CS412/413

Introduction to Compilers
Radu Rugina

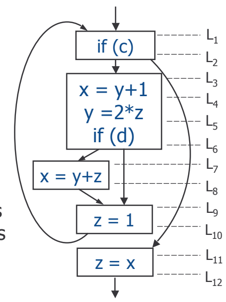
Lecture 22: Dataflow Analysis Frameworks
14 Mar 03

Live Variable Analysis

What are the live variables at each program point?

Method:

1. Define sets of live variables
1. Build constraints
2. Solve constraints



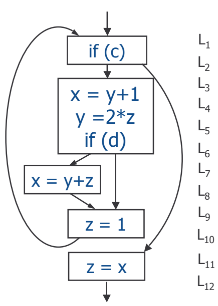
Derive Constraints

Constraints for each instruction:

in[I]=(out[I]-def[I]) ∪ use[I]

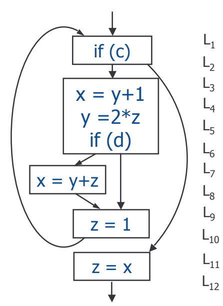
Constraints for control flow:

$out[B] = \bigcup_{B' \in succ(B)} in[B']$



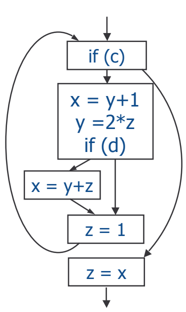
Derive Constraints

$L_1 = L_2 \cup \{c\}$
$L_2 = L_3 \cup L_{11}$
$L_3 = (L_4\text{-}\{x\}) \cup \{y\}$
$L_4 = (L_5\text{-}\{y\}) \cup \{z\}$
$L_5 = L_6 \cup \{d\}$
$L_6 = L_7 \cup L_9$
$L_7 = (L_8\text{-}\{x\}) \cup \{y,z\}$
$L_8 = L_9$
$L_9 = L_{10}\text{-}\{z\}$
$L_{10} = L_1$
$L_{11} = (L_{12}\text{-}\{z\}) \cup \{x\}$



Initialization

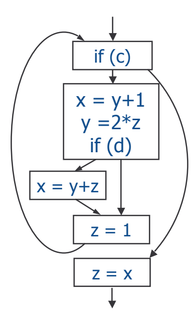
$L_1 = L_2 \cup \{c\}$
$L_2 = L_3 \cup L_{11}$
$L_3 = (L_4\text{-}\{x\}) \cup \{y\}$
$L_4 = (L_5\text{-}\{y\}) \cup \{z\}$
$L_5 = L_6 \cup \{d\}$
$L_6 = L_7 \cup L_9$
$L_7 = (L_8\text{-}\{x\}) \cup \{y,z\}$
$L_8 = L_9$
$L_9 = L_{10}\text{-}\{z\}$
$L_{10} = L_1$
$L_{11} = (L_{12}\text{-}\{z\}) \cup \{x\}$



$L_1=\{\}$
$L_2=\{\}$
$L_3=\{\}$
$L_4=\{\}$
$L_5=\{\}$
$L_6=\{\}$
$L_7=\{\}$
$L_8=\{\}$
$L_9=\{\}$
$L_{10}=\{\}$
$L_{11}=\{\}$
$L_{12}=\{\}$

Iteration 1

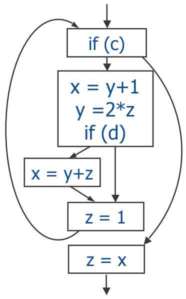
$L_1 = L_2 \cup \{c\}$
$L_2 = L_3 \cup L_{11}$
$L_3 = (L_4\text{-}\{x\}) \cup \{y\}$
$L_4 = (L_5\text{-}\{y\}) \cup \{z\}$
$L_5 = L_6 \cup \{d\}$
$L_6 = L_7 \cup L_9$
$L_7 = (L_8\text{-}\{x\}) \cup \{y,z\}$
$L_8 = L_9$
$L_9 = L_{10}\text{-}\{z\}$
$L_{10} = L_1$
$L_{11} = (L_{12}\text{-}\{z\}) \cup \{x\}$



$L_1=\{x,y,z,c,d\}$
$L_2=\{x,y,z,d\}$
$L_3=\{y,z,d\}$
$L_4=\{z,d\}$
$L_5=\{y,z,d\}$
$L_6=\{y,z\}$
$L_7=\{y,z\}$
$L_8=\{\}$
$L_9=\{\}$
$L_{10}=\{\}$
$L_{11}=\{x\}$
$L_{12}=\{\}$

Iteration 2

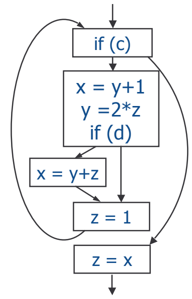
$L_1 = L_2 \cup \{c\}$
$L_2 = L_3 \cup L_{11}$
$L_3 = (L_4 - \{x\}) \cup \{y\}$
$L_4 = (L_5 - \{y\}) \cup \{z\}$
$L_5 = L_6 \cup \{d\}$
$L_6 = L_7 \cup L_9$
$L_7 = (L_8 - \{x\}) \cup \{y,z\}$
$L_8 = L_9$
$L_9 = L_{10} - \{z\}$
$L_{10} = L_1$
$L_{11} = (L_{12} - \{z\}) \cup \{x\}$



L_1={x,y,z,c,d}
L_2 ={x,y,z,c,d}
L_3 ={y,z,c,d}
L_4 ={x,z,c,d}
L_5 ={x,y,z,c,d}
L_6 ={x,y,z,c,d}
L_7 ={y,z,c,d}
L_8 ={x,y,c,d}
L_9 ={x,y,c,d}
L_{10} ={x,y,z,c,d}
L_{11} ={x}
L_{12} ={}

Fixed-point!

$L_1 = L_2 \cup \{c\}$
$L_2 = L_3 \cup L_{11}$
$L_3 = (L_4 - \{x\}) \cup \{y\}$
$L_4 = (L_5 - \{y\}) \cup \{z\}$
$L_5 = L_6 \cup \{d\}$
$L_6 = L_7 \cup L_9$
$L_7 = (L_8 - \{x\}) \cup \{y,z\}$
$L_8 = L_9$
$L_9 = L_{10} - \{z\}$
$L_{10} = L_1$
$L_{11} = (L_{12} - \{z\}) \cup \{x\}$

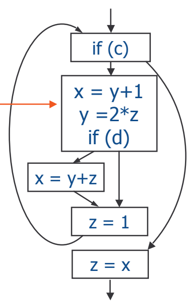


L_1={x,y,z,c,d}
L_2 ={x,y,z,c,d}
L_3 ={y,z,c,d}
L_4 ={x,z,c,d}
L_5 ={x,y,z,c,d}
L_6 ={x,y,z,c,d}
L_7 ={y,z,c,d}
L_8 ={x,y,c,d}
L_9 ={x,y,c,d}
L_{10} ={x,y,z,c,d}
L_{11} ={x}
L_{12} ={}

Final Result

x live here !

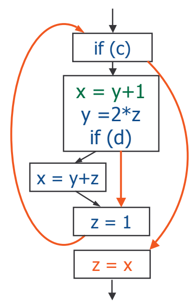
Final result: sets of live variables at each program point



L_1={x,y,z,c,d}
L_2 ={x,y,z,c,d}
L_3 ={y,z,c,d}
L_4 ={x,z,c,d}
L_5 ={x,y,z,c,d}
L_6 ={x,y,z,c,d}
L_7 ={y,z,c,d}
L_8 ={x,y,c,d}
L_9 ={x,y,c,d}
L_{10} ={x,y,z,c,d}
L_{11} ={x}
L_{12} ={}

Characterize All Executions

The analysis detects that there is an execution which uses the value x = y+1



L_1={x,y,z,c,d}
L_2 ={x,y,z,c,d}
L_3 ={y,z,c,d}
L_4 ={x,z,c,d}
L_5 ={x,y,z,c,d}
L_6 ={x,y,z,c,d}
L_7 ={y,z,c,d}
L_8 ={x,y,c,d}
L_9 ={x,y,c,d}
L_{10} ={x,y,z,c,d}
L_{11} ={x}
L_{12} ={}

Generalization

- Live variable analysis and detection of available copies are similar:
  - Define some information that they need to compute
  - Build constraints for the information
  - Solve constraints iteratively:
    - The information always "increases" during iteration
    - Eventually, it reaches a fixed point.
- We would like a general framework
  - Framework applicable to many other analyses
  - Live variable/copy propagation = instances of the framework

Dataflow Analysis Framework

- Dataflow analysis = a common framework for many compiler analyses
  - Computes some information at each program point
  - The computed information characterizes all possible executions of the program
- Basic methodology:
  - Describe information about the program using an algebraic structure called lattice
  - Build constraints which show how instructions and control flow modify the information in the lattice
  - Iteratively solve constraints

Lattices and Partial Orders

- Lattice definition uses the concept of partial order relation
- A partial order (P,⊑) consists of:
  - A set P
  - A partial order relation ⊑ which is:
    1. Reflexive $x \sqsubseteq x$
    2. Anti-symmetric $x \sqsubseteq y, y \sqsubseteq x \Rightarrow x = y$
    3. Transitive: $x \sqsubseteq y, y \sqsubseteq z \Rightarrow x \sqsubseteq z$
- Called "partial order" because not all elements are comparable

Lattices and Lower/Upper Bounds

- Lattice definition uses the concept of lower and upper bounds
- If (P,⊑) is a partial order and S ⊆ P, then:
  1. x∈P is a lower bound of S if $x \sqsubseteq y$, for all y∈S
  2. x∈P is an upper bound of S if $y \sqsubseteq x$, for all y∈S
- There may be multiple lower and upper bounds of the same set S

LUB and GLB

- Define least upper bounds (LUB) and greatest lower bounds (GLB)
- If (P,⊑) is a partial order and S ⊆ P, then:
  1. x∈P is GLB of S if:
     a) x is an lower bound of S
     b) $y \sqsubseteq x$, for any lower bound y of S
  2. x∈P is a LUB of S if:
     a) x is an upper bound of S
     b) $x \sqsubseteq y$, for any upper bound y of S
- … are GLB and LUB unique?

Lattices

- A pair (L,⊑) is a lattice if:
  1. (L,⊑) is a partial order
  2. Any finite subset S ⊆ L has a LUB and a GLB
- Can define two operators in lattices:
  1. Meet operator: $x \sqcap y$ = GLB({x,y})
  2. Join operator: $x \sqcup y$ = LUB({x,y})
- Meet and join are well-defined for lattices

Complete Lattices

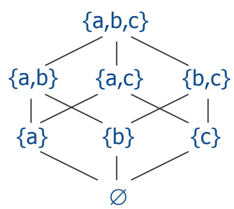
- A pair (L,⊑) is a complete lattice if:
  1. (L,⊑) is a partial order
  2. Any subset S ⊆ L has a LUB and a GLB
- Can define meet and join operators
- Can also define two special elements:
  1. Bottom element: $\bot$ = GLB(L)
  2. Top element: $\top$ = LUB(L)
- All finite lattices are complete

Example Lattice

- Consider S = {a,b,c} and its power set P = {∅, {a}, {b}, {c}, {a,b}, {b,c}, {a,c} {a,b,c}}
- Define partial order as set inclusion: X⊆Y
  - Reflexive $X \subseteq Y$
  - Anti-symmetric $X \subseteq Y, Y \subseteq X \Rightarrow X = Y$
  - Transitive $X \subseteq Y, Y \subseteq Z \Rightarrow X \subseteq Z$
- Also, for any two elements of P, there is a set which includes both and another set which is included in both
- Therefore (P,⊆) is a (complete) lattice

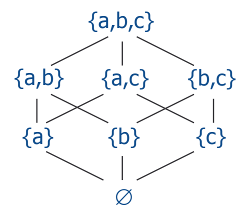
Hasse Diagrams

- Hasse diagram = graphical representation of a lattice where x is below y when $x \sqsubseteq y$ and $x \neq y$



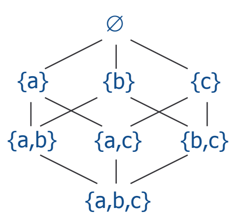
Power Set Lattice

- Partial order: $\subseteq$
  (set inclusion)
- Meet: $\cap$
  (set intersection)
- Join: $\cup$
  (set union)
- Top element: {a,b,c}
  (whole set)
- Bottom element: $\emptyset$
  (empty set)



Reversed Lattice

- Partial order: $\supseteq$
  (set inclusion)
- Meet: $\cup$
  (set union)
- Join: $\cap$
  (set intersection)
- Top element: $\emptyset$
  (empty set)
- Bottom element: {a,b,c}
  (whole set)



Relation To Dataflow Analysis

- Information computed by live variable analysis and available copies can be expressed as elements of lattices
- Live variables: if V is the set of all variables in the program and P the power set of V, then:
  - ($P,\subseteq$) is a lattice
  - sets of live variables are elements of this lattice

Relation To Analysis of Programs

- Copy Propagation:
  - V is the set of all variables in the program
  - V x V the cartesian product representing all possible copy instructions
  - P the power set of V x V
- Then:
  - ($P,\subseteq$) is a lattice
  - sets of available copies are lattice elements

More About Lattices

- In a lattice ($L, \sqsubseteq$), the following are equivalent:
  1. $x \sqsubseteq y$
  2. $x \sqcap y = x$
  3. $x \sqcup y = y$
- Note: meet and join operations were defined using the partial order relation

Proof

- Prove that x ⊑ y implies x ⊓ y = x:
  - x is a lower bound of {x,y}
  - All lower bounds of {x,y} are less than x,y
  - In particular, they are less than x
- Prove that x ⊓ y = x implies x ⊑ y :
  - x is a lower bound of {x,y}
  - x is less than x and y
  - In particular, x is less than y

Proof

- Prove that x ⊑ y implies x ⊔ y = y:
  - y is an upper bound of {x,y}
  - All upper bounds of {x,y} greater than x,y
  - In particular, they are greater than y
- Prove that x ⊔ y = y implies x ⊑ y :
  - y is a upper bound of {x,y}
  - y is greater than x and y
  - In particular, y is greater than x

Properties of Meet and Join

- The meet and join operators are:
  1. Associative (x ⊓ y) ⊓ z = x ⊓ (y ⊓ z)
  2. Commutative x ⊓ y = y ⊓ x
  3. Idempotent: x ⊓ x = x
- Property: If "⊓" is an associative, commutative, and idempotent operator, then the relation "⊑" defined as x ⊑ y iff x ⊓ y = x is a partial order
- Above property provides an alternative definition of a partial orders and lattices starting from the meet (join) operator

Using Lattices

- Assume information we want to compute in a program is expressed using a lattice L
- To compute the information at each program point we need to:
  - Determine how each instruction in the program changes the information in the lattice
  - Determine how lattice information changes at join/split points in the control flow

Transfer Functions

- Dataflow analysis defines a transfer function $F : L \rightarrow L$ for each instruction in the program
- Describes how the instruction modifies the information in the lattice
- Consider in[I] is information before I, and out[I] is information after I
- Forward analysis: out[I] = F(in[I])
- Backward analysis: in[I] = F(out[I])

Basic Blocks

- Can extend the concept of transfer function to basic blocks using function composition
- Consider:
  - Basic block B consists of instructions (I_1, …, I_n) with transfer functions F_1, …, F_n
  - in[B] is information before B
  - out[B] is information after B
- Forward analysis: out[B] = F_n(…(F_1(in[B])))
- Backward analysis: in[I] = F_1(… (F_n(out[i])))

Split/Join Points

- Dataflow analysis uses meet/join operations at split/join points in the control flow
- Consider in[B] is lattice information at beginning of block B and out[B] is lattice information at end of B
- Forward analysis: $in[B] = \sqcap \{out[B'] \mid B' \in pred(B)\}$
- Backward analysis: $out[B] = \sqcap \{in[B'] \mid B' \in succ(B)\}$
- Can alternatively use join operation $\sqcup$ (equivalent to using the meet operation $\sqcap$ in the reversed lattice)