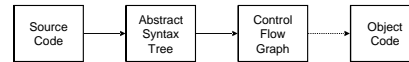


CMSC 631 — Program Analysis and Understanding
Fall 2006

Data Flow Analysis

Compiler Structure



- Source code parsed to produce AST
- AST transformed to CFG
- Data flow analysis operates on control flow graph (and other intermediate representations)

CMSC 631

2

Abstract Syntax Tree (AST)

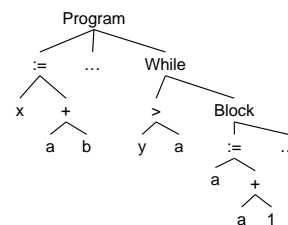
- Programs are written in text
 - ‡ I.e., sequences of characters
 - ‡ Awkward to work with
- First step: Convert to structured representation
 - ‡ Use lexer (like flex) to recognize *tokens*
 - Sequences of characters that make words in the language
 - ‡ Use parser (like bison) to group words structurally
 - And, often, to produce AST

CMSC 631

3

Abstract Syntax Tree Example

```
*x := a + b;  
*y := a * b;  
*while (y > a) {  
• a := a + 1;  
• x := a + b  
*}
```



CMSC 631

4

ASTs

- ASTs are *abstract*
 - ‡ They don't contain all information in the program
 - E.g., spacing, comments, brackets, parentheses
 - ‡ Any ambiguity has been resolved
 - E.g., $a + b + c$ produces the same AST as $(a + b) + c$
- For more info, see CMSC 430
 - ‡ In this class, we will generally begin with the AST

CMSC 631

5

Disadvantages of ASTs

- AST has many similar forms
 - ‡ E.g., for, while, repeat...until
 - ‡ E.g., if, ?, switch
- Expressions in AST may be complex, nested
 - ‡ $(42 * y) + (z > 5 ? 12 * z : z + 20)$
- Want simpler representation for analysis
 - ‡ ...at least, for dataflow analysis

CMSC 631

6

Control-Flow Graph (CFG)

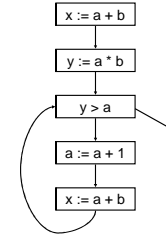
- A directed graph where
 - ‡ Each node represents a statement
 - ‡ Edges represent control flow
- Statements may be
 - ‡ Assignments $x := y \text{ op } z$ or $x := \text{op } z$
 - ‡ Copy statements $x := y$
 - ‡ Branches `goto L` or `if x relop y goto L`
 - ‡ etc.

CMSC 631

7

Control-Flow Graph Example

```
•x := a + b;  
•y := a * b;  
•while (y > a) {  
• a := a + 1;  
• x := a + b  
•}
```



CMSC 631

8

Variations on CFGs

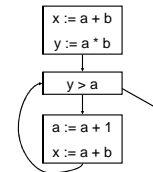
- Usually don't include declarations (e.g., `int x;`)
 - ‡ But there's usually something in the implementation
- May want a unique entry and exit node
 - ‡ Won't matter for the examples we give
- May group statements into basic blocks
 - ‡ A sequence of instructions with no branches into or out of the block

CMSC 631

9

Control-Flow Graph w/Basic Blocks

```
x := a + b;  
y := a * b;  
while (y > a + b) {  
 a := a + 1;  
 x := a + b  
}
```



- Can lead to more efficient implementations
- But more complicated to explain, so...
 - ‡ We'll use single-statement blocks in lecture today

CMSC 631

10

CFG vs. AST

- CFGs are much simpler than ASTs
 - ‡ Fewer forms, less redundancy, simple expressions
- But...AST is a more faithful representation
 - ‡ CFGs introduce temporaries
 - ‡ Lose block structure of program
- So for AST,
 - ‡ Easier to report error + other messages
 - ‡ Easier to explain to programmer
 - ‡ Easier to unparse to produce readable code

CMSC 631

11

Data Flow Analysis

- A framework for proving facts about programs
- Reasons about lots of little facts
- Little or no interaction between facts
 - ‡ Works best on properties about *how* program computes
- Based on all paths through program
 - ‡ Including infeasible paths

CMSC 631

12

Available Expressions

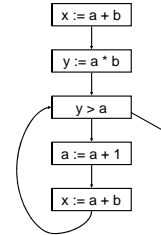
- Expression e is *available* at program point p if
 - ‡ e is computed on every path to p , and
 - ‡ the value of e has not changed since the last time e is computed on p
- Optimization
 - ‡ If an expression is available, need not be recomputed
 - (At least, if it's still in a register somewhere)

CMSC 631

13

Data Flow Facts

- Is expression e available?
- Facts:
 - ‡ $a + b$ is available
 - ‡ $a * b$ is available
 - ‡ $a + 1$ is available



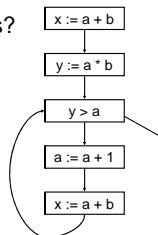
CMSC 631

14

Gen and Kill

- What is the effect of each statement on the set of facts?

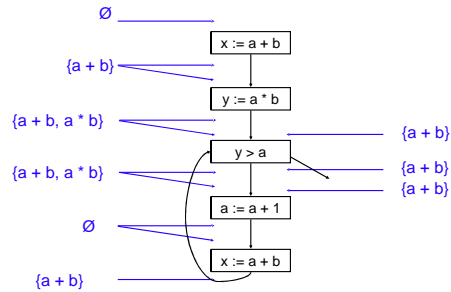
| Stmt | Gen | Kill |
|--------------|---------|-----------------------------------|
| $x := a + b$ | $a + b$ | |
| $y := a * b$ | $a * b$ | |
| $a := a + 1$ | | $a + 1$, $a + b$, $a * b$ |



CMSC 631

15

Computing Available Expressions



CMSC 631

16

Terminology

- A *join point* is a program point where two branches meet
- Available expressions is a *forward must* problem
 - ‡ Forward = Data flow from *in* to *out*
 - ‡ Must = At join point, property must hold on all paths that are joined

CMSC 631

17

Data Flow Equations

- Let s be a statement
 - ‡ $\text{succ}(s) = \{ \text{immediate successor statements of } s \}$
 - ‡ $\text{pred}(s) = \{ \text{immediate predecessor statements of } s \}$
 - ‡ $\text{In}(s)$ = program point just before executing s
 - ‡ $\text{Out}(s)$ = program point just after executing s
- $\text{In}(s) = \bigcap_{s' \in \text{pred}(s)} \text{Out}(s')$
- $\text{Out}(s) = \text{Gen}(s) \cup (\text{In}(s) - \text{Kill}(s))$
 - ‡ These are also called *transfer functions*

CMSC 631

18

Liveness Analysis

- A variable v is *live* at program point p if
 - ‡ v will be used on some execution path originating from p ...
 - ‡ before v is overwritten
- Optimization
 - ‡ If a variable is not live, no need to keep it in a register
 - ‡ If variable is dead at assignment, can eliminate assignment

CMSC 631

19

Data Flow Equations

- Available expressions is a forward must analysis
 - ‡ Data flow propagate in same dir as CFG edges
 - ‡ Expr is available only if available on all paths
- Liveness is a *backward may* problem
 - ‡ To know if variable live, need to look at future uses
 - ‡ Variable is live if used on some path
- $Out(s) = \bigcup_{s' \in succ(s)} In(s')$
- $In(s) = Gen(s) \cup (Out(s) - Kill(s))$

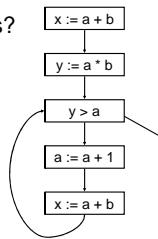
CMSC 631

20

Gen and Kill

- What is the effect of each statement on the set of facts?

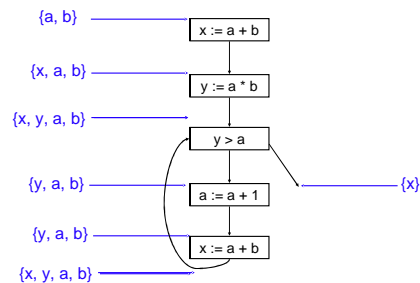
| Stmnt | Gen | Kill |
|--------------|--------|------|
| $x := a + b$ | a, b | x |
| $y := a * b$ | a, b | y |
| $y > a$ | a, y | |
| $a := a + 1$ | a | a |



CMSC 631

21

Computing Live Variables



CMSC 631

22

Very Busy Expressions

- An expression e is *very busy* at point p if
 - ‡ On every path from p , expression e is used before any component of e is changed
- Optimization
 - ‡ Can hoist very busy expression computation to p
- What kind of problem?
 - ‡ Forward or backward? **backward**
 - ‡ May or must? **must**

CMSC 631

23

Reaching Definitions

- A *definition* of a variable v is an assignment to v
- A definition of variable v reaches point p if
 - ‡ There is no intervening assignment to v
- Also called def-use information
- What kind of problem?
 - ‡ Forward or backward? **forward**
 - ‡ May or must? **may**

CMSC 631

24

Space of Data Flow Analyses

| | | |
|----------|----------------------|-----------------------|
| | May | Must |
| Forward | Reaching definitions | Available expressions |
| Backward | Live variables | Very busy expressions |

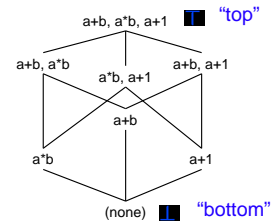
- Most data flow analyses can be classified this way
 - ‡ A few don't fit: bidirectional analysis
- Lots of literature on data flow analysis

CMSC 631

25

Data Flow Facts and Lattices

- Typically, data flow facts form a lattice
 - ‡ Example: Available expressions



CMSC 631

26

Partial Orders

- A partial order is a pair (P, \leq) such that
 - ‡ $\leq \subseteq P \times P$
 - ‡ \leq is reflexive: $x \leq x$
 - ‡ \leq is anti-symmetric: $x \leq y$ and $y \leq x \Rightarrow x = y$
 - ‡ \leq is transitive: $x \leq y$ and $y \leq z \Rightarrow x \leq z$

CMSC 631

27

Meet and Join Operations

- \sqcap is the *meet* or *greatest lower bound* operation:
 - ‡ $x \sqcap y \leq x$ and $x \sqcap y \leq y$
 - ‡ If $z \leq x$ and $z \leq y$, then $z \leq x \sqcap y$
- \sqcup is the *join* or *least upper bound* operation:
 - ‡ $x \leq x \sqcup y$ and $y \leq x \sqcup y$
 - ‡ If $x \leq z$ and $y \leq z$, then $x \sqcup y \leq z$

CMSC 631

28

Lattices

- A partial order (P, \leq) is a *lattice* if meet and join exist for every pair of elements in P
- A lattice has unique elements \perp and \top such that
 - ‡ $x \sqcap \perp = \perp$ and $x \sqcup \perp = x$
 - ‡ $x \sqcap \top = x$ and $x \sqcup \top = \top$
- In a lattice,
 - ‡ $x \leq y$ iff $x \sqcap y = x$
 - ‡ $x \leq y$ iff $x \sqcup y = y$
- A partial order is a *complete lattice* if meet and join are defined on any set $S \subseteq P$

CMSC 631

29

Useful Lattices

- $(2^S, \subseteq)$ forms a lattice for any set S
 - ‡ 2^S is the powerset of S (set of all subsets)
- If (S, \leq) is a lattice, so is (S, \geq)
 - ‡ i.e., lattices can be flipped
- The lattice for constant propagation



CMSC 631

30

Forward Must Data Flow Algorithm

- Out(s) = Top for all statements s
 - ‡ // Slight acceleration: Could set Out(s) = Gen(s) U (Top - Kill(s))
- W := { all statements } (worklist)
- repeat
 - ‡ Take s from W
 - ‡ In(s) := $\bigcap_{s' \in \text{pred}(s)} \text{Out}(s')$
 - ‡ temp := Gen(s) U (In(s) - Kill(s))
 - ‡ if (temp != Out(s)) {
 - Out(s) := temp
 - W := W U succ(s)
 - ‡ }
- until W = \emptyset

CMSC 631

31

Monotonicity

- A function f on a partial order is *monotonic* if

$$x \leq y \Rightarrow f(x) \leq f(y)$$

- Easy to check that operations to compute In and Out are monotonic

‡ In(s) := $\bigcap_{s' \in \text{pred}(s)} \text{Out}(s')$ a function $f_s(\text{In}(s))$

- Putting these two together

‡ Out(s) := $f_s(\bigcap_{s' \in \text{pred}(s)} \text{Out}(s'))$

CMSC 631

32

Termination

- We know the algorithm terminates because
 - ‡ The lattice has finite height
 - ‡ The operations to compute In and Out are monotonic
 - ‡ On every iteration, we remove a statement from the worklist and/or move down the lattice

CMSC 631

33

Forward Data Flow, Again

- Out(s) = Top for all statements s
- W := { all statements } (worklist)
- repeat
 - ‡ Take s from W
 - ‡ temp := $f_s(\bigcap_{s' \in \text{pred}(s)} \text{Out}(s'))$ (f_s monotonic transfer fn)
 - ‡ if (temp != Out(s)) {
 - Out(s) := temp
 - W := W U succ(s)
 - ‡ }
- until W = \emptyset

CMSC 631

34

Lattices (P, <=)

- Available expressions
 - ‡ P = sets of expressions
 - ‡ $S1 \sqcap S2 = S1 \cap S2$
 - ‡ Top = set of all expressions
- Reaching Definitions
 - ‡ P = set of definitions (assignment statements)
 - ‡ $S1 \sqcap S2 = S1 \cup S2$
 - ‡ Top = empty set

CMSC 631

35

Fixpoints

- We always start with Top
 - ‡ Most optimistic assumption
 - "every expression is available," "no defs reach this point"
 - ‡ Strongest possible hypothesis
 - = true of fewest number of states
- Revise as we encounter contradictions
 - ‡ Always move down in the lattice (with meet)
- Result: A greatest fixpoint

CMSC 631

36

Lattices (P, <=), cont'd

- Live variables
 - ‡ P = sets of variables
 - ‡ $S1 \sqcap S2 = S1 \cup S2$
 - ‡ Top = empty set
- Very busy expressions
 - ‡ P = set of expressions
 - ‡ $S1 \sqcap S2 = S1 \cap S2$
 - ‡ Top = set of all expressions

CMSC 631

37

Forward vs. Backward

| | |
|--|--|
| <p>Out(s) = Top for all s W := { all statements } repeat Take s from W temp := f ($\bigcap_{s' \in \text{pred}(s)} \text{Out}(s')$) if (temp != Out(s)) { Out(s) := temp W := W U succ(s) } until W = \emptyset</p> | <p>In(s) = Top for all s W := { all statements } repeat Take s from W temp := f ($\bigcap_{s' \in \text{succ}(s)} \text{In}(s')$) if (temp != In(s)) { In(s) := temp W := W U pred(s) } until W = \emptyset</p> |
|--|--|

CMSC 631

38

Termination Revisited

- How many times can we apply this step:

```

temp := f (  $\bigcap_{s' \in \text{pred}(s)} \text{Out}(s')$  )
if (temp != Out(s)) { ... }
    
```

‡ Claim: Out(s) only "shrinks"

- Proof: Out(s) starts out as top
- So temp = Top after first step
- Assume Out(s') shrinks for all predecessors s' of s
- Then $\bigcap_{s' \in \text{pred}(s)} \text{Out}(s')$ shrinks
- Since f monotonic, $f(\bigcap_{s' \in \text{pred}(s)} \text{Out}(s'))$ shrinks

CMSC 631

39

Termination Revisited (cont'd)

- A *descending chain* in a lattice is a sequence

‡ $x_0 > x_1 > x_2 > \dots$

- The *height* of a lattice is the length of the longest descending chain in the lattice

- Then, dataflow must terminate in $O(nk)$ time

‡ n = # of statements in program

‡ k = height of lattice

‡ assumes meet operation takes $O(1)$ time

CMSC 631

40

Least vs. Greatest Fixpoints

- Dataflow tradition: Start with Top, use meet

‡ To do this, we need a *complete meet semilattice with top, of finite height*

- complete meet semilattice = meets defined for any set

- finite height ensures termination

‡ Computes greatest fixpoint

- Denotational semantics tradition: Start with Bottom, use join

‡ Computes least fixpoint

CMSC 631

41

Distributive Data Flow Problems

- By monotonicity, we also have

$$f(x \sqcap y) \leq f(x) \sqcap f(y)$$

- A function f is distributive if

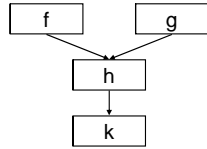
$$f(x \sqcap y) = f(x) \sqcap f(y)$$

CMSC 631

42

Benefit of Distributivity

- Joins lose no information



$$\begin{aligned}
 k(h(f(T)) \sqcap h(g(T))) &= \\
 k(h(f(T)) \sqcap h(g(T))) &= \\
 k(h(f(T)) \sqcap h(g(T))) &=
 \end{aligned}$$

CMSC 631

43

Accuracy of Data Flow Analysis

- Ideally, we would like to compute the meet over all paths (MOP) solution:

- ‡ Let f_s be the transfer function for statement s
- ‡ If p is a path $\{s_1, \dots, s_n\}$, let $f_p = f_{s_1}; \dots; f_{s_n}$
- ‡ Let $\text{path}(s)$ be the set of paths from the entry to s

$$\text{MOP}(s) = \sqcap_{p \in \text{path}(s)} f_p(T)$$

- If a data flow problem is distributive, then solving the data flow equations in the standard way yields the MOP solution

CMSC 631

44

What Problems are Distributive?

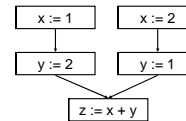
- Analyses of *how* the program computes
 - ‡ Live variables
 - ‡ Available expressions
 - ‡ Reaching definitions
 - ‡ Very busy expressions
- All Gen/Kill problems are distributive

CMSC 631

45

A Non-Distributive Example

- Constant propagation



- In general, analysis of *what* the program computes is not distributive

CMSC 631

46

Practical Implementation

- Data flow facts = assertions that are true or false at a program point
- Represent set of facts as bit vector
 - ‡ Fact_i represented by bit i
 - ‡ Intersection = bitwise and, union = bitwise or, etc
- “Only” a constant factor speedup
 - ‡ But very useful in practice

CMSC 631

47

Basic Blocks

- A *basic block* is a sequence of statements s.t.
 - ‡ No statement except the last in a branch
 - ‡ There are no branches to any statement in the block except the first
- In practical data flow implementations,
 - ‡ Compute Gen/Kill for each basic block
 - Compose transfer functions
 - ‡ Store only In/Out for each basic block
 - ‡ Typical basic block ~5 statements

CMSC 631

48

Order Matters

- Assume forward data flow problem
 - ‡ Let $G = (V, E)$ be the CFG
 - ‡ Let k be the height of the lattice
- If G acyclic, visit in topological order
 - ‡ Visit head before tail of edge
- Running time $O(|E|)$
 - ‡ No matter what size the lattice

CMSC 631

49

Order Matters — Cycles

- If G has cycles, visit in reverse postorder
 - ‡ Order from depth-first search
- Let $Q = \max \#$ back edges on cycle free path
 - ‡ Nesting depth
 - ‡ Back edge is from node to ancestor on DFS tree
- Then if $\forall x, f(x) \leq x$ (sufficient, but not necessary)
 - ‡ Running time is $O((Q+1)|E|)$
 - Note direction of req't depends on top vs. bottom

CMSC 631

50

Flow-Sensitivity

- Data flow analysis is *flow-sensitive*
 - ‡ The order of statements is taken into account
 - ‡ I.e., we keep track of facts per program point
- Alternative: *Flow-insensitive* analysis
 - ‡ Analysis the same regardless of statement order
 - ‡ Standard example: types
 - /* x : int */ x := ... /* x : int */

CMSC 631

51

Terminology Review

- Must vs. May
 - ‡ (Not always followed in literature)
- Forwards vs. Backwards
- Flow sensitive vs. Flow insensitive
- Distributive vs. Non distributive

CMSC 631

52

Another Approach: Elimination

- Recall in practice, one transfer function per basic block
- Why not generalize this idea beyond a basic block?
 - ‡ "Collapse" larger constructs into smaller ones, combining data flow equations
 - ‡ Eventually program collapsed into a single node!
 - ‡ "Expand out" back to original constructs, rebuilding information

CMSC 631

53

Lattices of Functions

- Let $(P, =)$ be a lattice
- Let M be the set of monotonic functions on P
- Define $f =_f g$ if for all $x, f(x) = g(x)$
- Define the function $f \sqcap g$ as
 - ‡ $(f \sqcap g)(x) = f(x) \sqcap g(x)$
- Claim: $(M, =_f)$ forms a lattice

CMSC 631

54

Elimination Methods: Conditionals

$$f_{ite} = (f_{then} \circ f_{if}) \sqcap (f_{else} \circ f_{if})$$

$$\text{Out}(\text{if}) = f_{if}(\text{In}(\text{ite}))$$

$$\text{Out}(\text{then}) = (f_{then} \circ f_{if})(\text{In}(\text{ite}))$$

$$\text{Out}(\text{else}) = (f_{else} \circ f_{if})(\text{In}(\text{ite}))$$

CMSC 631 55

Elimination Methods: Loops

$$f_{while} = f_{head} \sqcap$$

$$f_{head} \circ f_{body} \circ f_{head} \sqcap$$

$$f_{head} \circ f_{body} \circ f_{head} \circ f_{body} \circ f_{head} \sqcap \dots$$

CMSC 631 56

Elimination Methods: Loops (cont'd)

- Let $f^i = f \circ f \circ \dots \circ f$ (i times)
 $\vdash f^0 = \text{id}$
- Let $g(j) = \bigsqcap_{i \in [0..j]} (f_{head} \circ f_{body})^i \circ f_{head}$
- Need to compute limit as j goes to infinity
 \vdash Does such a thing exist?
- Observe: $g(j+1) \leq g(j)$

CMSC 631 57

Height of Function Lattice

- Assume underlying lattice (P, \leq) has finite height
 \vdash What is height of lattice of monotonic functions?
 \vdash Claim: finite (see homework)
- Therefore, $g(j)$ converges

CMSC 631 58

Non-Reducible Flow Graphs

- Elimination methods usually only applied to *reducible* flow graphs
 \vdash Ones that can be collapsed
 \vdash Standard constructs yield only reducible flow graphs
- Unrestricted goto can yield irreducible graphs

CMSC 631 59

Comments

- Can also do backwards elimination
 \vdash Not quite as nice (regions are usually single *entry* but often not single *exit*)
- For bit vector problems, elimination efficient
 \vdash Easy to compose functions, compute meet, etc.
- Elimination originally seemed like it might be faster than iteration
 \vdash Not really the case

CMSC 631 60

Data Flow Analysis and Functions

- What happens at a function call?
 - ‡ Lots of solutions in data flow analysis literature
- In practice, only analyze one procedure at a time
- Consequences
 - ‡ Call to function kills all data flow facts
 - ‡ May be able to improve depending on language, e.g., function call may not affect locals

CMSC 631

61

More Terminology

- An analysis that models only a single function at a time is *intraprocedural*
- An analysis that takes multiple functions into account is *interprocedural*
- An analysis that takes the whole program into account is...guess?

- Note: *global* analysis means “more than one basic block,” but still within a function

CMSC 631

62

Data Flow Analysis and The Heap

- Data Flow is good at analyzing local variables
 - ‡ But what about values stored in the heap?
 - ‡ Not modeled in traditional data flow
- In practice: $*x := e$
 - ‡ Assume all data flow facts killed (!)
 - ‡ Or, assume write through x may affect any variable whose address has been taken
- In general, hard to analyze pointers

CMSC 631

63

Data Flow Analysis and Optimization

- Moore's Law: Hardware advances double computing power every 18 months.
- Proebsting's Law: Compiler advances double computing power every 18 years.
- We'll focus on other uses of data flow analysis in this class (later in the semester)

CMSC 631

64

This document was created with Win2PDF available at <http://www.win2pdf.com>.
The unregistered version of Win2PDF is for evaluation or non-commercial use only.
This page will not be added after purchasing Win2PDF.