (RPO) ensures:
  - For every node, all its predecessors are computed beforehand
  - Data Still Flow Forwards

For Backward Problems



- Reverse Pre-Order ensures:
  - For every node, all its successors are computed beforehand
  - Data Still Flow Backwards

Meaning of a Data-Flow Solution

- **Ideal Solution:**
  - Solution for every block B is found by tracing and applying the transfer function of all possible execution paths leading from the program entry to the beginning of B.

$$P = \text{ENTRY} \rightarrow B_1 \rightarrow B_2 \rightarrow \ldots \rightarrow B_{k-1} \rightarrow B_k$$

$$\text{IDEAL}[B] = \bigwedge_{P,\ \text{a possible path from ENTRY to } B} f_P(v_{\text{ENTRY}}).$$

  - A path is "possible" only if there is some computation of the program that follows exactly that path.
  - The ideal solution would then compute the data-flow value at the end of each possible path and apply the meet operator to these values to find their greatest lower bound.
  - Then no execution of the program can produce a smaller value for that program point.
  - The bound is tight; there is no greater data-flow value that is a glb for the value computed along every possible path to B in the flow graph.
- **Meet Over All Paths Solution:**
  - We assume that every path in the flow graph can be taken.
  - We define the meet-over-paths solution for B to be

$$\text{MOP}[B] = \bigwedge_{P,\ \text{a path from ENTRY to } B} f_P(v_{\text{ENTRY}})$$

  - The paths considered in the MOP solution are a superset of all the paths that are possibly executed.
  - Thus, the MOP solution meets together not only the data-flow values of all the executable paths, but also additional values associated with the paths that cannot possibly be executed.
  - Taking the meet of the ideal solution plus additional terms cannot create a solution larger than the ideal.
  - Thus, for all B we have MOP[B] ≤ IDEAL[B], and we will simply say that MOP ≤ IDEAL.

Def-Use and Use-Def Chains

- Def-Use (DU) Chain
  - Connects a definition of each variable v_k to all the possible uses of that variable
  - A definition of v_k at point p reaches point q if there is a path from p to q where v_k is not redefined.
- Use-Def (UD) Chain
  - Connects a use of a variable v_k to all the possible definitions of that variable

Second Summary

Apenas uma passagem rápida

- Formulating a Data-Flow Analysis Problem
- DU Chains
- SSA Form

→ não sei → bastante fácil

See DV Example

Traditional Optimizations

Available Expressions

- Domain
  - Set of Expressions
- Data-Flow Direction
  - Forward: Out values computed based on In values
- Data-Flow Functions:
  - OUT = gen ∪(IN - kill)
  - gen = { exp | exp is calculated in the Basic Block }
  - kill = { exp | ∃a variable v ∈ exp that is defined in the Basic Block }
- Meet Operation
  - IN = ⋂ OUT for all the predecessors of a Basic Block
- Initial values
  - Empty Set

DU-Chain (Reaching Definitions)

- Domain
  - Set of definitions
- Data-Flow Direction
  - Forward: Out values computed based on In values
- Data-Flow Transfer Function
  - OUT = Gen ∪(IN - Kill)
  - Gen = { x | x is defined in the Basic Block/Statement }
  - Kill = { x | LHS var. of x is redefined in the Basic Block/Statement }
- Meet Operation
  - IN = ⋃ OUT for all the predecessors of a Basic Block
- Initial Values
  - Empty set

Algebraic Simplification

- Apply our knowledge from algebra, number theory etc. to simplify expressions
- Example

a + 0	⇒ a	a ^ 2	⇒ a*a
a * 1	⇒ a	a * 2	⇒ a + a
a / 1	⇒ a	a * 8	⇒ a << 3
a * 0	⇒ 0	a ∧ true	⇒ a
0 - a	⇒ -a	a ∧ false	⇒ false
a + (-b)	⇒ a - b	a ∨ true	⇒ true
-(-a)	⇒ a	a ∨ false	⇒ a

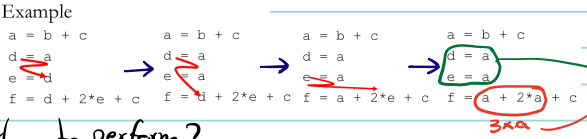
- Reduces the number of instructions
- Uses less expensive instructions
- Enable other optimizations

Not a Data-Flow Optimization

Copy Propagation

After other optimizations (ex. algebraic simplification).
Can lead to further algebraic simplification.

- Bypass Multiple Copying
  - propagate a value directly to its use
- Example



Can reduce instructions by Eliminating Copy Operations.

How to perform?

- At each RHS Expression and for each variable `v` used in the RHS expression
  - if the variable v is defined by a statement of the form `v = u`
  - replace the variable `v` by `u`
- At each Point of the Program Need to Know
  - The variables that are equal
  - Track equal variables by keeping tuples of the form `<u,v>` in a set iff `v = u` at that point in the program (`u`, `v` are variables)
- An assignment of `v = u` is still valid at a given point of the execution if and only if
  - An statement of `v = u` occurs in every execution path that reaches the current point
  - The variable `v` is not redefined in ***any these execution paths*** between the assign statement and the current point
  - The variable `u` is not redefined in ***any these execution paths*** between the assign statement and the current point
- A Data-Flow Problem !!!

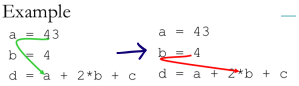
See examples in slides 43 - 88

Data-Flow Problem!

- Domain
  - set of tuples `<v,u>` representing a statement `v = u`
- Data-Flow Direction
  - Forward
- Data-Flow Function
  - OUT = Gen ∪ (IN - Kill)
  - Gen = { <v,u> | v = u is the statement }
  - Kill = { <v,u> | LHS var. of an assignment stmt. is either v or u }
- Meet Operation
  - IN = ⋂ OUT
- Initial Values
  - Empty Set

Constant Propagation

- Use Constant Values
  - Use the known constant of a variable
- Example



How to perform?

- At each RHS Expression
  - For each variable v used in the RHS
    - If the variable v is a known constant k
    - Replace the variable v by k
- At Each Point of the Program need to know:
  - For each variable v, if v is a constant,
  - If so, what the specific constant value is
- A variable v is the constant k at a point of the execution if and only if
  - The current statement is v = k or
  - *Every path reaching the current point* has k assigned to v
- A Data-Flow Problem !!!

Opportunities

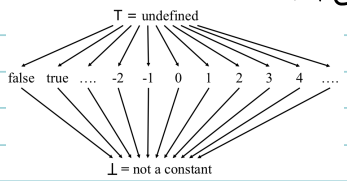
- User-Defined Constants
  - Same Constants propagating from many different paths
  - Symbolic Constants defined as variables
- Constants Known to the Compiler
  - data sizes, stack offsets
- Constants Available after Other Optimizations
  - Algebraic Simplification
  - Copy propagation

→ Enables Further Optimizations
See slides 102 - 118

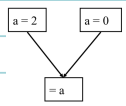
Data-Flow Problem!

- Domain
  - For each variable a lattice Lv
- Data-Flow Direction
  - Forward
- Data-Flow Function
  - OUT = gen ∨ (IN ∧ prsv)
  - gen = $\left\{ x_v \middle| x_v = \begin{cases} \top & \text{if v is not LHS} \\ \text{value} & \text{if v is the LHS \& RHS value is a const.} \\ \bot & \text{otherwise} \end{cases} \right\}$
  - prsv = $\left\{ x_v \middle| x_v = \begin{cases} \top & \text{if v is the LHS} \\ \bot & \text{if v is not the LHS} \end{cases} \right\}$
- **Obs:** For the prsv set, if a variable is not the LHS, we get ⊥ and the value of Lv is ⊥; For the gen set, if a variable is not the LHS, we get ⊤ and the value of Lv does not change.

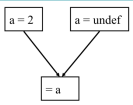
Lattice For Constant Propagation



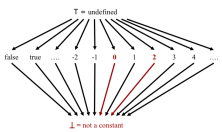
Meet Operations on the Lattice

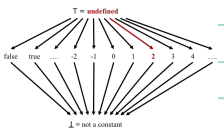


2 ∧ 0 = not a constant



2 ∧ undef = 2





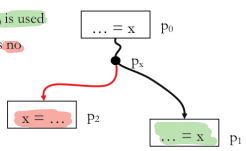
Third Summary

- Overview of Control-Flow Analysis
- Algebraic Simplification
- Copy Propagation
- Constant Propagation

Live-Variable Analysis

- What is Live-Variable Analysis?
  - For each Variable x where is the last program point *p* where the a specific value of x is used.
  - In other words, for x and program point *p* determine if the value of x at *p* can still be used along some path starting at *p*.
    - If so, x is live at *p*
    - If not x is dead at *p*
  - Must take Control-Flow into account : a Data-Flow Problem !!!
- Applications:
  - Register Allocation: If a variable is dead at a given point *p*
    - Can reuse its storage, *i.e.* the register it occupies if any;
    - If its value as been modified must save the value to storage unless it is not live on exit of the procedure or loop

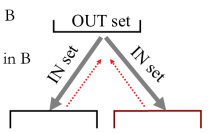
- At point p_0 the x variable is live:
  - There is a path to p_1 where value at p_0 is used
  - Beyond p_x towards p_2 the value of x is no longer needed and is dead



- Need to observe for each variable and for each program point:
  - Where is the last program point beyond which the value is not used
  - Trace back from uses to definitions and observe the first definition (backwards) that reaches that use.
  - That definition kills all uses backwards of it.

Formulation

- Variable is *live* at a point *p* if its value is used along *at least one Path*
  - A use of x prior to any definition in basic block means x must be alive
  - A definition of x in B prior to any subsequent use means previous uses must be dead
- Gen Set: Set of Variables Used in B
  - Upward Exposed Reads of B
- Kill Set: Set of Variables Defined in B



$$OUT(B) = \bigcup_{S \text{ a successor of } B} IN(s)$$

$$IN(B) = Use(B) \cup (OUT(B) - Def(B))$$

- Initialize IN(B) to Empty Set
- Compute Gen/Use and Kill/Def for each Basic Block
  - Tracing backwards from end of block to beginning of block
  - Initialize Last Instruction's Out(i) to Empty
  - Use IN(i) = use(i) ∪ (OUT(i) - def(i))
- Iteratively Apply Relations to Basic Block Until Convergence
  - $OUT(B) = \bigcup_{S \text{ a successor of } B} IN(s)$
  - IN(B) = Use(B) ∪ (OUT(B) - Def(B))
- Given OUT(B) use relations at instruction level to determine the live variables after each instruction

See algorithm example of Use & Def Functions for a Basic Block (slides 6-20)

Use(B) = {a}

t = a +1	use = { a } def = { t }	InUse = {a}
b = t	use = { t } def = { b }	OutUse = {a,t}
if (a = b) goto L_2	use = { a, b } def = { }	OutUse = {a,b}
		OutUse = { }

Def(B) = {t,b}

t = a +1	use = { a } def = { t }	InDef = {t, b}
b = t	use = { t } def = { b }	OutDef = {b}
if (a = b) goto L_2	use = { a, b } def = { }	OutDef = { }
		OutDef = { }

$$InUse(i) = Use(i) \cup (OutUse(i) - Def(i))$$

$$InDef(i) = Def(i) \cup OutDef(i)$$

- Can be Accomplished by a Forward Scanning of the Block
  - Keep Track of Which Variables are Read before they are written thus computing the Upwards Exposed Reads (UpExp) or Use Function
  - Track Variables that are Written or Killed (VarKill) or Def Function

```
// Assume instruction in format "x ← y op z"
for i ← 1 to Num Instructions in B do
    if (instr(i) is leader of B) then
        b ← Number(B);
        UpExp(b) ← ∅;
        VarKill(b) ← ∅;
    if y ∉ VarKill(b) then
        UpExp(b) ← UpExp(b) ∪{y}
    if z ∉ VarKill(b) then
        UpExp(b) ← UpExp(b) ∪{z}
    VarKill(b) ← VarKill(b) ∪{x}
```

See Algorithm with the blocks between the slides 22 and 40.

Fourth Summary

- What is Live-Variable Analysis?
  - Backward Data-Flow Analysis Problem
  - Upwards-Exposed (Gen): Forward Pass computation
- Most Significant Application
  - Register Allocation