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:

```
a[i] = 13;
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:

```
a[2] = 42;
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];
```

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:

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

It is impossible to know which variable is defined by the second line. Even if we know 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-path 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 nonlocal 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 interprocedural analysis to calculate sets of aliases or the locations that each pointer or reference can refer to.

6.6 Interprocedural Analysis

Most important program properties involve more than one procedure, and as mentioned earlier, some interprocedural analysis (e.g., to detect potential aliases) is often required as a prelude even to intraprocedural analysis. One might expect the interprocedural analysis and models to be a natural extension of the intraprocedural analysis, following procedure calls and returns like intraprocedural 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 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).

context-sensitive analysis

It is possible to represent procedure calls and returns precisely, for example 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.

Some interprocedural properties are quite independent of context and lend themselves naturally to analysis in a hierarchical, piecemeal fashion. Such a hierarchical

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.

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) are 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 either to the size of the called procedure, or 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; that is, 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 thousands 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 interprocedural 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. 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 originally developed 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].

Interprocedural 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].