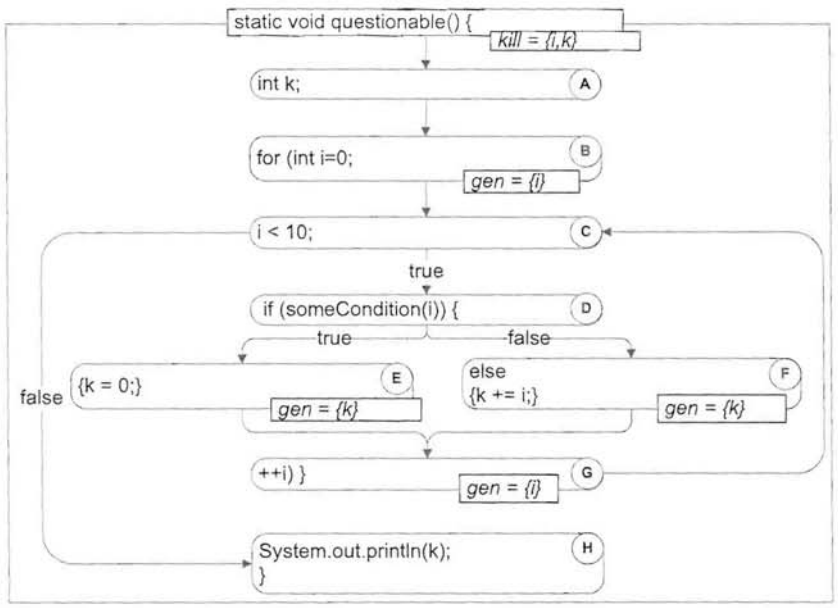

Figure 6.10: Control flow graph of the source code in Figure 6.8, annotated with gen and kill sets for checking variable initialization using a forward, all-paths Avail *analysis. (Empty gen and kill sets are omitted.) The* Avail *set flowing from node G to node C will be $\{i,k\}$, but the* Avail *set 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.*

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.

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.

$$Live(n) = \bigcup_{m \in succ(n)} LiveOut(m)$$

$$LiveOut(n) = (Live(n) \setminus kill(n)) \cup Gen(n)$$

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 typically indicates that a variable name has been misspelled, as in the common gateway interface (CGI) 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 reenabled after executing an interrupt-handling routine in low-level code, files are closed after opening them, and so on.

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 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 evalu-

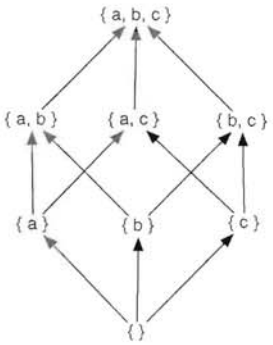

Figure 6.12: The powerset lattice of set $\{a,b,c\}$. The powerset contains all subsets of the set and is ordered by set inclusion.

ation, 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 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 in the preceding section 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:

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

Δ alias

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.

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

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