CS412/413

Introduction to Compilers
Radu Rugina

Lecture 22: Using Dataflow Analysis
15 Mar 02

Outline

- Apply dataflow framework to several analysis problems:
  - Live variable analysis
  - Available expressions
  - Reaching definitions
  - Constant folding
- Also covered:
  - Implementation issues
  - Classification of dataflow analyses

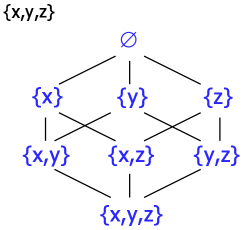
Problem 1: Live Variables

- Compute live variables at each program point
- Live variable = variable whose value may be used later, in some execution of the program
- Dataflow information: sets of live variables
- Example: variables {x,z} may be live at program point p
- Is a backward analysis
- Let V = set of all variables in the program
- Lattice (L, ⊑), where:
  - $L = 2^V$ (power set of V, i.e. set of all subsets of V)
  - Partial order ⊑ is set inclusion: ⊇

$$S_1 \sqsubseteq S_2 \text{ iff } S_1 \supseteq S_2$$

LV: The Lattice

- Consider set of variables V = {x,y,z}
- Partial order: ⊇
- Set V is finite implies lattice has finite height
- Meet operator: ∪ (set union: out[B] is union of in[B′], for all B′∈succ(B))
- Top element: ∅ (empty set)



- Smaller sets of live variables = more precise analysis
- All variables may be live = least precise

LV: Dataflow Equations

- General dataflow equations (X_0 is information at the end of exit basic block):

  $in[B] = F_B(out[B])$, for all B
  $out[B] = \sqcap \{in[B'] \mid B' \in succ(B)\}$, for all B
  $out[B_e] = X_0$

- Replace meet with set union:

  $in[B] = F_B(out[B])$, for all B
  $out[B] = \cup \{in[B'] \mid B' \in succ(B)\}$, for all B
  $out[B_e] = X_0$

- Meaning of union meet operator:
  "A variable is live at the end of a basic block B if it is live at the beginning of one of its successor blocks"

LV: Transfer Functions

- Transfer functions for basic blocks are composition of transfer functions of instructions in the block
- Define transfer functions for instructions
- General form of transfer functions:

  $F_I(X) = (X - def[I]) \cup use[I]$

  where:
  def[I] = set of variables defined (written) by I
  use[I] = set of variables used (read) by I

- Meaning of transfer functions:
  "Variables live before instruction I include: 1) variables live after I, not written by I, and 2) variables used by I"

LV: Transfer Functions

- Define def/use for each type of instruction

if I is x = y OP z :	use[I] = {y, z}	def[I] = {x}
if I is x = OP y :	use[I] = {y}	def[I] = {x}
if I is x = y :	use[I] = {y}	def[I] = {x}
if I is x = addr y :	use[I] = {}	def[I] = {x}
if I is if (x) :	use[I] = {x}	def[I] = {}
if I is return x :	use[I] = {x}	def[I] = {}
if I is x = f(y_1,…, y_n) :	use[I] = {y_1,…, y_n} def[I] = {x}	

- Transfer functions $F_I(X)$ = (X – def[I]) ∪ use[I]
- For each F_I, def[I] and use[I] are constants: they don't depend on input information X

LV: Monotonicity

- Are transfer functions: $F_I(X)$ = (X – def[I]) ∪ use[I] monotonic?
- Because def[I] is constant, X – def[I] is monotonic:
  X1 ⊇ X2 implies X1 – def[I] ⊇ X2 – def[I]
- Because use[I] is constant, Y ∪ use[I] is monotonic:
  Y1 ⊇ Y2 implies Y1 ∪ use[I] ⊇ Y2 ∪ use[I]
- Put pieces together: $F_I(X)$ is monotonic
  X1 ⊇ X2 implies
  (X1 – def[I]) ∪ use[I] ⊇ (X2 – def[I]) ∪ use[I]

LV: Distributivity

- Are transfer functions: $F_I(X)$ = (X – def[I]) ∪ use[I] distributive?
- Since def[I] is constant: X – def[I] is distributive:
  (X1 ∪ X2) – def[I] = (X1 – def[I]) ∪ (X2 – def[I])
  because: (a ∪ b) – c = (a – c) ∪ (b – c)
- Since use[I] is constant: Y ∪ use[I] is distributive:
  (Y1 ∪ Y2) ∪ use[I] = (Y1 ∪ use[I]) ∪ (Y2 ∪ use[I])
  because: (a ∪ b) ∪ c = (a ∪ c) ∪ (b ∪ c)
- Put pieces together: $F_I(X)$ is distributive
  F_I(X1 ∪ X2) = F_I(X1) ∪ F_I(X2)

Live Variables: Summary

- Lattice: (2^V, ⊇); has finite height
- Meet is set union, top is empty set
- Is a backward dataflow analysis
- Dataflow equations:
  in[B] = F_B(out[B]), for all B
  out[B] = ∪ {in[B′] | B′∈succ(B)}, for all B
  out[B_e] = X_0
- Transfer functions: $F_I(X)$ = (X – def[I]) ∪ use[I]
  - are monotonic and distributive
- Iterative solving of dataflow equation:
  - terminates
  - computes MOP solution

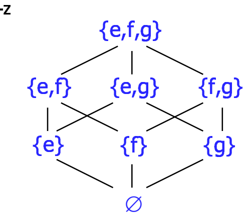
Problem 2: Available Expressions

- Compute available expressions at each program point
- Available expression = expression evaluated in all program executions, and its value would be the same if re-evaluated
- Is similar to available copies discussed earlier
- Dataflow information: sets of available expressions
- Example: expressions {x+y, y-z} are available at point p
- Is a forward analysis
- Let E = set of all expressions in the program
- Lattice (L, ⊑), where:
  - L = 2^E (power set of E, i.e. set of all subsets of E)
  - Partial order ⊑ is set inclusion: ⊆

  $S_1 \sqsubseteq S_2$ iff $S_1 \subseteq S_2$

AE: The Lattice

- Consider set of expressions = {x*z, x+y, y-z}
- Denote e = x*z, f=x+y, g=y-z
- Partial order: ⊆
- Set E is finite implies lattice has finite height
- Meet operator: ∩ (set intersection)
- Top element: {e,f,g} (set of all expressions)



- Larger sets of available variables = more precise analysis
- No available expressions = least precise

AE: Dataflow Equations

- General forward dataflow equations (X_0 is information at beginning of entry basic block):
  - out[I] = F_B(in[I]), for all B
  - in[B] = ⊓ {out[B′] | B′∈pred(B)}, for all B
  - in[B_s] = X_0
- Replace meet with set intersection:
  - out[I] = F_B(in[I]), for all B
  - in[B] = ∩ {out[B′] | B′∈pred(B)}, for all B
  - in[B_s] = X_0
- Meaning of intersection meet operator: "An expression is available at entry of block B if it is available at exit of all predecessor nodes"

AE: Transfer Functions

- Define transfer functions for instructions
- General form of transfer functions:
  - $F_I(X)$ = (X – kill[I]) ∪ gen[I]

  where:
  - kill[I] = expressions "killed" by I
  - gen[I] = new expressions "generated" by I
- Note: this kind of transfer function is typical for the majority of the dataflow analyses!
- Meaning of transfer functions: "Expressions available after instruction I include: 1) expressions available before I, not killed by I, and 2) expressions generated by I"

AE: Transfer Functions

- Define kill/gen for each type of instruction

if I is x = y OP z :	gen[I] = {y OP z}	kill[I] = {E | x∈E}
if I is x = OP y :	gen[I] = {OP z}	kill[I] = {E | x∈E}
if I is x = y :	gen[I] = {}	kill[I] = {E | x∈E}
if I is x = addr y :	gen[I] = {}	kill[I] = {E | x∈E}
if I is if (x) :	gen[I] = {}	kill[I] = {}
if I is return x :	gen[I] = {}	kill[I] = {}
if I is x = f(y_1,…, y_n) :	gen[I] = {}	kill[I] = {E | x∈E}

- Transfer functions $F_I(X)$ = (X – kill[I]) ∪ gen[I]
- For each F_I, kill[I] and gen[I] are constants: they don't depend on input information X

AE: Monotonicity

- Are transfer functions: $F_I(X)$ = (X – kill[I]) ∪ gen[I] monotonic?
- Because kill[I] is constant, X – kill[I] is monotonic:
  - X1 ⊆ X2 implies X1 – kill[I] ⊆ X2 – kill[I]
- Because gen[I] is constant, Y ∪ gen[I] is monotonic:
  - Y1 ⊆ Y2 implies Y1 ∪ gen[I] ⊆ Y2 ∪ gen[I]
- Put pieces together: $F_I(X)$ is monotonic
  - X1 ⊆ X2 implies
  - (X1 – kill[I]) ∪ gen[I] ⊆ (X2 – kill[I]) ∪ gen[I]

AE: Distributivity

- Are transfer functions: $F_I(X)$ = (X – kill[I]) ∪ gen[I] distributive?
- Since kill[I] is constant: X – kill[I] is distributive:
  - (X1 ∩ X2) – def[I] = (X1 – def[I]) ∩ (X2 – def[I])
  - because: (a ∩ b) – c = (a – c) ∩ (b – c)
- Since gen[I] is constant: Y ∪ gen[I] is distributive:
  - (Y1 ∩ Y2) ∪ gen[I] = (Y1 ∩ gen[I]) ∪ (Y2 ∩ gen[I])
  - because: (a ∩ b) ∪ c = (a ∩ c) ∪ (b ∩ c)
- Put pieces together: $F_I(X)$ is distributive
  - F_I(X1 ∩ X2) = F_I(X1) ∩ F_I(X2)

Available Expressions: Summary

- Lattice: (2^E, ⊆); has finite height
- Meet is set intersection, top element is E
- Is a forward dataflow analysis
- Dataflow equations:
  - out[I] = F_B(in[I]), for all B
  - in[B] = ∩ {out[B′] | B′∈pred(B)}, for all B
  - in[B_s] = X_0
- Transfer functions: $F_I(X)$ = (X – kill[I]) ∪ gen[I]
  - - are monotonic and distributive
- Iterative solving of dataflow equation:
  - - terminates
  - - computes MOP solution

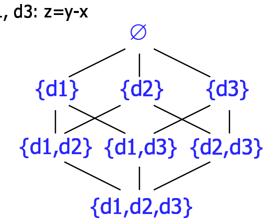
Problem 3: Reaching Definitions

- Compute reaching definitions for each program point
- Reaching definition = definition of a variable whose assigned value may be observed at current program point in some execution of the program
- Dataflow information: sets of reaching definitions
- Example: definitions {d2, d7} may reach program point p
- Is a forward analysis
- Let D = set of all definitions (assignments) in the program
- Lattice (D, $\sqsubseteq$), where:
  - L = 2^D (power set of D)
  - Partial order $\sqsubseteq$ is set inclusion: $\supseteq$

$S_1 \sqsubseteq S_2$ iff $S_1 \supseteq S_2$

RD: The Lattice

- Consider set of expressions = {d1, d2, d3} where d1: x = y, d2: x=x+1, d3: z=y-x
- Partial order: $\supseteq$
- Set D is finite implies lattice has finite height
- Meet operator: $\cup$ (set union)
- Top element: $\emptyset$ (empty set)



- Smaller sets of reaching definitions = more precise analysis
- All definitions may reach current point = least precise

RD: Dataflow Equations

- General forward dataflow equations (X_0 is information at beginning of entry basic block):
  - out[I] = F_B(in[I]), for all B
  - in[B] = $\sqcap$ {out[B′] | B′$\in$pred(B)}, for all B
  - in[B_s] = X_0
- Replace meet with set union:
  - out[I] = F_B(in[I]), for all B
  - in[B] = $\cup$ {out[B′] | B′$\in$pred(B)}, for all B
  - in[B_s] = X_0
- Meaning of intersection meet operator: "A definition reaches the entry of block B if it reaches the exit of at least one of its predecessor nodes"

RD: Transfer Functions

- Define transfer functions for instructions
- General form of transfer functions:

  $F_I(X)$ = (X – kill[I]) $\cup$ gen[I]

  where:

  kill[I] = definitions "killed" by I

  gen[I] = definitions "generated" by I
- Meaning of transfer functions: "Reaching definitions after instruction I include: 1) reaching definitions before I, not killed by I, and 2) reaching definitions generated by I"

RD: Transfer Functions

- Define kill/gen for each type of instruction
- If I is a definition d:

  gen[I] = {d} kill[I] = {d′ | d′ defines x}
- If I is not a definition:

  gen[I] = {} kill[I] = {}
- Transfer functions $F_I(X)$ = (X – kill[I]) $\cup$ gen[I]
- For each F_I, kill[I] and gen[I] are constants: they don't depend on input information X

RD : Monotonicity

- Transfer function: $F_I(X)$ = (X – kill[I]) $\cup$ gen[I]
- $F_I(X)$ is monotonic

  X1 $\supseteq$ X2 implies

  (X1 – kill[I]) $\cup$ gen[I] $\supseteq$ (X2 – kill[I]) $\cup$ gen[I]
- $F_I(X)$ is distributive

  F_I(X1 $\cup$ X2) = F_I(X1) $\cup$ F_I(X2)
- Same reasoning as before

Reaching Definitions: Summary

- Lattice: (2^D, ⊇); has finite height
- Meet is set union, top element is ∅
- Is a forward dataflow analysis
- Dataflow equations:
  - out[I] = F_B(in[I]), for all B
  - in[B] = ∪ {out[B′] | B′∈pred(B)}, for all B
  - in[B_s] = X_0
- Transfer functions: $F_I(X)$ = (X – kill[I]) ∪ gen[I]
  - are monotonic and distributive
- Iterative solving of dataflow equation:
  - terminates
  - computes MOP solution

Implementation

- Lattices in these analyses = power sets
- Information in these analyses = subsets of a set
- How to implement subsets?

1. Set implementation
   - Data structure with as many elements as the subset has
   - Usually list implementation
2. Bitvectors:
   - Use a bit for each element in the overall set
   - Bit for element x is: 1 if x is in subset, 0 otherwise
   - Example: S = {a,b,c}, use 3 bits
   - Subset {a,c} is 101, subset {b} is 010, etc.

Implementation Tradeoffs

- Advantages of bitvectors:
  - Efficient implementation of set union/intersection:
    set union is bitwise "or" of bitvectors
    set intersection is bitwise "and" of bitvectors
  - Drawback: inefficient for subsets with few elements
- Advantage of list implementation:
  - Efficient for sparse representation
  - Drawback: inefficient for set union or intersection
- In general, bitvectors work well if the size of the (original) set is linear in the program size

Problem 4: Constant Folding

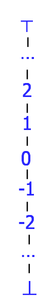
- Compute constant variables at each program point
- Constant variable = variable having a constant value on all program executions
- Dataflow information: sets of constant values
- Example: {x=2, y=3} at program point p
- Is a forward analysis
- Let V = set of all variables in the program, nvar = |V|
- Let N = set of integer constants
- Use a lattice over the set V x N
- Construct the lattice starting from a lattice for N
- Problem: (N, ≤) is not a complete lattice!
  - … because there is no LUB(N) and GLB(N)

Constant Folding Lattice

- Second try: lattice (N∪{⊤,⊥}, ≤)
  - Where ⊥≤n, for all n∈N
  - And n≤⊤, for all n∈N
  - Is complete!
- Meaning:
  - v=⊤: don't know if v is constant
  - v=⊥: v is not constant

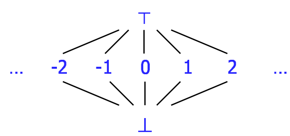
Constant Folding Lattice

- Second try: lattice (N∪{⊤,⊥}, ≤)
  - Where ⊥≤n, for all n∈N
  - And n≤⊤, for all n∈N
  - Is complete!
- Problem:
  - Is incorrect for constant folding
  - Meet of two constants c≠d is min(c,d)
  - Meet of different constants should be ⊥
- Another problem: has infinite height …



Constant Folding Lattice

- Solution: flat lattice $L = (N \cup \{\top,\bot\}, \sqsubseteq)$
  - Where $\bot \sqsubseteq n$, for all $n \in N$
  - And $n \sqsubseteq \top$, for all $n \in N$
  - And distinct integer constants are not comparable



- Note: meet of any two distinct numbers is $\bot$!

Constant Folding Lattice

- Denote $N^* = N \cup \{\top,\bot\}$
- Use flat lattice $L = (N^*, \sqsubseteq)$
- Constant folding lattice: $L' = (V \to N^*, \sqsubseteq_C)$
- Where partial order on $V \to N^*$ is defined as:

  $X \sqsubseteq_C Y$ iff for each variable v: $X(v) \sqsubseteq Y(v)$
- Can represent a function in $V \to N^*$ as a set of assignments: { {v1=c1}, {v2=c2}, …, {vn=cn} }

CF: Transfer Functions

- Transfer function for instruction I:

  $F_I(X) = (X - kill[I]) \cup gen[I]$

  where:

  kill[I] = constants "killed" by I

  gen[I] = constants "generated" by I
- $X[v] = c \in N^*$ if $\{v=c\} \in X$
- If I is v = c (constant): gen[I] = {v=c} kill[I] = {v} x N*
- If I is v = u+w: gen[I] = {v=e} kill[I] = {v} x N*

  where e = X[u] + X[w], if X[u] and X[w] are not $\top,\bot$

  e = $\bot$, if X[u] = $\bot$ or X[w] = $\bot$

  e = $\top$, if X[u] = $\top$ and X[w] = $\top$

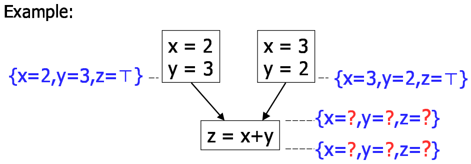
CF: Transfer Functions

- Transfer function for instruction I:

  $F_I(X) = (X - kill[I]) \cup gen[I]$
- Here gen[I] is not constant, it depends on X
- However transfer functions are monotonic (easy to prove)
- … but are transfer functions distributive?

CF: Distributivity

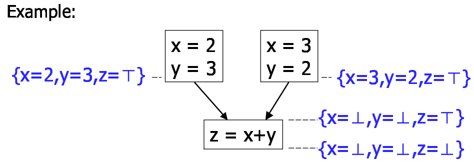
- Example:



- At join point, apply meet operator
- Then use transfer function for z=x+y

CF: Distributivity

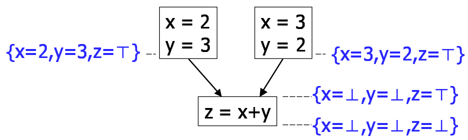
- Example:



- Dataflow result (MFP) at the end: {x=⊥,y=⊥,z=⊥}
- MOP solution at the end: {x=⊥,y=⊥,z=5} !

CF: Distributivity

- Example:



- Reason for MOP $\neq$ MFP:
  transfer function F of z=x+y is not distributive!

  $F(X1 \sqcap X2) \neq F(X1) \sqcap F(X2)$

  where X1 = {x=2,y=3,z=⊤} and X2 = {x=3,y=2,z=⊤}

Classification of Analyses

- Forward analyses: information flows from
  - CFG entry block to CFG exit block
  - Input of each block to its output
  - Output of each block to input of its successor blocks
  - Examples: available expressions, reaching definitions, constant folding
- Backward analyses: information flows from
  - CFG exit block to entry block
  - Output of each block to its input
  - Input of each block to output of its predecessor blocks
  - Example: live variable analysis

Another Classification

- "may" analyses:
  - information describes a property that MAY hold in SOME executions of the program
  - Usually: $\sqcap = \cup$, $\top = \emptyset$
  - Hence, initialize info to empty sets
  - Examples: live variable analysis, reaching definitions
- "must" analyses:
  - information describes a property that MUST hold in ALL executions of the program
  - Usually: $\sqcap = \cap$, $\top = S$
  - Hence, initialize info to the whole set
  - Examples: available expressions