Chapter 6

Dependence and Data Flow Models

The control flow graph and state machine models introduced in the previous chapter capture one aspect of the dependencies among parts of a program. They explicitly represent control flow, but de-emphasize transmission of information through program variables. Data flow models provide a complementary view, emphasizing and making explicit relations involving transmission of information.

Models of data flow and dependence in software were developed originally in the field of compiler construction, where they were (and still are) used to detect opportunities for optimization. They also have many applications in software engineering, from testing to refactoring to reverse engineering. In test and analysis, applications range from selecting test cases based on dependence information (as described in Chapter 13) to detecting anomalous patterns that indicate probable programming errors, such as uses of potentially uninitialized values. Moreover, the basic algorithms used to construct data flow models have even wider application, and are of particular interest because they can often be quite efficient in time and space.

6.1 Definition-Use Pairs

The most fundamental class of data flow model associates the point in a program where a value is produced (called a "definition") with the points at which the value may be accessed (called a "use"). Associations of definitions and uses fundamentally capture the flow of information through a program, from input to output.

Definitions occur where variables are declared or initialized, assigned values, or received as parameters, and in general at all statements that change the value of one or more variables. Uses occur in expressions, conditional statements, parameter passing, return statements, and in general in all statements whose execution extracts a value from a variable. For example, in the standard GCD algorithm of Figure 6.1, line 1 contains a definition of parameters x and y, line 3 contains a use of variable y, line 6 contains a use of variable tmp and a definition of variable y, and the return in line 8 is

```
1   public int gcd(int x, int y) {        /* A: def x,y   */
2       int tmp;                          /*    def tmp   */
3       while (y != 0) {                  /* B: use y     */
4           tmp = x % y;                  /* C: use x,y, def tmp */
5           x = y;                        /* D: use y, def x  */
6           y = tmp;                      /* E: use tmp, def y */
7       }
8       return x;                         /* F: use x */
9   }
```

Figure 6.1: Java implementation of Euclid's algorithm for calculating the greatest common denominator of two positive integers. The labels A–F are provided to relate statements in the source code to graph nodes in subsequent figures.

a use of variable x.

Each definition-use pair associates a definition of a variable (e.g., the assignment to y in line 6) with a use of the same variable (e.g., the expression `y != 0` in line 3). A single definition can be paired with more than one use, and vice versa. For example, the definition of variable y in line 6 is paired with a use in line 3 (in the loop test), as well as additional uses in lines 4 and 5. The definition of x in line 5 is associated with uses in lines 4 and 8.

Δ kill

A definition-use pair is formed only if there is a program path on which the value assigned in the definition can reach the point of use without being overwritten by another value. If there is another assignment to the same value on the path, we say that the first definition is *killed* by the second. For example, the declaration of tmp in line 2 is not paired with the use of tmp in line 6, because the definition at line 2 is killed by the definition at line 4. A *definition-clear* path is a path from definition to use on which the definition is not killed by another definition of the same variable. For example, with reference to the node labels in Figure 6.2, path E,B,C,D is a definition-clear path from the definition of y in line 6 (node E of the control flow graph) to the use of y in line 5 (node D). Path A,B,C,D,E is not a definition-clear path with respect to tmp, because of the intervening definition at node C.

Δ definition-clear path

Δ direct data dependence

Definition-use pairs record a kind of program dependence, sometimes called direct data dependence. These dependencies can be represented in the form of a graph, with a directed edge for each definition-use pair. The data dependence graph representation of the GCD method is illustrated in Figure 6.3 with nodes that are program statements. Different levels of granularity are possible. For use in testing, nodes are typically basic blocks. Compilers often use a finer-grained data dependence representation, at the level of individual expressions and operations, to detect opportunities for performance-improving transformations.

The data dependence graph in Figure 6.3 captures only dependence through flow of data. Dependence of the body of the loop on the predicate governing the loop is not represented by data dependence alone. Control dependence can be also represented with a graph, as in Figure 6.5, which shows the control dependencies for the GCD

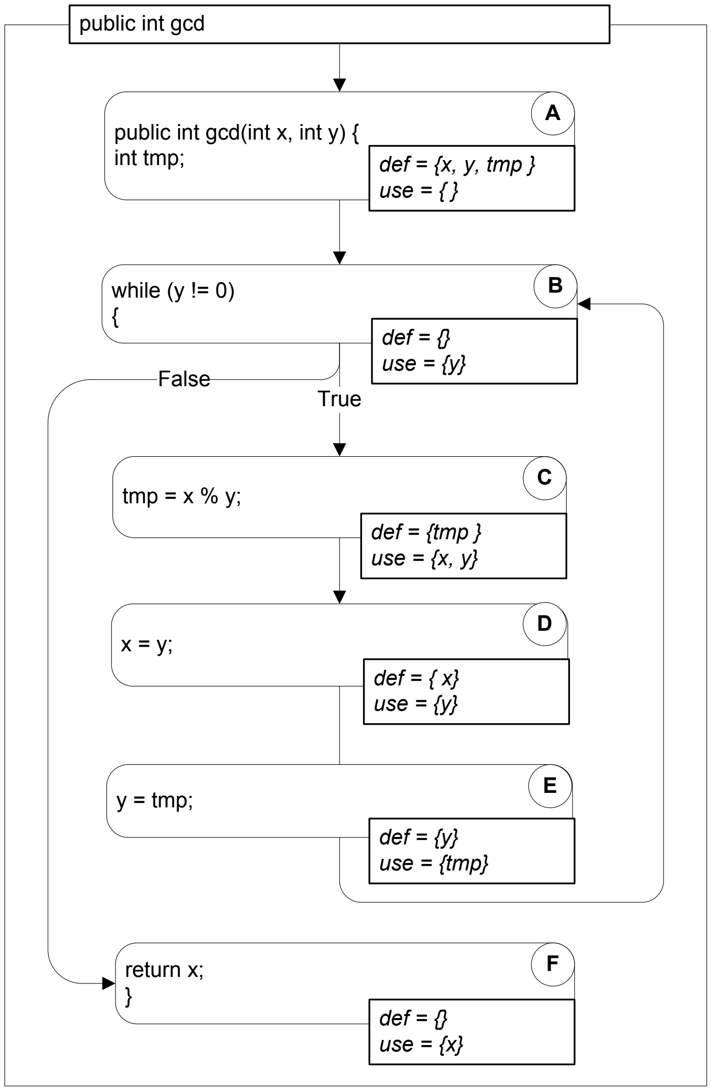

Figure 6.2: Control flow graph of GCD method in Figure 6.1

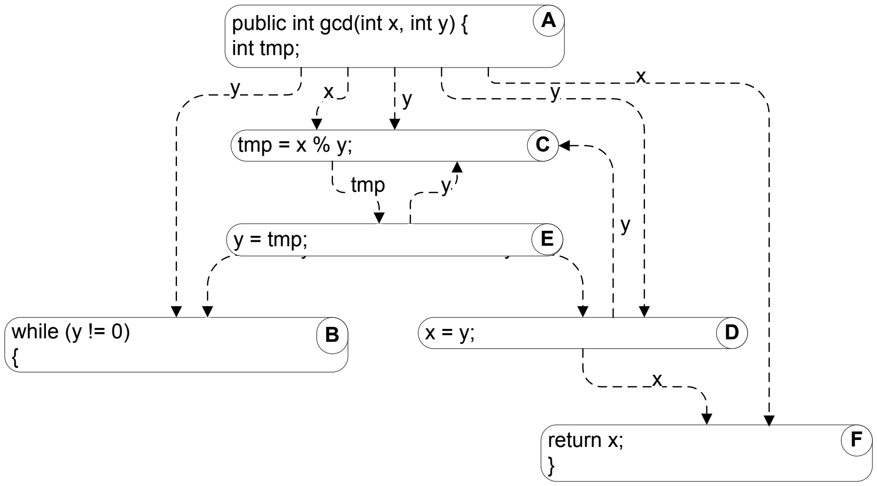

Figure 6.3: Data dependence graph of GCD method in Figure 6.1, with nodes for statements corresponding to the control flow graph in Figure 6.2. Each directed edge represents a direct data dependence, and the edge label indicates the variable that transmits a value from the definition at the head of the edge to the use at the tail of the edge.

method. The control dependence graph shows direct control dependencies, that is, where execution of one statement controls whether another is executed. For example, execution of the body of a loop or if statement depends on the result of a predicate.

Control dependence differs from the sequencing information captured in the control flow graph. The control flow graph imposes a definite order on execution even when two statements are logically independent and could be executed in either order with the same results. If a statement is control- or data-dependent on another, then their order of execution is not arbitrary. Program dependence representations typically include both data dependence and control dependence information in a single graph with the two kinds of information appearing as different kinds of edges among the same set of nodes.

A node in the control flow graph that is reached on every execution path from entry point to exit is control dependent only on the entry point. For any other node *N*, reached on some but not all execution paths, there is some branch which controls execution of *N* in the sense that, depending on which way execution proceeds from the branch, execution of *N* either does or does not become inevitable. It is this notion of control that control dependence captures.

Δ dominator

The notion of dominators in a rooted, directed graph can be used to make this intuitive notion of "controlling decision" precise. Node *M* dominates node *N* if every path from the root of the graph to *N* passes through *M*. A node will typically have many dominators, but except for the root, there is a unique *immediate dominator* of node N which is closest to N on any path from the root, and which is in turn dominated by all the other dominators of N. Because each node (except the root) has a unique

Δ immediate dominator

immediate dominator, the immediate dominator relation forms a tree.

The point at which execution of a node becomes inevitable is related to paths from a node to the end of execution — that is, to dominators that are calculated in the reverse of the control flow graph, using a special "exit" node as the root. Dominators in this direction are called post-dominators, and dominators in the normal direction of execution can be called pre-dominators for clarity.

Δ post-dominator
Δ pre-dominator

We can use post-dominators to give a more precise definition of control dependence. Consider again a node N that is reached on some but not all execution paths. There must be some node C with the following property: C has at least two successors in the control flow graph (i.e., it represents a control flow decision); C is not post-dominated by N (N is not already inevitable when C is reached); and there is a successor of C in the control flow graph that is post-dominated by N. When these conditions are true, we say node N is control-dependent on node C. Figure 6.4 illustrates the control dependence calculation for one node in the GCD example, and Figure 6.5 shows the control dependence relation for the method as a whole.

6.2 Data Flow Analysis

Definition-use pairs can be defined in terms of paths in the program control flow graph. As we have seen in the former section, there is an association (d,u) between a definition of variable v at d and a use of variable v at u iff there is at least one control flow path from d to u with no intervening definition of v. We also say that definition v_d *reaches* u, and that v_d is a *reaching definition* at u. If, on the other hand, a control flow path passes through another definition e of the same variable v, we say that v_e *kills* v_d at that point.

Δ reaching definition

It would be possible to compute definition-use pairs by searching the control flow graph for individual paths of the form described above. However, even if we consider only loop-free paths, the number of paths in a graph can be exponentially larger than the number of nodes and edges. Practical algorithms therefore cannot search every individual path. Instead, they summarize the reaching definitions at a node over all the paths reaching that node.

An efficient algorithm for computing reaching definitions (and several other properties, as we will see below) is based on the way reaching definitions at one node are related to the reaching definitions at an adjacent node. Suppose we are calculating the reaching definitions of node n, and there is an edge (p,n) from an immediate predecessor node p. We observe:

- If the predecessor node p can assign a value to variable v, then the definition v_p reaches n. We say the definition v_p is *generated* at p.
- If a definition v_d of variable v reaches a predecessor node p, and if v is not redefined at that node (in which case we say the v_d is *killed* at that point), then the definition is propagated on from p to n.

These observations can be stated in the form of an equation describing sets of reaching definitions. For example, reaching definitions at node E in Figure 6.2 are those at

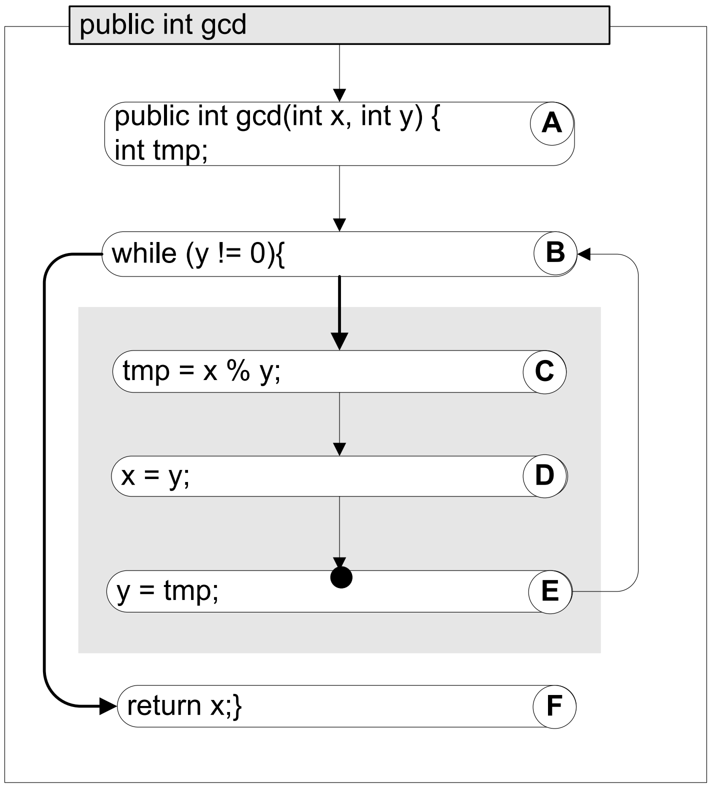

Figure 6.4: Calculating control dependence for node *E* in the control flow graph of the GCD method. Nodes *C*, *D*, and *E* in the gray region are post-dominated by *E*, i.e., execution of *E* is inevitable in that region. Node *B* has successors both within and outside the gray region, so it controls whether *E* is executed; thus *E* is control-dependent on *B*.

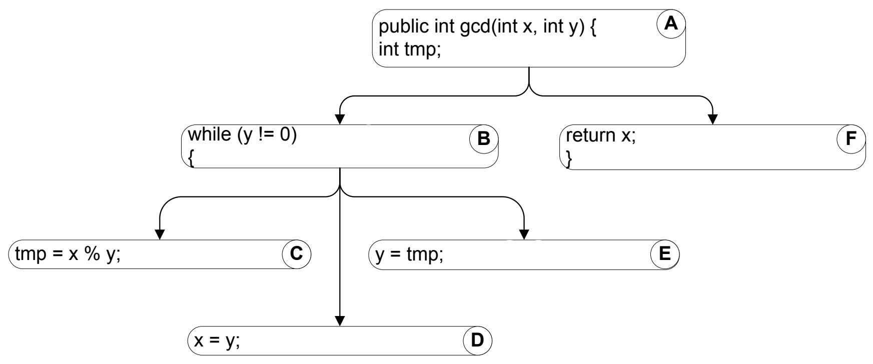

Figure 6.5: Control dependence tree of the GCD method. The loop test and the return statement are reached on every possible execution path, so they are control dependent only on the entry point. The statements within the loop are control dependent on the loop test.

node *D*, except that *D* adds a definition of y and replaces (kills) an earlier definition of y:

$$Reach(E) = (Reach(D) \setminus \{x_A\}) \cup \{x_D\}$$

This rule can be broken into two parts to make it a little more intuitive, and also more efficient to implement. The first part describes how node *E* receives values from its predecessor *D*, and the second describes how it modifies those values for its successors:

$$\begin{aligned} Reach(E) &= ReachOut(D) \\ ReachOut(D) &= (Reach(D) \setminus \{x_A\}) \cup \{x_D\} \end{aligned}$$

In this form, we can easily express what should happen at the head of the while loop (node *B* in Figure 6.2), where values may be transmitted both from the beginning of the procedure (node *A*) and through the end of the body of the loop (node *E*). The beginning of the procedure (node *A*) is treated as an initial definition of parameters and local variables. (If a local variable is declared but not initialized, it is treated as a definition to the special value "uninitialized.")

$$\begin{aligned} Reach(B) &= ReachOut(A) \cup ReachOut(E) \\ ReachOut(A) &= gen(A) = \{x_A, y_A, tmp_A\} \\ ReachOut(E) &= (ReachIn(E) \setminus \{y_A\}) \cup \{y_E\} \end{aligned}$$

In general, for any node *n* with predecessors *pred*(*n*),

$$\begin{aligned} \mathit{Reach}(n) &= \bigcup_{m \in \mathit{pred}(n)} \mathit{ReachOut}(m) \\ \mathit{ReachOut}(n) &= (\mathit{ReachIn}(n) \setminus \mathit{kill}(n)) \cup \mathit{gen}(n) \end{aligned}$$

Remarkably, the reaching definitions can be calculated simply and efficiently, first initializing the reaching definitions at each node in the control flow graph to the empty set, and then applying these equations repeatedly until the results stabilize. The algorithm is given as pseudocode in Figure 6.6.

6.3 Classic Analyses: Live and Avail

Reaching definition is a classic data flow analysis adapted from compiler construction to applications in software testing and analysis. Other classical data flow analyses from compiler construction can likewise be adapted. Moreover, they follow a common pattern that can be used to devise a wide variety of additional analyses.

Available expressions is another classical data flow analysis, used in compiler construction to determine when the value of a sub-expression can be saved and re-used rather than re-computed. This is permissible when the value of the sub-expression remains unchanged regardless of the execution path from the first computation to the second.

Available expressions can be defined in terms of paths in the control flow graph. An expression is *available* at a point if, for all paths through the control flow graph from procedure entry to that point, the expression has been computed and not subsequently modified. We say an expression is *generated* (becomes available) where it is computed, and is *killed* (ceases to be available) when the value of any part of it changes, e.g., when a new value is assigned to a variable in the expression.

As with reaching definitions, we can obtain an efficient analysis by describing the relation between the available expressions that reach a node in the control flow graph and those at adjacent nodes. The expressions that become available at each node (the *gen* set) and the expressions that change and cease to be available (the *kill* set) can be computed simply, without consideration of control flow. Their propagation to a node from its predecessors is described by a pair of set equations:

$$\begin{aligned} \mathit{Avail}(n) &= \bigcap_{m \in \mathit{pred}(n)} \mathit{AvailOut}(m) \\ \mathit{AvailOut}(n) &= (\mathit{Avail}(n) \setminus \mathit{kill}(n)) \cup \mathit{Gen}(n) \end{aligned}$$

The similarity to the set equations for reaching definitions is striking. Both propagate sets of values along the control flow graph in the direction of program execution (they are *forward* analyses), and both combine sets propagated along different control flow paths. However, reaching definitions combines propagated sets using set union,

forward analysis

Algorithm *Reaching definitions*

```
Input:   A control flow graph $G = (\text{nodes}, \text{edges})$
         pred$(n) = \{m \in$ nodes $\mid (m,n) \in$ edges$\}$
         succ$(m) = \{n \in$ nodes $\mid (m,n) \in$ edges$\}$
         gen$(n) = \{v_n\}$ if variable $v$ is defined at $n$, otherwise $\{\}$
         kill$(n)$ = all other definitions of $v$ if $v$ is defined at $n$, otherwise $\{\}$

Output:  Reach$(n)$ = the reaching definitions at node $n$


for $n \in$ nodes loop
    ReachOut$(n) = \{\}$ ;
end loop;
workList = nodes ;
while (workList $\neq \{\}$) loop
    // Take a node from worklist (e.g., pop from stack or queue)
    $n$ = any node in workList ;
    workList = workList $\setminus \{n\}$ ;

    oldVal = ReachOut$(n)$ ;

    // Apply flow equations, propagating values from predecessars
    Reach$(n) = \bigcup_{m \in \text{pred}(n)}$ ReachOut$(m)$;
    ReachOut$(n)$ = (Reach$(n) \setminus$ kill$(n)) \cup$ gen$(n)$ ;
    if ( ReachOut$(n) \neq$ oldVal ) then
        // Propagate changed value to successor nodes
        workList = workList $\cup$ succ$(n)$
    end if;
end loop;
```

Figure 6.6: An iterative work-list algorithm to compute reaching definitions by applying each flow equation until the solution stabilizes.

since a definition can reach a use along *any* execution path. Available expressions combines propagated sets using set intersection, since an expression is considered available at a node only if it reaches that node along *all* possible execution paths. Thus we say that, while reaching definitions is a *forward, any-path* analysis, available expressions is a *forward, all-paths* analysis. A work-list algorithm to implement available expressions analysis is nearly identical to that for reaching definitions, except for initialization and the flow equations, as shown in Figure 6.7.

all-paths analysis

any-path analysis

Applications of a forward, all-paths analysis extend beyond the common sub-expression detection for which the Avail algorithm was originally developed. We can think of available expressions as tokens that are propagated from where they are generated through the control flow graph to points where they might be used. We obtain different analyses by choosing tokens that represent some other property that becomes true (is generated) at some points, may become false (be killed) at some other points, and is evaluated (used) at certain points in the graph. By associating appropriate sets of tokens in gen and kill sets for a node, we can evaluate other properties that fit the pattern

> "*G* occurs on all execution paths leading to *U*, and there is no intervening occurrence of *K* between the last occurrence of *G* and *U*."

G, *K*, and *U* can be any events we care to check, so long as we can mark their occurrences in a control flow graph.

An example problem of this kind is variable initialization. We noted in Chapter 3 that Java requires a variable to be initialized before use on all execution paths. The analysis that enforces this rule is an instance of Avail. The tokens propagated through the control flow graph record which variables have been assigned initial values. Since there is no way to "uninitialize" a variable in Java, the kill sets are empty. Figure 6.8 repeats the source code of an example program from Chapter 3, and the corresponding control flow graph is shown with definitions and uses in Figure 6.9 and annotated with gen and kill sets for the initialized variable check in Figure 6.10.

Reaching definitions and available expressions are forward analyses, i.e., they propagate values in the direction of program execution. Given a control flow graph model, it is just as easy to propagate values in the opposite direction, backward from nodes that represent the next steps in computation. *Backward* analyses are useful for determining what happens after an event of interest. Live variables is a backward analysis that determines whether the value held in a variable may be subsequently used. Because a variable is considered live if there is any possible execution path on which it is used, a *backward*, *any-path* analysis is used.

backward analysis

A variable is live at a point in the control flow graph if, on some execution path, its current value may be used before it is changed. Live variables analysis can be expressed as set equations as before. Where *Reach* and *Avail* propagate values to a node from its predecessors, *Live* propagates values from the successors of a node. The gen sets are variables used at a node, and the kill sets are variables whose values are replaced. Set union is used to combine values from adjacent nodes, since a variable is live at a node if it is live at any of the succeeding nodes.

Algorithm *Available expressions*

```
Input:   A control flow graph $G = (\text{nodes}, \text{edges})$, with a distinguished root node $start$.
         $\text{pred}(n) = \{m \in \text{nodes} \mid (m,n) \in \text{edges}\}$
         $\text{succ}(m) = \{n \in \text{nodes} \mid (m,n) \in \text{edges}\}$
         $\text{gen}(n)$ = all expressions $e$ computed at node $n$
         $\text{kill}(n)$ = expressions $e$ computed anywhere, whose value is changed at $n$;
             $\text{kill}(start)$ is the set of all $e$.

Output:  $\text{Avail}(n)$ = the available expressions at node $n$
```

```
for $n \in$ nodes loop
    $\text{AvailOut}(n)$ = set of all $e$ defined anywhere ;
end loop;
workList = nodes ;
while (workList $\neq \{\}$) loop
    // Take a node from worklist (e.g., pop from stack or queue)
    $n$ = any node in workList ;
    workList = workList $\setminus \{n\}$ ;
    oldVal = $\text{AvailOut}(n)$ ;
    // Apply flow equations, propagating values from predecessors
    $\text{Avail}(n) = \bigcap_{m \in \text{pred}(n)} \text{AvailOut}(m)$;
    $\text{AvailOut}(n) = (\text{Avail}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$ ;
    if ( $\text{AvailOut}(n) \neq$ oldVal ) then
        // Propagate changes to successors
        workList = workList $\cup\ \text{succ}(n)$
    end if;
end loop;
```

Figure 6.7: An iterative work-list algorithm for computing available expressions.

```
/** A trivial method with a potentially uninitialized variable.
  * Java compilers reject the program. The compiler uses
  * data flow analysis to determine that there is a potential
  * (syntactic) execution path on which k is used before it
  * has been assigned an initial value.
  */
static void questionable() {
    int k;
    for (int i=0; i < 10; ++i) {
        if (someCondition(i)) {
            k = 0;
        } else {
            k += i;
        }
    }
    System.out.println(k);
}
}
```

Figure 6.8: Function questionable (repeated from Chapter 3) has a potentially uninitialized variable, which the Java compiler can detect using data flow analysis.

$$\begin{aligned} \mathit{Live}(n) &= \bigcup_{m \in \mathit{succ}(n)} \mathit{LiveOut}(m) \\ \mathit{LiveOut}(n) &= (\mathit{Live}(n) \setminus \mathit{kill}(n)) \cup \mathit{Gen}(n) \end{aligned}$$

These set equations can be implemented using a work-list algorithm analogous to those already shown for reaching definitions and available expressions, except that successor edges are followed in place of predecessors and vice versa.

Like available expressions analysis, live variables analysis is of interest in testing and analysis primarily as a pattern for recognizing properties of a certain form. A backward, any-paths analysis allows us to check properties of the following form:

> "After D occurs, there is at least one execution path on which G occurs with no intervening occurrence of K."

Again we choose tokens that represent properties, using gen sets to mark occurrences of G events (where a property becomes true) and kill sets to mark occurrences of K events (where a property ceases to be true).

One application of live variables analysis is to recognize *useless definitions*, that is, assigning a value that can never be used. A useless definition is not necessarily a program error, but is often symptomatic of an error. In scripting languages like *Perl* and *Python*, which do not require variables to be declared before use, a useless definition

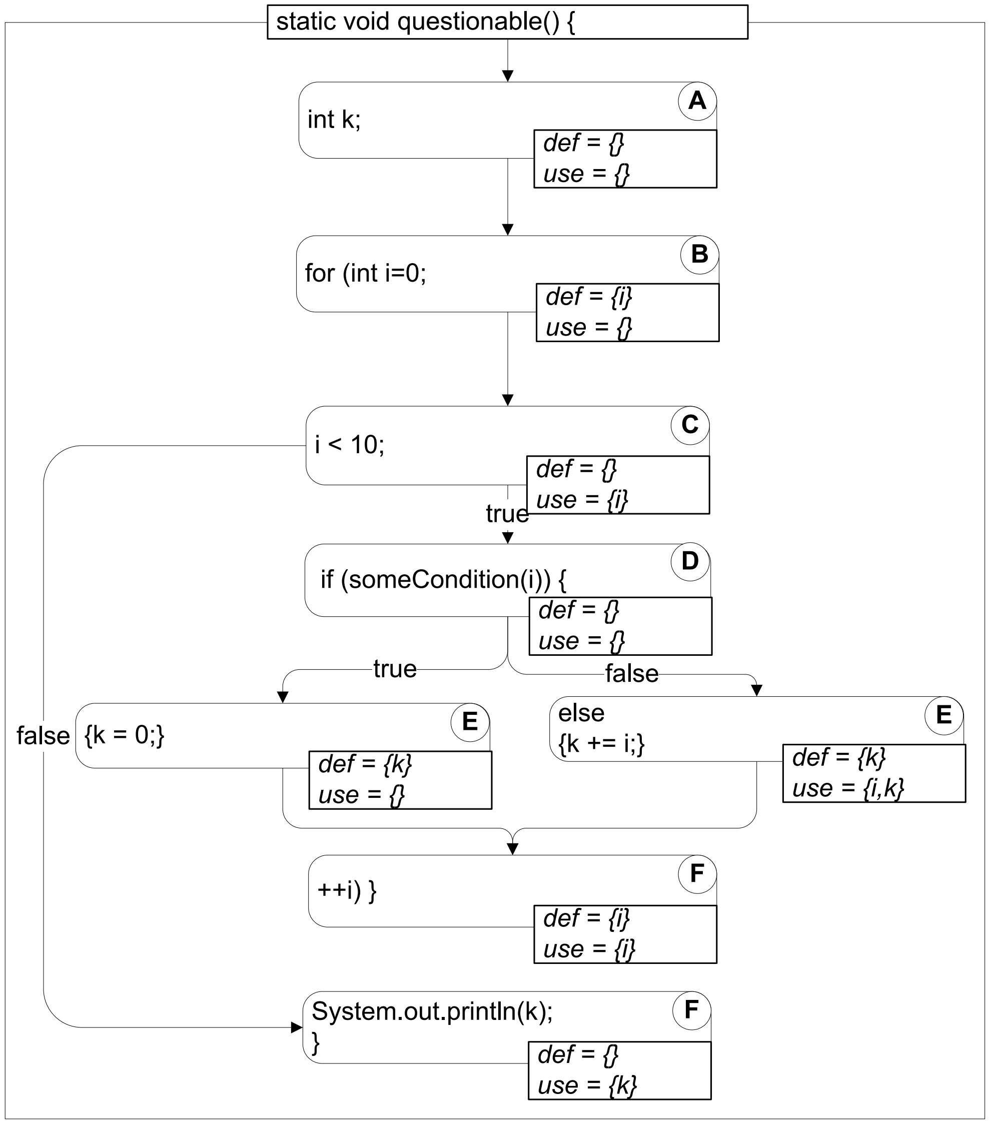

Figure 6.9: Control flow graph of the source code in Figure 6.8, annotated with variable definitions and uses.

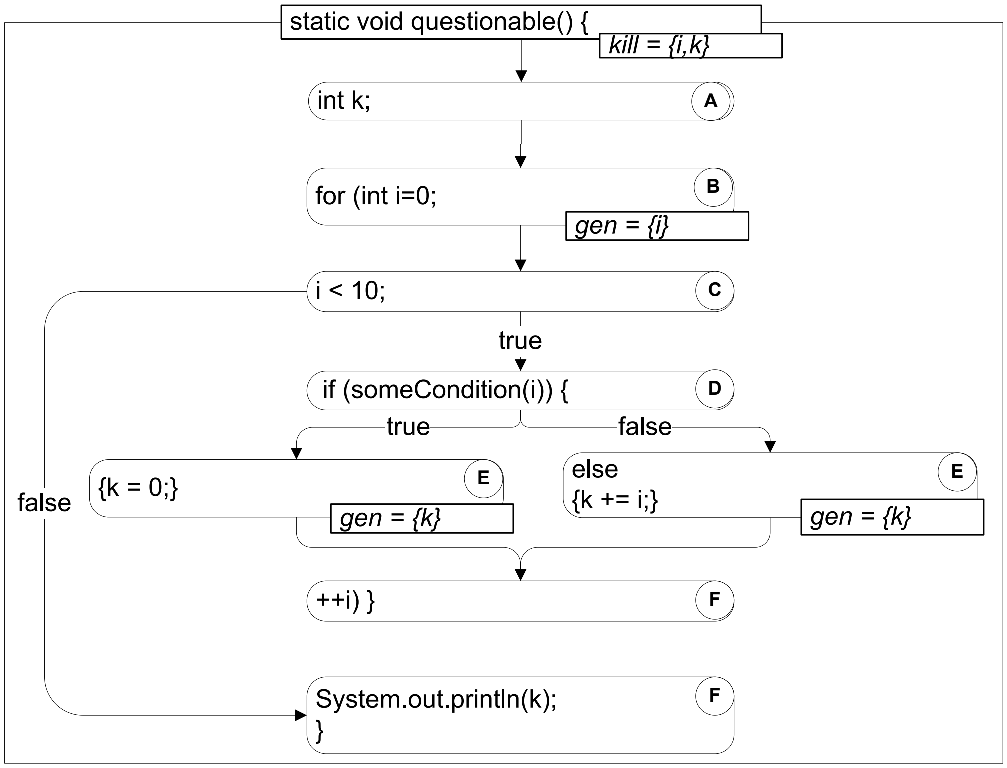

Figure 6.10: Control flow graph of the source code in Figure 6.8, annotated with gen and kill sets for checking variable initialization using a forward, all-paths *Avail* analysis. (Empty gen and kill sets are omitted.) The *Avail* set flowing from node G to node C will be $\{i,k\}$, but the *Avail* set flowing from node B to node C is $\{i\}$. The all-paths analysis intersects these values, so the resulting $Avail(C)$ is $\{i\}$. This value propagates through nodes C and D to node F, which has a use of k as well as a definition. Since $k \notin Avail(F)$, a possible use of an uninitialized variable is detected.

```
class SampleForm(FormData):
    """ Used with Python cgi module
        to hold and validate data
        from HTML form """

    fieldnames = ('name', 'email', 'comment')

    # Trivial example of validation.  The bug would be
    # harder to see in a real validation method.
    def validate(self):
        valid = 1;
        if self.name == "" : valid = 0
        if self.email == "" : vald = 0
        if self.comment == "" : valid = 0
        return valid
```

Figure 6.11: Part of a CGI program (web form processing) in Python. The misspelled variable name in the data validation method will be implicitly declared, and will not be rejected by the Python compiler or interpreter, which could allow invalid data to be treated as valid. The classic live variables data flow analysis can show that the assignment to valid is a useless definition, suggesting that the programmer probably intended to assign the value to a different variable.

typically indicates that a variable name has been misspelled, as in the CGI-bin script of Figure 6.11.

We have so-far seen a forward, any-path analysis (reaching definitions), a forward, all-paths analysis (available definitions), and a backward, any-path analysis (live variables). One might expect, therefore, to round out the repertoire of patterns with a backward, all-paths analysis, and this is indeed possible. Since there is no classical name for this combination, we will call it "inevitability," and use it for properties of the form

"After *D* occurs, *G* always occurs with no intervening occurrence of *K*"

or, informally,

"*D* inevitably leads to *G* before *K*"

Examples of inevitability checks might include ensuring that interrupts are re-enabled after executing an interrupt-handling routine in low-level code, files are closed after opening them, etc.

6.4 From Execution to Conservative Flow Analysis

Data flow analysis algorithms can be thought of as a kind of simulated execution. In place of actual values, much smaller sets of possible values are maintained (e.g., a single bit to indicate whether a particular variable has been initialized). All possible execution paths are considered at once, but the number of different states is kept small by associating just one summary state at each program point (node in the control flow graph). Since the values obtained at a particular program point when it is reached along one execution path may be different from those obtained on another execution path, the summary state must combine the different values. Considering flow analysis in this light, we can systematically derive a conservative flow analysis from a dynamic (that is, run-time) analysis.

As an example, consider the "taint-mode" analysis that is built into the programming language Perl. Taint mode is used to prevent some kinds of program errors that result from neglecting to fully validate data before using it, particularly where invalidated data could present a security hazard. For example, if a Perl script wrote to a file whose name was taken from a field in a web form, a malicious user could provide a full path to sensitive files. Taint mode detects and prevents use of the "tainted" web form input in a sensitive operation like opening a file. Other languages used in CGI scripts do not provide such a monitoring function, but we will consider how an analogous static analysis could be designed for a programming language like C.

When Perl is running in taint mode, it tracks the sources from which each variable value was derived, and distinguishes between safe and tainted data. Tainted data is any input (e.g., from a web form), and any data derived from tainted data. For example, if a tainted string is concatenated with a safe string, the result is a tainted string. One exception is that pattern-matching always returns safe strings, even when matching against tainted data — this reflects the common Perl idiom in which pattern matching is used to validate user input. Perl's taint mode will signal a program error if tainted data is used in a potentially dangerous way, e.g., as a file name to be opened.

Perl monitors values dynamically, tagging data values and propagating the tags through computation. Thus, it is entirely possible that a Perl script might run without errors in testing, but an unanticipated execution path might trigger a taint mode program error in production use. Suppose we want to perform a similar analysis, but instead of checking whether "tainted" data is used unsafely on a particular execution, we want to ensure that tainted data can never be used unsafely on any execution. We may also wish to perform the analysis on a language like C, for which run-time tagging is not provided and would be expensive to add. So, we can consider deriving a conservative, static analysis that is like Perl's taint mode except that it considers all possible execution paths.

A data flow analysis for taint would be a forward, any-path analysis with tokens representing tainted variables. The gen set at a program point would be a set containing any variable that is assigned a tainted value at that point. Sets of tainted variables would be propagated forward to a node from its predecessors, with set union where a node in the control flow graph has more than one predecessor (e.g., the head of a loop).

There is one fundamental difference between such an analysis and the classic data flow analyses we have seen so far: The gen and kill sets associated with a program point are not constants. Whether or not the value assigned to a variable is tainted (and thus whether the variable belongs in the gen set or in the kill set) depends on the set of tainted variables at that program point, which will vary during the course of the analysis.

There is a kind of circularity here — the gen set and kill set depend on the set of tainted variables, and the set of tainted variables may in turn depend on the gen and kill set. Such circularities are common in defining flow analyses, and there is a standard approach to determining whether they will make the analysis unsound. To convince ourselves that the analysis is sound, we must show that the output values computed by each flow equation are monotonically increasing functions of the input values. We will say more precisely what "increasing" means below.

The determination of whether a computed value is tainted will be a simple function of the set of tainted variables at a program point. For most operations of one or more arguments, the output is tainted if any of the inputs are tainted. As in Perl, we may designate one or a few operations (operations used to check an input value for validity) as taint removers. These special operations will simply always return an untainted value regardless of their inputs.

Suppose we evaluate the taintedness of an expression with the input set of tainted variables being $\{a,b\}$, and again with the input set of tainted variables being $\{a,b,c\}$. Even without knowing what the expression is, we can say with certainty that if the expression is tainted in the first evaluation, it must also be tainted in the second evaluation, in which the set of tainted input variables is larger. This also means that adding elements to the input tainted set can only add elements to the gen set for that point, or leave it the same, and conversely the kill set can only grow smaller or stay the same. We say that the computation of tainted variables at a point increases monotonically.

To be more precise, the monotonicity argument is made by arranging the possible values in a lattice. In the sorts of flow analysis framework considered here, the lattice is almost always made up of subsets of some set (the set of definitions, or the set of tainted variables, etc.); this is called a powerset lattice, because the powerset of set A is the set of all subsets of A. The bottom element of the lattice is the empty set, the top is the full set, and lattice elements are ordered by inclusion as in Figure 6.12. If we can follow the arrows in a lattice from element x to element y (e.g., from $\{a\}$ to $\{a,b,c\}$), then we say $y > x$. A function f is monotonically increasing if

powerset lattice

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

Not only are all of the individual flow equations for taintedness monotonic in this sense, but in addition the function applied to merge values where control flow paths come together is also monotonic:

$$A \supseteq B \Rightarrow A \cup C \supseteq B \cup C$$

If we have a set of data flow equations that is monotonic in this sense, and if we begin by initializing all values to the bottom element of the lattice (the empty set in this case), then we are assured that an iterative data flow analysis will converge on a unique minimum solution to the flow equations.

The standard data flow analyses for reaching definitions, live variables, and available expressions can all be justified in terms of powerset lattices. In the case of available expressions, though, and also in the case of other all-paths analyses such as the one we have called "inevitability," the lattice must be flipped over, with the empty set at the top

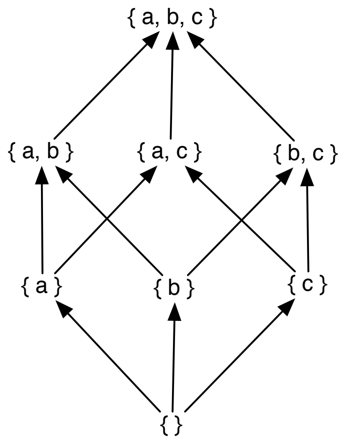

Figure 6.12: The powerset lattice of set $\{a,b,c\}$. The powerset contains all subsets of the set, and is ordered by set inclusion.

and the set of all variables or propositions at the bottom. (This is why we used the set of all tokens, rather than the empty set, to initialize the Avail sets in Figure 6.7.)

6.5 Data Flow Analysis with Arrays and Pointers

The models and flow analyses described above have been limited to simple scalar variables in individual procedures. Arrays and pointers (including object references and procedure arguments) introduce additional issues, because it is not possible in general to determine whether two accesses refer to the same storage location. For example, consider the following code fragment:

```
1           a[i] = 13;
2           k = a[j];
```

Are these two lines a definition-use pair? They are if the values of i and j are equal, which might be true on some executions and not on others. A static analysis cannot, in general, determine whether they are always, sometimes, or never equal, so a source of imprecision is necessarily introduced into data flow analysis.

Pointers and object references introduce the same issue, often in less obvious ways. Consider the following snippet:

```
1           a[2] = 42;
2           i = b[2];
```

It seems that there cannot possibly be a definition-use pair involving these two lines, since they involve none of the same variables. However, arrays in Java are dynamically allocated objects accessed through pointers. Pointers of any kind introduce the possibility of *aliasing*, that is, of two different names referring to the same storage location. For example, the two lines above might have been part of the following program fragment:

```
int [ ] a = new int[3];
int [ ] b = a;
a[2] = 42;
i = b[2];
```

Here a and b are *aliases*, two different names for the same dynamically allocated array object, and an assignment to part of a is also an assignment to part of b.

Δ alias

The same phenomenon, and worse, appears in languages with lower-level pointer manipulation. Perhaps the most egregious example is pointer arithmetic in C:

```
p = &b;
*(p + i) = k;
```

It is impossible to know which variable is defined by the second line. Even if we knew the value of i, the result is dependent on how a particular compiler arranges variables in memory.

Dynamic references and the potential for aliasing introduce uncertainty into data flow analysis. In place of a definition or use of a single variable, we may have a potential definition or use of a whole set of variables or locations that could be aliases of each other. The proper treatment of this uncertainty depends on the use to which the analysis will be put. For example, if we seek strong assurance that v is always initialized before it is used, we may not wish to treat an assignment to a potential alias of v as initialization, but we may wish to treat a use of a potential alias of v as a use of v.

A useful mental trick for thinking about treatment of aliases is to translate the uncertainty introduced by aliasing into uncertainty introduced by control flow. After all, data flow analysis already copes with uncertainty about which potential execution paths will actually be taken; an infeasible path in the control flow graph may add elements to an any-paths analysis or remove results from an all-paths analysis. It is usually appropriate to treat uncertainty about aliasing consistently with uncertainty about control flow. For example, considering again the first example of an ambiguous reference:

```
a[i] = 13;
k = a[j];
```

We can imagine replacing this by the equivalent code:

```
a[i] = 13;
if (i == j) {
    k = a[i];
} else {
    k = a[j];
}
```

In the (imaginary) transformed code, we could treat all array references as distinct, because the possibility of aliasing is fully expressed in control flow. Now, if we are using an any-paths analysis like reaching definitions, the potential aliasing will result in creating a definition-use pair. On the other hand, an assignment to a[j] would not kill a previous assignment to a[i]. This suggests that, for an any-path analysis, gen sets should include everything that might be referenced, but kill sets should include only what is definitely referenced.

If we were using an all-paths analysis, like available expressions, we would obtain a different result. Because the sets of available expressions are intersected where control flow merges, a definition of a[i] would make only that expression, and none of its potential aliases, available. On the other hand, an assignment to a[j] would kill a[i]. This suggests that, for an all-paths analysis, gen sets should include only what is definitely referenced, but kill sets should include all the possible aliases.

Even in analysis of a single procedure, the effect of other procedures must be considered at least with respect to potential aliases. Consider, for example, this fragment of a Java method:

```
public void transfer (CustInfo fromCust, CustInfo toCust) {

    PhoneNum fromHome = fromCust.gethomePhone();
    PhoneNum fromWork = fromCust.getworkPhone();

    PhoneNum toHome = toCust.gethomePhone();
    PhoneNum toWork = toCust.getworkPhone();
```

We cannot determine whether the two arguments fromCust and toCust are references to the same object without looking at the context in which this method is called. Moreover, we cannot determine whether fromHome and fromWork are (or could be) references to the same object without more information about how CustInfo objects are treated elsewhere in the program.

Sometimes it is sufficient to treat all non-local information as unknown. For example, we could treat the two CustInfo objects as potential aliases of each other, and similarly treat the four PhoneNum objects as potential aliases. Sometimes, though, large sets of aliases will result in analysis results that are so imprecise as to be useless. Therefore data flow analysis is often preceded by an inter-procedural analysis to calculate sets of aliases or the locations that each pointer or reference can refer to.

6.6 Inter-Procedural Analysis

Most important program properties involve more than one procedure, and as mentioned above, some inter-procedural analysis (e.g., to detect potential aliases) is often required as a prelude even to intra-procedural analysis. One might expect the inter-procedural analysis and models to be a natural extension of the intra-procedural analysis, following procedure calls and returns like intra-procedural control flow. Unfortunately this is seldom a practical option.

If we were to extend data flow models by following control flow paths through procedure calls and returns, using the control flow graph model and the call graph model together in the obvious way, we would observe many spurious paths. Figure 6.13

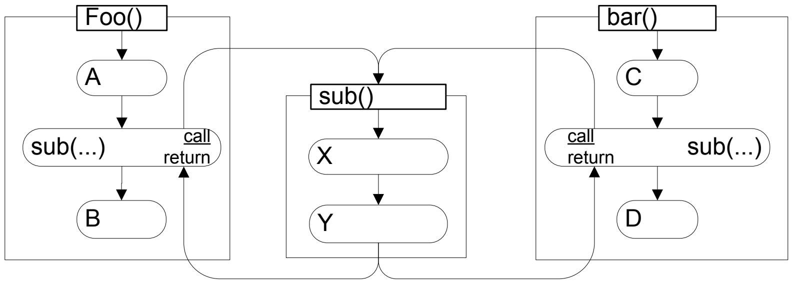

Figure 6.13: Spurious execution paths result when procedure calls and returns are treated as normal edges in the control flow graph. The path (A, X, Y, D) appears in the combined graph, but it does not correspond to an actual execution order.

illustrates the problem: Procedure foo and procedure bar each make a call on procedure sub. When procedure call and return are treated as if they were normal control flow, in addition to the execution sequences (A,X,Y,B) and (C,X,Y,D), the combined graph contains the impossible paths (A,X,Y,D) and (C,X,Y,B).

It is possible to represent procedure calls and returns precisely, e.g., by making a copy of the called procedure for each point at which it is called. This would result in a *context sensitive* analysis. The shortcoming of context sensitive analysis was already mentioned in the previous chapter: The number of different contexts in which a procedure must be considered could be exponentially larger than the number of procedures. In practice, a context sensitive analysis can be practical for a small group of closely related procedures (e.g., a single Java class), but is almost never a practical option for a whole program.

context sensitive analysis

Some inter-procedural properties are quite independent of context, and lend themselves naturally to analysis in a hierarchical, piecemeal fashion. Such a hierarchical analysis can be both precise and efficient. The analyses that are provided as part of normal compilation are often of this sort. The unhandled exception analysis of Java is a good example: Each procedure (method) is required to declare the exceptions that it may throw without handling. If method M calls method N in the same or another class, and if N can throw some exception, then M must either handle that exception or declare that it, too, can throw the exception. This analysis is simple and efficient because, when analyzing method M, the internal structure of N is irrelevant; only the results of the analysis at N (which, in Java, is also part of the signature of N) is needed.

Two conditions are necessary to obtain an efficient, hierarchical analysis like the exception analysis routinely carried out by Java compilers. First, the information needed to analyze a calling procedure must be small: It must not be proportional to the size of the called procedure, nor to the number of procedures that are directly or indirectly called. Second, it is essential that information about the called procedure be independent of the caller, i.e., it must be context independent. When these two conditions are true, it is straightforward to develop an efficient analysis that works upward from leaves of the call graph. (When there are cycles in the call graph from recursive or mutually recursive procedures, an iterative approach similar to data flow analysis algorithms can usually be devised.)

Unfortunately, not all important properties are amenable to hierarchical analysis. Potential aliasing information, which is essential to data flow analysis even within individual procedures, is one of those that are not. We have seen that potential aliasing can depend in part on the arguments passed to a procedure, so it does not have the context independence property required for an efficient hierarchical analysis. For such an analysis, additional sacrifices of precision must be made for the sake of efficiency.

Even when a property is context dependent, an analysis for that property may be context insensitive, although the context insensitive analysis will necessarily be less precise as a consequence of discarding context information. At the extreme, a linear time analysis can be obtained by discarding both context and control flow information.

flow-insensitive

Context and flow insensitive algorithms for pointer analysis typically treat each statement of a program as a constraint. For example, on encountering an assignment

```
1            x = y;
```

where y is a pointer, such an algorithm simply notes that x may refer to any of the same objects that y may refer to. $References(x) \supseteq References(y)$ is a constraint that is completely independent of the order in which statements are executed. A procedure call, in such an analysis, is just an assignment of values to arguments. Using efficient data structures for merging sets, some analyzers can process hundreds of thousand of lines of source code in a few seconds. The results are imprecise, but still much better than the worst-case assumption that any two compatible pointers might refer to the same object.

The best approach to inter-procedural pointer analysis will often lie somewhere between the astronomical expense of a precise, context and flow sensitive pointer analysis and the imprecision of the fastest context and flow insensitive analyses. Unfortunately there is not one best algorithm or tool for all uses. In addition to context and flow sensitivity, important design trade-offs include the granularity of modeling references (e.g., whether individual fields of an object are distinguished) and the granularity of modeling the program heap (that is, which allocated objects are distinguished from each other).

Summary

Data flow models are used widely in testing and analysis, and the data flow analysis algorithms used for deriving data flow information can be adapted to additional uses. The most fundamental model, complementary to models of control flow, represents the ways values can flow from the points where they are defined (computed and stored) to points where they are used.

Data flow analysis algorithms efficiently detect the presence of certain patterns in the control flow graph. Each pattern involves some nodes that initiate the pattern and some that conclude it, and some nodes that may interrupt it. The name “data flow analysis” reflects the historical development of analyses for compilers, but patterns may be used to detect other control flow patterns.

An any-path analysis determines whether there is any control flow path from the initiation to the conclusion of a pattern without passing through an interruption. An all-paths analysis determines whether every path from the initiation necessarily reaches a concluding node without first passing through an interruption. Forward analyses check for paths in the direction of execution, and backward analyses check for paths in the opposite direction. The classic data flow algorithms can all be implemented using simple work-list algorithms.

A limitation of data flow analysis, whether for the conventional purpose or to check other properties, is that it cannot distinguish between a path that can actually be executed and a path in the control flow graph that cannot be followed in any execution. A related limitation is that it cannot always determine whether two names or expressions refer to the same object.

Fully detailed data flow analysis is usually limited to individual procedures or a few closely related procedures, e.g., a single class in an object-oriented program. Analyses that span whole programs must resort to techniques that discard or summarize some information about calling context, control flow, or both. If a property is independent of calling context, a hierarchical analysis can be both precise and efficient. Potential aliasing is a property for which calling context is significant, and there is therefore a trade-off between very fast but imprecise alias analysis techniques and more precise but much more expensive techniques.

Further Reading

Data flow analysis techniques were developed originally for compilers, as a systematic way to detect opportunities for code-improving transformations, and to ensure that those transformations would not introduce errors into programs (an all-too-common experience with early optimizing compilers). The compiler construction literature remains an important source of reference information for data flow analysis, and the classic “Dragon Book” text [ASU86] is a good starting point.

Fosdick and Osterweil recognized the potential of data flow analysis to detect program errors and anomalies that suggested the presence of errors more than two decades ago [FO76]. While the classes of data flow anomaly detected by Fosdick and Osterweil’s system has largely been obviated by modern strongly-typed programming languages, they are still quite common in modern scripting and prototyping languages. Olender and Osterweil later recognized that the power of data flow analysis algorithms for recognizing execution patterns is not limited to properties of data flow, and developed a system for specifying and checking general sequencing properties [OO90, OO92].

Inter-procedural pointer analyses — either directly determining potential aliasing relations, or deriving a “points-to” relation from which aliasing relations can be derived — remains an area of active research. At one extreme of the cost-versus-precision spectrum of analyses are completely context and flow insensitive analyses like those described by Steensgaard [Ste96]. Many researchers have proposed refinements that obtain significant gains in precision at small costs in efficiency. An important direction for future work is obtaining acceptably precise analyses of a portion of a large program, either because a whole program analysis cannot obtain sufficient precision at acceptable cost, or because modern software development practices (e.g., incorporating externally developed components) mean that the whole program is never available in any case. Rountev et al present initial steps toward such analyses [RRL99]. A very readable overview of the state of the art and current research directions (circa 2001) is provided by Hind [Hin01].

Exercises

6.1. For a graph $G = (N, V)$ with a root $r \in N$, node m dominates node n if every path from r to n passes through m. The root node is dominated only by itself.

The relation can be restated using flow equations.

1. When dominance is restated using flow equations, will it be stated in the form of an any-path problem or an all-paths problem? Forward or backward? What are the tokens to be propagated, and what are the gen and kill sets?
2. Give a flow equation for $Dom(n)$.
3. If the flow equation is solved using an iterative data flow analysis, what should the set $Dom(n)$ be initialized to at each node n?
4. Implement an iterative solver for the dominance relation in a programming language of your choosing.

   The first line of input to your program is an integer between 1 and 100 indicating the number k of nodes in the graph. Each subsequent line of input will consist of two integers, m and n, representing an edge from node m to node n. Node 0 designates the root, and all other nodes are designated by integers between 0 and $k-1$. The end of the input is signaled by the pseudo-edge $(-1, -1)$.

   The output of your program should be a sequences of lines, each containing two integers separated by blanks. Each line represents one edge of the *Dom* relation of the input graph.
5. The *Dom* relation itself is not a tree. The immediate dominators relation is a tree. Write flow equations to calculate immediate dominators, and then modify the program from part d to compute the immediate dominance relation.

6.2. Write flow equations for inevitability, a backward, all-paths intra-procedural analysis. Event (or program point) q is inevitable at program point p if every execution path from p to a normal exit point passes through q.