Chapter 12

Interprocedural Analysis

In this chapter, we motivate the importance of interprocedural analysis by discussing a number of important optimization problems that cannot be solved with intraprocedural analysis. We begin by describing the common forms of interprocedural analysis and explaining the difficulties in their implementation. We then describe applications for interprocedural analysis. For widely used programming languages like C and Java, pointer alias analysis is key to any interprocedural analysis. Thus, for much of the chapter, we discuss techniques needed to compute pointer aliases. To start, we present Datalog, a notation that greatly hides the complexity of an efficient pointer analysis. We then describe an algorithm for pointer analysis, and show how we use the abstraction of binary decision diagrams (BDD's) to implement the algorithm efficiently.

Most compiler optimizations, including those described in Chapters 9, 10, and 11, are performed on procedures one at a time. We refer to such analyses as *intraprocedural.* These analyses conservatively assume that procedures invoked may alter the state of all the variables visible to the procedures and that they may create all possible side effects, such as modifying any of the variables visible to the procedure or generating exceptions that cause the unwinding of the call stack. Intraprocedural analysis is thus relatively simple, albeit imprecise. Some optimizations do not need interprocedural analysis, while others may yield almost no useful information without it.

An interprocedural analysis operates across an entire program, flowing information from the caller to its callees and vice versa. One relatively simple but useful technique is to *inline* procedures, that is, to replace a procedure invocation by the body of the procedure itself with suitable modifications to account for parameter passing and the return value. This method is applicable only if we know the target of the procedure call.

If procedures are invoked indirectly through a pointer or via the method-dispatch mechanism prevalent in object-oriented programming, analysis of the program's pointers or references can in some cases determine the targets of the indirect invocations. If there is a unique target, inlining can be applied.

Even if a unique target is determined for each procedure invocation, inlining must be applied judiciously. In general, it is not possible to inline recursive procedures directly, and even without recursion, inlining can expand the code size exponentially.

12.1 Basic Concepts

In this section, we introduce call graphs — graphs that tell us which procedures can call which. We also introduce the idea of "context sensitivity," where data-flow analyses are required to take cognizance of what the sequence of procedure calls has been. That is, context-sensitive analysis includes (a synopsis of) the current sequence of activation records on the stack, along with the current point in the program, when distinguishing among different "places" in the program.

12.1.1 Call Graphs

A *call graph* for a program is a set of nodes and edges such that

1. There is one node for each procedure in the program.
2. There is one node for each *call site*, that is, a place in the program where a procedure is invoked.
3. If call site c may call procedure p, then there is an edge from the node for c to the node for p.

Many programs written in languages like C and Fortran make procedure calls directly, so the call target of each invocation can be determined statically. In that case, each call site has an edge to exactly one procedure in the call graph. However, if the program includes the use of a procedure parameter or function pointer, the target generally is not known until the program is run and, in fact, may vary from one invocation to another. Then, a call site can link to many or all procedures in the call graph.

Indirect calls are the norm for object-oriented programming languages. In particular, when there is overriding of methods in subclasses, a use of method m may refer to any of a number of different methods, depending on the subclass of the receiver object to which it was applied. The use of such *virtual* method invocations means that we need to know the type of the receiver before we can determine which method is invoked.

Example 12.1: Figure 12.1 shows a C program that declares `pf` to be a global pointer to a function whose type is "integer to integer." There are two functions of this type, `fun1` and `fun2`, and a main function that is not of the type that `pf` points to. The figure shows three call sites, denoted `c1`, `c2`, and `c3`; the labels are not part of the program.

```
          int (*pf)(int);

          int fun1(int x) {
              if (x < 10)
c1:               return (*pf)(x+1);
              else
                  return x;
          }

          int fun2(int y) {
              pf = &fun1;
c2:           return (*pf)(y);
          }

          void main() {
              pf = &fun2;
c3:           (*pf)(5);
          }
```

Figure 12.1: A program with a function pointer

The simplest analysis of what `pf` could point to would simply observe the types of functions. Functions `fun1` and `fun2` are of the same type as what `pf` points to, while `main` is not. Thus, a conservative call graph is shown in Fig. 12.2(a). A more careful analysis of the program would observe that `pf` is made to point to `fun2` in `main` and is made to point to `fun1` in `fun2`. But there are no other assignments to any pointer, so, in particular, there is no way for `pf` to point to `main`. This reasoning yields the same call graph as Fig. 12.2(a).

An even more precise analysis would say that at `c3`, it is only possible for `pf` to point to `fun2`, because that call is preceded immediately by that assignment to `pf`. Similarly, at `c2` it is only possible for `pf` to point to `fun1`. As a result, the initial call to `fun1` can come only from `fun2`, and `fun1` does not change `pf`, so whenever we are within `fun1`, `pf` points to `fun1`. In particular, at `c1`, we can be sure `pf` points to `fun1`. Thus, Fig. 12.2(b) is a more precise, correct call graph. □

In general, the presence of references or pointers to functions or methods requires us to get a static approximation of the potential values of all procedure parameters, function pointers, and receiver object types. To make an accurate approximation, interprocedural analysis is necessary. The analysis is iterative, starting with the statically observable targets. As more targets are discovered, the analysis incorporates the new edges into the call graph and repeats discovering more targets until convergence is reached.

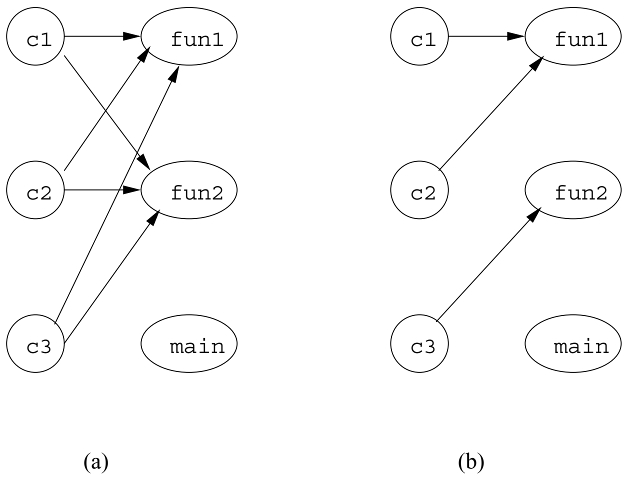

Figure 12.2: Call graphs derived from Fig. 12.1

12.1.2 Context Sensitivity

Interprocedural analysis is challenging because the behavior of each procedure is dependent upon the context in which it is called. Example 12.2 uses the problem of interprocedural constant propagation on a small program to illustrate the significance of contexts.

Example 12.2 : Consider the program fragment in Fig. 12.3. Function f is invoked at three call sites: `c1`, `c2` and `c3`. Constant 0 is passed in as the actual parameter at `c1`, and constant 243 is passed in at `c2` and `c3` in each iteration; the constants 1 and 244 are returned, respectively. Thus, function f is invoked with a constant in each of the contexts, but the value of the constant is context-dependent.

As we shall see, it is not possible to tell that `t1`, `t2`, and `t3` each are assigned constant values (and thus so is $X[i]$), unless we recognize that when called in context `c1`, f returns 1, and when called in the other two contexts, f returns 244. A naive analysis would conclude that f can return either 1 or 244 from any call. □

One simplistic but extremely inaccurate approach to interprocedural analysis, known as *context-insensitive analysis*, is to treat each call and return statement as "goto" operations. We create a *super* control-flow graph where, besides the normal intraprocedural control flow edges, additional edges are created connecting

1. Each call site to the beginning of the procedure it calls, and

2. The return statements back to the call sites.[1]

[1]The return is actually to the instruction following the call site.

```
           for (i = 0; i < n; i++) {
c1:              t1 = f(0);
c2:              t2 = f(243);
c3:              t3 = f(243);
                 X[i] = t1+t2+t3;
           }

           int f (int v) {
                 return (v+1);
           }
```

Figure 12.3: A program fragment illustrating the need for context-sensitive analysis

Assignment statements are added to assign each actual parameter to its corresponding formal parameter and to assign the returned value to the variable receiving the result. We can then apply a standard analysis intended to be used within a procedure to the super control-flow graph to find context-insensitive interprocedural results. While simple, this model abstracts out the important relationship between input and output values in procedure invocations, causing the analysis to be imprecise.

Example 12.3: The super control-flow graph for the program in Fig. 12.3 is shown in Figure 12.4. Block B_6 is the function f. Block B_3 contains the call site `c1`; it sets the formal parameter v to 0 and then jumps to the beginning of f, at B_6. Similarly, B_4 and B_5 represent the call sites `c2` and `c3`, respectively. In B_4, which is reached from the end of f (block B_6), we take the return value from f and assign it to `t1`. We then set formal parameter v to 243 and call f again, by jumping to B_6. Note that there is no edge from B_3 to B_4. Control must flow through f on the way from B_3 to B_4.

B_5 is similar to B_4. It receives the return from f, assigns the return value to `t2`, and initiates the third call to f. Block B_7 represents the return from the third call and the assignment to $X[i]$.

If we treat Fig. 12.4 as if it were the flow graph of a single procedure, then we would conclude that coming into B_6, v can have the value 0 or 243. Thus, the most we can conclude about `retval` is that it is assigned 1 or 244, but no other value. Similarly, we can only conclude about `t1`, `t2`, and `t3` that they can each be either 1 or 244. Thus, $X[i]$ appears to be either 3, 246, 489, or 732. In contrast, a context-sensitive analysis would separate the results for each of the calling contexts and produces the intuitive answer described in Example 12.2: `t1` is always 1, `t2` and `t3` are always 244, and $X[i]$ is 489. □

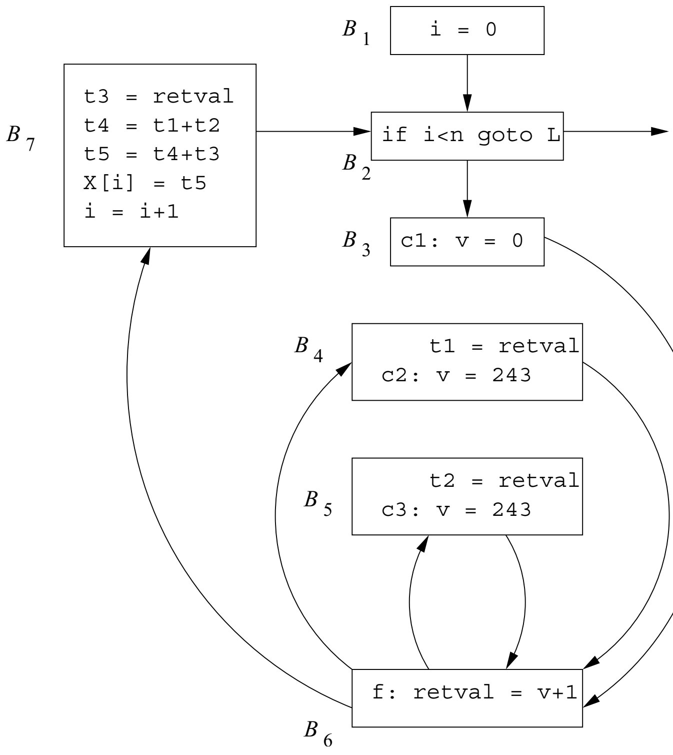

Figure 12.4: The control-flow graph for Fig. 12.3, treating function calls as control flow

12.1.3 Call Strings

In Example 12.2, we can distinguish among the contexts by just knowing the call site that calls the procedure f. In general, a calling context is defined by the contents of the entire call stack. We refer to the string of call sites on the stack as the *call string*.

Example 12.4: Figure 12.5 is a slight modification of Fig. 12.3. Here we have replaced the calls to f by calls to g, which then calls f with the same argument. There is an additional call site, c4, where g calls f.

There are three call strings to f: $(\texttt{c1}, \texttt{c4})$, $(\texttt{c2}, \texttt{c4})$, and $(\texttt{c3}, \texttt{c4})$. As we see in this example, the value of v in function f depends not on the immediate or last site c4 on the call string. Rather, the constants are determined by the first element in each of the call strings. □

Example 12.4 illustrates that information relevant to the analysis can be introduced early in the call chain. In fact, it is sometimes necessary to consider the entire call string to compute the most precise answer, as illustrated in Example 12.5.

```
         for (i = 0; i < n; i++) {
c1:            t1 = g(0);
c2:            t2 = g(243);
c3:            t3 = g(243);
               X[i] = t1+t2+t3;
         }

        int g (int v) {
c4:          return f(v);
        }

        int f (int v) {
            return (v+1);
        }
```

Figure 12.5: Program fragment illustrating call strings

```
        for (i = 0; i < n; i++) {
c1:         t1 = g(0);
c2:         t2 = g(243);
c3:         t3 = g(243);
            X[i] =  t1+t2+t3;
        }

        int g (int v) {
            if (v > 1) {
c4:             return g(v-1);
            } else {
c5:             return f(v);
        }

        int f (int v) {
            return (v+1);
        }
```

Figure 12.6: Recursive program requiring analysis of complete call strings

Example 12.5: This example illustrates how the ability to reason about unbounded call strings can yield more precise results. In Fig. 12.6 we see that if g is called with a positive value c, then g will be invoked recursively c times. Each time g is called, the value of its parameter v decreases by 1. Thus, the value of g's parameter v in the context whose call string is $\texttt{c2}(\texttt{c4})^n$ is $243 - n$. The effect of g is thus to increment 0 or any negative argument by 1, and to return 2 on any argument 1 or greater.

There are three possible call strings for f. If we start with the call at `c1`, then g calls f immediately, so $(\texttt{c1}, \texttt{c5})$ is one such string. If we start at `c2` or `c3`, then we call g a total of 243 times, and then call f. These call strings are $(\texttt{c2}, \texttt{c4}, \texttt{c4}, \dots, \texttt{c5})$ and $(\texttt{c3}, \texttt{c4}, \texttt{c4}, \dots, \texttt{c5})$, where in each case there are 242 `c4`'s in the sequence. In the first of these contexts, the value of f's parameter v is 0, while in the other two contexts it is 1. □

In designing a context-sensitive analysis, we have a choice in precision. For example, instead of qualifying the results by the full call string, we may just choose to distinguish between contexts by their k most immediate call sites. This technique is known as k-limiting context analysis. Context-insensitive analysis is simply a special case of k-limiting context analysis, where k is 0. We can find all the constants in Example 12.2 using a 1-limiting analysis and all the constants in Example 12.4 using a 2-limiting analysis. However, no k-limiting analysis can find all the constants in Example 12.5, provided the constant 243 were replaced by two different and arbitrarily large constants.

Instead of choosing a fixed value k, another possibility is to be fully context sensitive for all *acyclic* call strings, which are strings that contain no recursive cycles. For call strings with recursion, we can collapse all recursive cycles, in order to bound the number of different contexts analyzed. In Example 12.5, the calls initiated at call site `c2` may be approximated by the call string: $(\texttt{c2}, \texttt{c4}^*, \texttt{c5})$. Note that, with this scheme, even for programs without recursion, the number of distinct calling contexts can be exponential in the number of procedures in the program.

12.1.4 Cloning-Based Context-Sensitive Analysis

Another approach to context-sensitive analysis is to clone the procedure conceptually, one for each unique context of interest. We can then apply a context-insensitive analysis to the cloned call graph. Examples 12.6 and 12.7 show the equivalent of a cloned version of Examples 12.4 and 12.5, respectively. In reality, we do not need to clone the code, we can simply use an efficient internal representation to keep track of the analysis results of each clone.

Example 12.6: The cloned version of Fig. 12.5 is shown in Fig. 12.7. Because every calling context refers to a distinct clone, there is no confusion. For example, `g1` receives 0 as input and produces 1 as output, and `g2` and `g3` both receive 243 as input and produce 244 as output. □

```
          for (i = 0; i < n; i++) {
c1:                t1 = g1(0);
c2:                t2 = g2(243);
c3:                t3 = g3(243);
                   X[i] =  t1+t2+t3;
          }
         int g1 (int v) {
c4.1:              return f1(v);
         }
         int g2 (int v) {
c4.2:              return f2(v);
         }
         int g3 (int v) {
c4.3:              return f3(v);
         }

         int f1 (int v) {
             return (v+1);
         }
         int f2 (int v) {
             return (v+1);
         }
         int f3 (int v) {
             return (v+1);
         }
```

Figure 12.7: Cloned version of Fig. 12.5

Example 12.7 : The cloned version of Example 12.5 is shown in Fig. 12.8. For procedure g, we create a clone to represent all instances of g that are first called from sites `c1`, `c2`, and `c3`. In this case, the analysis would determine that the invocation at call site `c1` returns 1, assuming the analysis can deduce that with $v = 0$, the test $v > 1$ fails. This analysis does not handle recursion well enough to produce the constants for call sites `c2` and `c3`, however. □

12.1.5 Summary-Based Context-Sensitive Analysis

Summary-based interprocedural analysis is an extension of region-based analysis. Basically, in a summary-based analysis each procedure is represented by a concise description ("summary") that encapsulates some observable behavior of the procedure. The primary purpose of the summary is to avoid reanalyzing a procedure's body at every call site that may invoke the procedure.

Let us first consider the case where there is no recursion. Each procedure is modeled as a region with a single entry point, with each caller-callee pair sharing

```
           for (i = 0; i < n; i++) {
c1:              t1 = g1(0);
c2:              t2 = g2(243);
c3:              t3 = g3(243);
                 X[i] =  t1+t2+t3;
           }

          int g1 (int v) {
              if (v > 1) {
c4.1:             return g1(v-1);
              } else {
c5.1:             return f1(v);
          }}

          int g2 (int v) {
              if (v > 1) {
c4.2:             return g2(v-1);
              } else {
c5.2:             return f2(v);
          }}

          int g3 (int v) {
              if (v > 1) {
c4.3:             return g3(v-1);
              } else {
c5.3:             return f3(v);
          }}

          int f1 (int v) {
              return (v+1);
          }
          int f2 (int v) {
              return (v+1);
          }
          int f3 (int v) {
              return (v+1);
          }
```

Figure 12.8: Cloned version of Fig. 12.6

an outer-inner region relationship. The only difference from the intraprocedural version is that, in the interprocedural case, a procedure region can be nested inside several different outer regions.

The analysis consists of two parts:

1. A bottom-up phase that computes a transfer function to summarize the effect of a procedure, and

2. A top-down phase that propagates caller information to compute results of the callees.

To get fully context-sensitive results, information from different calling contexts must propagate down to the callees individually. For a more efficient, but less precise calculation, information from all callers can be combined, using a meet operator, then propagated down to the callees.

Example 12.8 : For constant propagation, each procedure is summarized by a transfer function specifying how it would propagate constants through its body. In Example 12.2, we can summarize f as a function that, given a constant c as an actual parameter to v, returns the constant $c+1$. Based on this information, the analysis would determine that `t1`, `t2`, and `t3` have the constant values 1, 244, and 244, respectively. Note that this analysis does not suffer the inaccuracy due to unrealizable call strings.

Recall that Example 12.4 extends Example 12.2 by having g call f. Thus, we could conclude that the transfer function for g is the same as the transfer function for f. Again we conclude that `t1`, `t2`, and `t3` have the constant values 1, 244, and 244, respectively.

Now, let us consider what is the value of parameter v in function f for Example 12.2. As a first cut, we can combine all the results for all calling contexts. Since v may have values 0 or 243, we can simply conclude that v is not a constant. This conclusion is fair, because there is no constant that can replace v in the code.

If we desire more precise results, we can compute specific results for contexts of interest. Information must be passed down from the context of interest to determine the context-sensitive answer. This step is analogous to the top-down pass in region-based analysis. For example, the value of v is 0 at call site `c1` and 243 at sites `c2` and `c3`. To get the advantage of constant propagation within f, we need to capture this distinction by creating two clones, with the first specialized for input value 0 and the latter with value 243, as shown in Fig. 12.9. □

With Example 12.8, we see that, in the end, if we wish to compile the code differently in different contexts, we still need to clone the code. The difference is that in the cloning-based approach, cloning is performed prior to the analysis, based on the call strings. In the summary-based approach, the cloning is performed after the analysis, using the analysis results as a basis.

```
             for (i = 0; i < n; i++) {
    c1:              t1 = f0(0);
    c2:              t2 = f243(243);
    c3:              t3 = f243(243);
                     X[i] = t1+t2+t3;
             }

            int f0 (int v) {
                return (1);
            }

            int f243 (int v) {
                return (244);
            }
```

Figure 12.9: Result of propagating all possible constant arguments to the function f

Even if cloning is not applied, in the summary-based approach inferences about the effect of a called procedure are made accurately, without the problem of unrealizable paths.

Instead of cloning a function, we could also inline the code. Inlining has the additional effect of eliminating the procedure-call overhead as well.

We can handle recursion by computing the fixedpoint solution. In the presence of recursion, we first find the strongly connected components in the call graph. In the bottom-up phase, we do not visit a strongly connected component unless all its successors have been visited. For a nontrivial strongly connected component, we iteratively compute the transfer functions for each procedure in the component until convergence is reached; that is, we iteratively update the transfer functions until no more changes occur.

12.1.6 Exercises for Section 12.1

Exercise 12.1.1: In Fig. 12.10 is a C program with two function pointers, p and q. N is a constant that could be less than or greater than 10. Note that the program results in an infinite sequence of calls, but that is of no concern for the purposes of this problem.

a) Identify all the call sites in this program.

b) For each call site, what can p point to? What can q point to?

c) Draw the call graph for this program.

! d) Describe all the call strings for f and g.

```
int (*p)(int);
int (*q)(int);

int f(int i) {
    if (i < 10)
        {p = &g; return (*q)(i);}
    else
        {p = &f; return (*p)(i);}
}

int g(int j) {
    if (j < 10)
        {q = &f; return (*p)(j);}
    else
        {q = &g; return (*q)(j);}
}

void main() {
    p = &f;
    q = &g;
    (*p)((*q)(N));
}
```

Figure 12.10: Program for Exercise 12.1.1

Exercise 12.1.2: In Fig. 12.11 is a function `id` that is the "identity function"; it returns exactly what it is given as an argument. We also see a code fragment consisting of a branch and following assignment that sums $x + y$.

a) Examining the code, what can we tell about the value of z at the end?

b) Construct the flow graph for the code fragment, treating the calls to `id` as control flow.

c) If we run a constant-propagation analysis, as in Section 9.4, on your flow graph from (b), what constant values are determined?

d) What are all the call sites in Fig. 12.11?

e) What are all the contexts in which `id` is called?

f) Rewrite the code of Fig. 12.11 by cloning a new version of `id` for each context in which it is called.

g) Construct the flow graph of your code from (f), treating the calls as control flow.

```
int id(int x) { return x;}

    ...
if (a == 1) { x = id(2); y = id(3); }
else        { x = id(3); y = id(2); }
z = x+y;
    ...
```

Figure 12.11: Code fragment for Exercise 12.1.2

h) Perform a constant-propagation analysis on your flow graph from (g). What constant values are determined now?

12.2 Why Interprocedural Analysis?

Given how hard interprocedural analysis is, let us now address the important problem of why and when we wish to use interprocedural analysis. Although we used constant propagation to illustrate interprocedural analysis, this interprocedural optimization is neither readily applicable nor particularly beneficial when it does occur. Most of the benefits of constant propagation can be obtained simply by performing intraprocedural analysis and inlining procedure calls of the most frequently executed sections of code.

However, there are many reasons why interprocedural analysis is essential. Below, we describe several important applications of interprocedural analysis.

12.2.1 Virtual Method Invocation

As mentioned above, object-oriented programs have many small methods. If we only optimize one method at a time, then there are few opportunities for optimization. Resolving method invocation enables optimization. A language like Java dynamically loads its classes. As a result, we do not know at compile-time to which of (perhaps) many methods named m a use of "m" refers in an invocation such as $x.m()$.

Many Java implementations use a just-in-time compiler to compile its bytecodes at run time. One common optimization is to profile the execution and determine which are the common receiver types. We can then inline the methods that are most frequently invoked. The code includes a dynamic check on the type and executes the inlined methods if the run-time object has the expected type.

Another approach to resolving uses of a method name m is possible as long as all the source code is available at compile time. Then, it is possible to perform an interprocedural analysis to determine the object types. If the type for a variable x turns out to be unique, then a use of $x.m()$ can be resolved.

We know exactly what method m refers to in this context. In that case, we can in-line the code for this m, and the compiler does not even have to include a test for the type of x.

12.2.2 Pointer Alias Analysis

Even if we do not wish to perform interprocedural versions of the common data-flow analyses like reaching definitions, these analyses can in fact benefit from interprocedural pointer analysis. All the analyses presented in Chapter 9 apply only to local scalar variables that cannot have aliases. However, use of pointers is common, especially in languages like C. By knowing whether pointers can be *aliases* (can point to the same location), we can improve the accuracy of the techniques from Chapter 9.

Example 12.9: Consider the following sequence of three statements, which might form a basic block:

```
*p = 1;
*q = 2;
 x = *p;
```

Without knowing if p and q can point to the same location — that is, whether they can be aliases — we cannot conclude that x is equal to 1 at the end of the block. □

12.2.3 Parallelization

As discussed in Chapter 11, the most effective way to parallelize an application is to find the coarsest granularity of parallelism, such as that found in the outermost loops of a program. For this task, interprocedural analysis is of great importance. There is a significant difference between *scalar* optimizations (those based on values of simple variables, as discussed in Chapter 9) and parallelization. In parallelization, just one spurious data dependence can render an entire loop not parallelizable, and greatly reduce the effectiveness of the optimization. Such amplification of inaccuracies is not seen in scalar optimizations. In scalar optimization, we only need to find the majority of the optimization opportunities. Missing one opportunity or two seldom makes much of a difference.

12.2.4 Detection of Software Errors and Vulnerabilities

Interprocedural analysis is not only important for optimizing code. The same techniques can be used to analyze existing software for many kinds of coding errors. These errors can render software unreliable; coding errors that hackers can exploit to take control of, or otherwise damage, a computer system can pose significant security vulnerability risks.

Static analysis is useful in detecting occurrences of many common error patterns. For example, a data item must be guarded by a lock. As another example, disabling an interrupt in the operating system must be followed by a re-enabling of the interrupt. Since a significant source of errors is the inconsistencies that span procedure boundaries, interprocedural analysis is of great importance. PREfix and Metal are two practical tools that use interprocedural analysis effectively to find many programming errors in large programs. Such tools find errors statically and can improve software reliability greatly. However, these tools are both incomplete and unsound, in the sense that they may not find all errors, and not all reported warnings are real errors. Unfortunately, the interprocedural analysis used is sufficiently imprecise that, were the tools to report all potential errors, the large number of false warnings would render the tools unusable. Nevertheless, even though these tools are not perfect, their systematic use has been shown to greatly improve software reliability.

When it comes to security vulnerabilities, it is highly desirable that we find all the potential errors in a program. In 2006, two of the "most popular" forms of intrusions used by hackers to compromise a system were

1. Lack of input validation on Web applications: SQL injection is one of the most popular forms of such vulnerability whereby hackers gain control of a database by manipulating inputs accepted by web applications.

2. Buffer overflows in C and C++ programs. Because C and C++ do not check if accesses to arrays are in bounds, hackers can write well-crafted strings into unintended areas and hence gain control of the program's execution.

In the next section, we shall discuss how we can use interprocedural analysis to protect programs against such vulnerabilities.

12.2.5 SQL Injection

SQL injection refers to the vulnerability where hackers can manipulate user input to a Web application and gain unintended access to a database. For example, banks want their users to be able to make transactions online, provided they supply their correct password. A common architecture for such a system is to have the user enter strings into a Web form, and then to have those strings form part of a database query written in the SQL language. If systems developers are not careful, the strings provided by the user can alter the meaning of the SQL statement in unexpected ways.

Example 12.10: Suppose a bank offers its customers access to a relation

```
AcctData(name, password, balance)
```

That is, this relation is a table of triples, each consisting of the name of a customer, the password, and the balance of the account. The intent is that customers can see their account balance only if they provide both their name and their correct password. Having a hacker see an account balance is not the worst thing that could occur, but this simple example is typical of more complicated situations where the hacker could execute payments from the account.

The system might implement a balance inquiry as follows:

1. Users invoke a Web form where they enter their name and password.

2. The name is copied to a variable n and the password to a variable p.

3. Later, perhaps in some other procedure, the following SQL query is executed:

```
SELECT balance FROM AcctData
WHERE name = ':n' and password = ':p'
```

For readers not familiar with SQL, this query says: "Find in the table `AcctData` a row with the first component (name) equal to the string currently in variable n and the second component (password) equal to the string currently in variable p; print the third component (balance) of that row." Note that SQL uses single quotes, not double quotes, to delimit strings, and the colons in front of n and p indicate that they are variables of the surrounding language.

Suppose the hacker, who wants to find Charles Dickens' account balance, supplies the following values for the strings n and p:

$n =$ `Charles Dickens' --` $p =$ `who cares`

The effect of these strange strings is to convert the query into

```
SELECT balance FROM AcctData
WHERE name = 'Charles Dickens' --' and password = 'who cares'
```

In many database systems `--` is a comment-introducing token and has the effect of making whatever follows on that line a comment. As a result, the query now asks the database system to print the balance for every person whose name is `'Charles Dickens'`, regardless of the password that appears with that name in a name-password-balance triple. That is, with comments eliminated, the query is:

```
SELECT balance FROM AcctData
WHERE name = 'Charles Dickens'
```

□

In Example 12.10, the "bad" strings were kept in two variables, which might be passed between procedures. However, in more realistic cases, these strings might be copied several times, or combined with others to form the full query. We cannot hope to detect coding errors that create SQL-injection vulnerabilities without doing a full interprocedural analysis of the entire program.

12.2.6 Buffer Overflow

A *buffer overflow attack* occurs when carefully crafted data supplied by the user writes beyond the intended buffer and manipulates the program execution. For example, a C program may read a string s from the user, and then copy it into a buffer b using the function call:

```
strcpy(b,s);
```

If the string s is actually longer than the buffer b, then locations that are not part of b will have their values changed. That in itself will probably cause the program to malfunction or at least to produce the wrong answer, since some data used by the program will have been changed.

But worse, the hacker who chose the string s can pick a value that will do more than cause an error. For example, if the buffer is on the run-time stack, then it is near the return address for its function. An insidiously chosen value of s may overwrite the return address, and when the function returns, it goes to a place chosen by the hacker. If hackers have detailed knowledge of the surrounding operating system and hardware, they may be able to execute a command that will give them control of the machine itself. In some situations, they may even have the ability to have the false return address transfer control to code that is part of the string s, thus allowing any sort of program to be inserted into the executing code.

To prevent buffer overflows, every array-write operation must be statically proven to be within bounds, or a proper array-bounds check must be performed dynamically. Because these bounds checks need to be inserted by hand in C and C++ programs, it is easy to forget to insert the test or to get the test wrong. Heuristic tools have been developed that will check if at least some test, though not necessarily a correct test, has been performed before a `strcpy` is called.

Dynamic bounds checking is unavoidable because it is impossible to determine statically the size of users' input. All a static analysis can do is assure that the dynamic checks have been inserted properly. Thus, a reasonable strategy is to have the compiler insert dynamic bounds checking on every write, and use static analysis as a means to optimize away as many bounds check as possible. It is no longer necessary to catch every potential violation; moreover, we only need to optimize only those code regions that execute frequently.

Inserting bounds checking into C programs is nontrivial, even if we do not mind the cost. A pointer may point into the middle of some array, and we do not know the extent of that array. Techniques have been developed to keep track of the extent of the buffer pointed to by each pointer dynamically. This information allows the compiler to insert array bounds checks for all accesses. Interestingly enough, it is not advisable to halt a program whenever a buffer overflow is detected. In fact, buffer overflows do occur in practice, and a program would likely fail if we disable all buffer overflows. The solution is to extend the size of the array dynamically to accommodate for the buffer overruns.

Interprocedural analysis can be used to speed up the cost of dynamic array bounds checks. For example, suppose we are interested only in catching buffer overflows involving user-input strings, we can use static analysis to determine which variables may hold contents provided by the user. Like SQL injection, being able to track an input as it is copied across procedures is useful in eliminating unnecessary bounds checks.

12.3 A Logical Representation of Data Flow

To this point, our representation of data-flow problems and solutions can be termed "set-theoretic." That is, we represent information as sets and compute results using operators like union and intersection. For instance, when we introduced the reaching-definitions problem in Section 9.2.4, we computed IN$[B]$ and OUT$[B]$ for a block B, and we described these as sets of definitions. We represented the contents of the block B by its gen and kill sets.

To cope with the complexity of interprocedural analysis, we now introduce a more general and succinct notation based on logic. Instead of saying something like "definition D is in IN$[B]$," we shall use a notation like $in(B, D)$ to mean the same thing. Doing so allows us to express succinct "rules" about inferring program facts. It also allows us to implement these rules efficiently, in a way that generalizes the bit-vector approach to set-theoretic operations. Finally, the logical approach allows us to combine what appear to be several independent analyses into one, integrated algorithm. For example, in Section 9.5 we described partial-redundancy elimination by a sequence of four data-flow analyses and two other intermediate steps. In the logical notation, all these steps could be combined into one collection of logical rules that are solved simultaneously.

12.3.1 Introduction to Datalog

Datalog is a language that uses a Prolog-like notation, but whose semantics is far simpler than that of Prolog. To begin, the elements of Datalog are *atoms* of the form $p(X_1, X_2, \ldots, X_n)$. Here,

1. p is a *predicate* — a symbol that represents a type of statement such as "a definition reaches the beginning of a block."

2. $X_1, X_2, \ldots, X_n$ are terms such as variables or constants. We shall also allow simple expressions as arguments of a predicate.[2]

A *ground atom* is a predicate with only constants as arguments. Every ground atom asserts a particular fact, and its value is either true or false. It

[2]Formally, such terms are built from function symbols and complicate the implementation of Datalog considerably. However, we shall use only a few operators, such as addition or subtraction of constants, in contexts that do not complicate matters.

is often convenient to represent a predicate by a *relation*, or table of its true ground atoms. Each ground atom is represented by a single row, or *tuple*, of the relation. The columns of the relation are named by *attributes*, and each tuple has a component for each attribute. The attributes correspond to the components of the ground atoms represented by the relation. Any ground atom in the relation is true, and ground atoms not in the relation are false.

Example 12.11: Let us suppose the predicate $in(B, D)$ means "definition D reaches the beginning of block B." Then we might suppose that, for a particular flow graph, $in(b_1, d_1)$ is true, as are $in(b_2, d_1)$ and $in(b_2, d_2)$. We might also suppose that for this flow graph, all other in facts are false. Then the relation in Fig. 12.12 represents the value of this predicate for this flow graph.

B	D
b_1	d_1
b_2	d_1
b_2	d_2

Figure 12.12: Representing the value of a predicate by a relation

The attributes of the relation are B and D. The three tuples of the relation are (b_1, d_1), (b_2, d_1), and (b_2, d_2). □

We shall also see at times an atom that is really a comparison between variables and constants. An example would be $X \neq Y$ or $X = 10$. In these examples, the predicate is really the comparison operator. That is, we can think of $X = 10$ as if it were written in predicate form: $equals(X, 10)$. There is an important difference between comparison predicates and others, however. A comparison predicate has its standard interpretation, while an ordinary predicate like in means only what it is defined to mean by a Datalog program (described next).

A *literal* is either an atom or a negated atom. We indicate negation with the word `NOT` in front of the atom. Thus, `NOT` $in(B, D)$ is an assertion that definition D does not reach the beginning of block B.

12.3.2 Datalog Rules

Rules are a way of expressing logical inferences. In Datalog, rules also serve to suggest how a computation of the true facts should be carried out. The form of a rule is

$$H \;\texttt{:-}\; B_1 \;\&\; B_2 \;\&\; \cdots \;\&\; B_n$$

The components are as follows:

- H is an atom, and $B_1, B_2, \ldots, B_n$ are literals (atoms, possibly negated).

Datalog Conventions

We shall use the following conventions for Datalog programs:

1. Variables begin with a capital letter.
2. All other elements begin with lowercase letters or other symbols such as digits. These elements include predicates and constants that are arguments of predicates.

- H is the *head* and $B_1, B_2, \ldots, B_n$ form the *body* of the rule.
- Each of the B_i's is sometimes called a *subgoal* of the rule.

We should read the `:-` symbol as "if." The meaning of a rule is "the head is true if the body is true." More precisely, we *apply* a rule to a given set of ground atoms as follows. Consider all possible substitutions of constants for the variables of the rule. If a substitution makes every subgoal of the body true (assuming that all and only the given ground atoms are true), then we can infer that the head with this substitution of constants for variables is a true fact. Substitutions that do not make all subgoals true give us no information; the head may or may not be true.

A *Datalog program* is a collection of rules. This program is applied to "data," that is, to a set of ground atoms for some of the predicates. The result of the program is the set of ground atoms inferred by applying the rules until no more inferences can be made.

Example 12.12 : A simple example of a Datalog program is the computation of paths in a graph, given its (directed) edges. That is, there is one predicate $edge(X, Y)$ that means "there is an edge from node X to node Y." Another predicate $path(X, Y)$ means that there is a path from X to Y. The rules defining paths are:

$$\begin{array}{lll} 1) & path(X,Y) & \texttt{:-} \quad edge(X,Y) \\ 2) & path(X,Y) & \texttt{:-} \quad path(X,Z) \;\&\; path(Z,Y) \end{array}$$

The first rule says that a single edge is a path. That is, whenever we replace variable X by a constant a and variable Y by a constant b, and $edge(a, b)$ is true (i.e., there is an edge from node a to node b), then $path(a, b)$ is also true (i.e., there is a path from a to b). The second rule says that if there is a path from some node X to some node Z, and there is also a path from Z to node Y, then there is a path from X to Y. This rule expresses "transitive closure." Note that any path can be formed by taking the edges along the path and applying the transitive closure rule repeatedly.

For instance, suppose that the following facts (ground atoms) are true: $edge(1,2)$, $edge(2,3)$, and $edge(3,4)$. Then we can use the first rule with three different substitutions to infer $path(1,2)$, $path(2,3)$, and $path(3,4)$. As an example, substituting $X = 1$ and $Y = 2$ instantiates the first rule to be $path(1,2)$:− $edge(1,2)$. Since $edge(1,2)$ is true, we infer $path(1,2)$.

With these three *path* facts, we can use the second rule several times. If we substitute $X = 1$, $Z = 2$, and $Y = 3$, we instantiate the rule to be $path(1,3)$:− $path(1,2)$ & $path(2,3)$. Since both subgoals of the body have been inferred, they are known to be true, so we may infer the head: $path(1,3)$. Then, the substitution $X = 1$, $Z = 3$, and $Y = 4$ lets us infer the head $path(1,4)$; that is, there is a path from node 1 to node 4. □

12.3.3 Intensional and Extensional Predicates

It is conventional in Datalog programs to distinguish predicates as follows:

1. EDB, or *extensional database*, predicates are those that are defined a-priori. That is, their true facts are either given in a relation or table, or they are given by the meaning of the predicate (as would be the case for a comparison predicate, e.g.).

2. IDB, or *intensional database*, predicates are defined only by the rules.

A predicate must be IDB or EDB, and it can be only one of these. As a result, any predicate that appears in the head of one or more rules must be an IDB predicate. Predicates appearing in the body can be either IDB or EDB. For instance, in Example 12.12, *edge* is an EDB predicate and *path* is an IDB predicate. Recall that we were given some *edge* facts, such as $edge(1,2)$, but the *path* facts were inferred by the rules.

When Datalog programs are used to express data-flow algorithms, the EDB predicates are computed from the flow graph itself. IDB predicates are then expressed by rules, and the data-flow problem is solved by inferring all possible IDB facts from the rules and the given EDB facts.

Example 12.13 : Let us consider how reaching definitions might be expressed in Datalog. First, it makes sense to think on a statement level, rather than a block level; that is, the construction of gen and kill sets from a basic block will be integrated with the computation of the reaching definitions themselves. Thus, the block b_1 suggested in Fig. 12.13 is typical. Notice that we identify points within the block numbered $0, 1, \ldots, n$, if n is the number of statements in the block. The ith definition is "at" point i, and there is no definition at point 0.

A point in the program must be represented by a pair (b, n), where b is a block name and n is an integer between 0 and the number of statements in block b. Our formulation requires two EDB predicates:

```
        0
            x = y+z
  b1    1
            *p = u
        2
            x = v
        3
```

Figure 12.13: A basic block with points between statements

1. $\mathit{def}(B, N, X)$ is true if and only if the Nth statement in block B may define variable X. For instance, in Fig. 12.13 $\mathit{def}(b_1, 1, x)$ is true, $\mathit{def}(b_1, 3, x)$ is true, and $\mathit{def}(b_1, 2, Y)$ is true for every possible variable Y that p may point to at that point. For the moment, we shall assume that Y can be any variable of the type that p points to.

2. $\mathit{succ}(B, N, C)$ is true if and only if block C is a successor of block B in the flow graph, and B has N statements. That is, control can flow from the point N of B to the point 0 of C. For instance, suppose that b_2 is a predecessor of block b_1 in Fig. 12.13, and b_2 has 5 statements. Then $\mathit{succ}(b_2, 5, b_1)$ is true.

There is one IDB predicate, $\mathit{rd}(B, N, C, M, X)$. It is intended to be true if and only if the definition of variable X at the Mth statement of block C reaches the point N in block B. The rules defining predicate rd are in Fig. 12.14.

$$\begin{array}{lrcl}
1) & \mathit{rd}(B, N, B, N, X) & \text{:-} & \mathit{def}(B, N, X) \\[1em]
2) & \mathit{rd}(B, N, C, M, X) & \text{:-} & \mathit{rd}(B, N-1, C, M, X) \ \& \\
 & & & \mathit{def}(B, N, Y) \ \& \\
 & & & X \neq Y \\[1em]
3) & \mathit{rd}(B, 0, C, M, X) & \text{:-} & \mathit{rd}(D, N, C, M, X) \ \& \\
 & & & \mathit{succ}(D, N, B)
\end{array}$$

Figure 12.14: Rules for predicate rd

Rule (1) says that if the Nth statement of block B defines X, then that definition of X reaches the Nth point of B (i.e., the point immediately after the statement). This rule corresponds to the concept of "gen" in our earlier, set-theoretic formulation of reaching definitions.

Rule (2) represents the idea that a definition passes through a statement unless it is "killed," and the only way to kill a definition is to redefine its variable with 100% certainty. In detail, rule (2) says that the definition of variable X from the Mth statement of block C reaches the point N of block B if

a) it reaches the previous point $N-1$ of B, and

b) there is at least one variable Y, other than X, that may be defined at the Nth statement of B.

Finally, rule (3) expresses the flow of control in the graph. It says that the definition of X at the Mth statement of block C reaches the point 0 of B if there is some block D with N statements, such that the definition of X reaches the end of D, and B is a successor of D. $\Box$

The EDB predicate *succ* from Example 12.13 clearly can be read off the flow graph. We can obtain *def* from the flow graph as well, if we are conservative and assume a pointer can point anywhere. If we want to limit the range of a pointer to variables of the appropriate type, then we can obtain type information from the symbol table, and use a smaller relation *def*. An option is to make *def* an IDB predicate and define it by rules. These rules will use more primitive EDB predicates, which can themselves be determined from the flow graph and symbol table.

Example 12.14: Suppose we introduce two new EDB predicates:

1. $assign(B, N, X)$ is true whenever the Nth statement of block B has X on the left. Note that X can be a variable or a simple expression with an l-value, like $*p$.

2. $type(X, T)$ is true if the type of X is T. Again, X can be any expression with an l-value, and T can be any expression for a legal type.

Then, we can write rules for *def*, making it an IDB predicate. Figure 12.15 is an expansion of Fig. 12.14, with two of the possible rules for *def*. Rule (4) says that the Nth statement of block B defines X, if X is assigned by the Nth statement. Rule (5) says that X can also be defined by the Nth statement of block B if that statement assigns to $*P$, and X is any of the variables of the type that P points to. Other kinds of assignments would need other rules for *def*.

As an example of how we would make inferences using the rules of Fig. 12.15, let us re-examine the block b_1 of Fig. 12.13. The first statement assigns a value to variable x, so the fact $assign(b_1, 1, x)$ would be in the EDB. The third statement also assigns to x, so $assign(b_1, 3, x)$ is another EDB fact. The second statement assigns indirectly through p, so a third EDB fact is $assign(b_1, 2, *p)$. Rule (4) then allows us to infer $def(b_1, 1, x)$ and $def(b_1, 3, x)$.

Suppose that p is of type pointer-to-integer (*int), and x and y are integers. Then we may use rule (5), with $B = b_1$, $N = 2$, $P = p$, $T = \text{int}$, and X equal to either x or y, to infer $def(b_1, 2, x)$ and $def(b_1, 2, y)$. Similarly, we can infer the same about any other variable whose type is integer or coerceable to an integer. $\Box$

```
1)  rd(B,N,B,N,X)   :-  def(B,N,X)

2)  rd(B,N,C,M,X)   :-  rd(B,N-1,C,M,X) &
                        def(B,N,Y) &
                        X ≠ Y

3)  rd(B,0,C,M,X)   :-  rd(D,N,C,M,X) &
                        succ(D,N,B)

4)      def(B,N,X)  :-  assign(B,N,X)

5)      def(B,N,X)  :-  assign(B,N,*P) &
                        type(X,T) &
                        type(P,*T)
```

Figure 12.15: Rules for predicates *rd* and *def*

12.3.4 Execution of Datalog Programs

Every set of Datalog rules defines relations for its IDB predicates, as a function of the relations that are given for its EDB predicates. Start with the assumption that the IDB relations are empty (i.e., the IDB predicates are false for all possible arguments). Then, repeatedly apply the rules, inferring new facts whenever the rules require us to do so. When the process converges, we are done, and the resulting IDB relations form the output of the program. This process is formalized in the next algorithm, which is similar to the iterative algorithms discussed in Chapter 9.

Algorithm 12.15 : Simple evaluation of Datalog programs.

INPUT: A Datalog program and sets of facts for each EDB predicate.

OUTPUT: Sets of facts for each IDB predicate.

METHOD: For each predicate p in the program, let R_p be the relation of facts that are true for that predicate. If p is an EDB predicate, then R_p is the set of facts given for that predicate. If p is an IDB predicate, we shall compute R_p. Execute the algorithm in Fig. 12.16. □

Example 12.16 : The program in Example 12.12 computes paths in a graph. To apply Algorithm 12.15, we start with EDB predicate *edge* holding all the edges of the graph and with the relation for *path* empty. On the first round, rule (2) yields nothing, since there are no *path* facts. But rule (1) causes all the *edge* facts to become *path* facts as well. That is, after the first round, we know $path(a, b)$ if and only if there is an edge from a to b.

```
for (each IDB predicate $p$)
        $R_p = \emptyset$;
while (changes to any $R_p$ occur) {
        consider all possible substitutions of constants for
            variables in all the rules;
        determine, for each substitution, whether all the
            subgoals of the body are true, using the current
            $R_p$'s to determine truth of EDB and IDB predicates;
        if (a substitution makes the body of a rule true)
                add the head to $R_q$ if $q$ is the head predicate;
}
```

Figure 12.16: Evaluation of Datalog programs

On the second round, rule (1) yields no new paths facts, because the EDB relation *edge* never changes. However, now rule (2) lets us put together two paths of length 1 to make paths of length 2. That is, after the second round, *path*(a, b) is true if and only if there is a path of length 1 or 2 from a to b. Similarly, on the third round, we can combine paths of length 2 or less to discover all paths of length 4 or less. On the fourth round, we discover paths of length up to to 8, and in general, after the ith round, *path*(a, b) is true if and only if there is a path from a to b of length 2^{i-1} or less. □

12.3.5 Incremental Evaluation of Datalog Programs

There is an efficiency enhancement of Algorithm 12.15 possible. Observe that a new IDB fact can only be discovered on round i if it is the result of substituting constants in a rule, such that at least one of the subgoals becomes a fact that was just discovered on round $i-1$. The proof of that claim is that if all the facts among the subgoals were known at round $i-2$, then the "new" fact would have been discovered when we made the same substitution of constants on round $i-1$.

To take advantage of this observation, introduce for each IDB predicate p a predicate *newP* that will hold only the newly discovered p-facts from the previous round. Each rule that has one or more IDB predicates among its subgoals is replaced by a collection of rules. Each rule in the collection is formed by replacing exactly one occurrence of some IDB predicate q in the body by *newQ*. Finally, for all rules, we replace the head predicate h by *newH*. The resulting rules are said to be in *incremental form.*

The relations for each IDB predicate p accumulates all the p-facts, as in Algorithm 12.15. In one round, we

1. Apply the rules to evaluate the *newP* predicates.

Incremental Evaluation of Sets

It is also possible to solve set-theoretic data-flow problems incrementally. For example, in reaching definitions, a definition can only be newly discovered to be in IN$[B]$ on the ith round if it was just discovered to be in OUT$[P]$ for some predecessor P of B. The reason we do not generally try to solve such data-flow problems incrementally is that the bit-vector implementation of sets is so efficient. It is generally easier to fly through the complete vectors than to decide whether a fact is new or not.

2. Then, subtract p from $newP$, to make sure the facts in $newP$ are truly new.

3. Add the facts in $newP$ to p.

4. Set all the $newX$ relations to $\emptyset$ for the next round.

These ideas will be formalized in Algorithm 12.18. However, first, we shall give an example.

Example 12.17: Consider the Datalog program in Example 12.12 again. The incremental form of the rules is given in Fig. 12.17. Rule (1) does not change, except in the head because it has no IDB subgoals in the body. However, rule (2), with two IDB subgoals, becomes two different rules. In each rule, one of the occurrences of *path* in the body is replaced by *newPath*. Together, these rules enforce the idea that at least one of the two paths concatenated by the rule must have been discovered on the previous round. □

```
 1)  newPath(X,Y)  :-  edge(X,Y)

2a)  newPath(X,Y)  :-  path(X,Z) &
                       newPath(Z,Y)

2b)  newPath(X,Y)  :-  newPath(X,Z) &
                       path(Z,Y)
```

Figure 12.17: Incremental rules for the path Datalog program

Algorithm 12.18: Incremental evaluation of Datalog programs.

INPUT: A Datalog program and sets of facts for each EDB predicate.

OUTPUT: Sets of facts for each IDB predicate.

METHOD: For each predicate p in the program, let R_p be the relation of facts that are true for that predicate. If p is an EDB predicate, then R_p is the set of facts given for that predicate. If p is an IDB predicate, we shall compute R_p. In addition, for each IDB predicate p, let R_{newP} be a relation of "new" facts for predicate p.

1. Modify the rules into the incremental form described above.
2. Execute the algorithm in Fig. 12.18.

□

```
for (each IDB predicate $p$) {
        $R_p = \emptyset$;
        $R_{newP} = \emptyset$;
}
repeat {
        consider all possible substitutions of constants for
            variables in all the rules;
        determine, for each substitution, whether all the
            subgoals of the body are true, using the current
            $R_p$'s and $R_{newP}$'s to determine truth of EDB
            and IDB predicates;
        if (a substitution makes the body of a rule true)
                add the head to $R_{newH}$, where $h$ is the head
                    predicate;
        for (each predicate $p$) {
                $R_{newP} = R_{newP} - R_p$;
                $R_p = R_p \cup R_{newP}$;
        }
} until (all $R_{newP}$'s are empty);
```

Figure 12.18: Evaluation of Datalog programs

12.3.6 Problematic Datalog Rules

There are certain Datalog rules or programs that technically have no meaning and should not be used. The two most important risks are

1. *Unsafe rules*: those that have a variable in the head that does not appear in the body in a way that constrains that variable to take on only values that appear in the EDB.
2. *Unstratified programs*: sets of rules that have a recursion involving a negation.

We shall elaborate on each of these risks.

Rule Safety

Any variable that appears in the head of a rule must also appear in the body. Moreover, that appearance must be in a subgoal that is an ordinary IDB or EDB atom. It is not acceptable if the variable appears only in a negated atom, or only in a comparison operator. The reason for this policy is to avoid rules that let us infer an infinite number of facts.

Example 12.19 : The rule

$$p(X,Y) \text{ :- } q(Z) \text{ \& } \texttt{NOT}\ r(X) \text{ \& } X \neq Y$$

is unsafe for two reasons. Variable X appears only in the negated subgoal $r(X)$ and the comparison $X \neq Y$. Y appears only in the comparison. The consequence is that p is true for an infinite number of pairs (X, Y), as long as $r(X)$ is false and Y is anything other than X. □

Stratified Datalog

In order for a program to make sense, recursion and negation must be separated. The formal requirement is as follows. We must be able to divide the IDB predicates into *strata*, so that if there is a rule with head predicate p and a subgoal of the form NOT $q(\cdots)$, then q is either EDB or an IDB predicate in a lower stratum than p. As long as this rule is satisfied, we can evaluate the strata, lowest first, by Algorithm 12.15 or 12.18, and then treat the relations for the IDB predicates of that strata as if they were EDB for the computation of higher strata. However, if we violate this rule, then the iterative algorithm may fail to converge, as the next example shows.

Example 12.20 : Consider the Datalog program consisting of the one rule:

$$p(X) \text{ :- } e(X) \text{ \& } \texttt{NOT}\ p(X)$$

Suppose e is an EDB predicate, and only $e(1)$ is true. Is $p(1)$ true?

This program is not stratified. Whatever stratum we put p in, its rule has a subgoal that is negated and has an IDB predicate (namely p itself) that is surely not in a lower stratum than p.

If we apply the iterative algorithm, we start with $R_p = \emptyset$, so initially, the answer is "no; $p(1)$ is not true." However, the first iteration lets us infer $p(1)$, since both $e(1)$ and NOT $p(1)$ are true. But then the second iteration tells us $p(1)$ is false. That is, substituting 1 for X in the rule does not allow us to infer $p(1)$, since subgoal NOT $p(1)$ is false. Similarly, the third iteration says $p(1)$ is true, the fourth says it is false, and so on. We conclude that this unstratified program is meaningless, and do not consider it a valid program. □

12.3.7 Exercises for Section 12.3

! **Exercise 12.3.1:** In this problem, we shall consider a reaching-definitions data-flow analysis that is simpler than that in Example 12.13. Assume that each statement by itself is a block, and initially assume that each statement defines exactly one variable. The EDB predicate $pred(I, J)$ means that statement I is a predecessor of statement J. The EDB predicate $defines(I, X)$ means that the variable defined by statement I is X. We shall use IDB predicates $in(I, D)$ and $out(I, D)$ to mean that definition D reaches the beginning or end of statement I, respectively. Note that a definition is really a statement number. Fig. 12.19 is a Datalog program that expresses the usual algorithm for computing reaching definitions.

```
1)  kill(I,D)  :-  defines(I,X) & defines(D,X)

2)   out(I,I)  :-  defines(I,X)
3)   out(I,D)  :-  in(I,D) & NOT kill(I,D)

4)    in(I,D)  :-  out(J,D) & pred(J,I)
```

Figure 12.19: Datalog program for a simple reaching-definitions analysis

Notice that rule (1) says that a statement kills itself, but rule (2) assures that a statement is in its own "out set" anyway. Rule (3) is the normal transfer function, and rule (4) allows confluence, since I can have several predecessors.

Your problem is to modify the rules to handle the common case where a definition is ambiguous, e.g., an assignment through a pointer. In this situation, $defines(I, X)$ may be true for several different X's and one I. A definition is best represented by a pair (D, X), where D is a statement, and X is one of the variables that may be defined at D. As a result, *in* and *out* become three-argument predicates; e.g., $in(I, D, X)$ means that the (possible) definition of X at statement D reaches the beginning of statement I.

Exercise 12.3.2: Write a Datalog program analogous to Fig. 12.19 to compute available expressions. In addition to predicate *defines*, use a predicate $eval(I, X, O, Y)$ that says statement I causes expression XOY to be evaluated. Here, O is the operator in the expression, e.g., $+$.

Exercise 12.3.3: Write a Datalog program analogous to Fig. 12.19 to compute live variables. In addition to predicate *defines*, assume a predicate $use(I, X)$ that says statement I uses variable X.

Exercise 12.3.4: In Section 9.5, we defined a data-flow calculation that involved six concepts: anticipated, available, earliest, postponable, latest, and used. Suppose we had written a Datalog program to define each of these in terms of EDB concepts derivable from the program (e.g., gen and kill information) and others of these six concepts. Which of the six depend on which others? Which of these dependences are negated? Would the resulting Datalog program be stratified?

Exercise 12.3.5: Suppose that the EDB predicate $edge(X, Y)$ consists of the following facts:

$$\begin{array}{lll} edge(1,2) & edge(2,3) & edge(3,4) \\ edge(4,1) & edge(4,5) & edge(5,6) \end{array}$$

a) Simulate the Datalog program of Example 12.12 on this data, using the simple evaluation strategy of Algorithm 12.15. Show the *path* facts discovered at each round.

b) Simulate the Datalog program of Fig. 12.17 on this data, as part of the incremental evaluation strategy of Algorithm 12.18. Show the *path* facts discovered at each round.

Exercise 12.3.6: The following rule

$$p(X,Y) \text{ :- } q(X,Z) \;\&\; r(Z,W) \;\&\; \texttt{NOT}\; p(W,Y)$$

is part of a larger Datalog program P.

a) Identify the head, body, and subgoals of this rule.

b) Which predicates are certainly IDB predicates of program P?

! c) Which predicates are certainly EDB predicates of P?

d) Is the rule safe?

e) Is P stratified?

Exercise 12.3.7: Convert the rules of Fig. 12.14 to incremental form.

12.4 A Simple Pointer-Analysis Algorithm

In this section, we begin the discussion of a very simple flow-insensitive pointer-alias analysis assuming that there are no procedure calls. We shall show in subsequent sections how to handle procedures first context insensitively, then context sensitively. Flow sensitivity adds a lot of complexity, and is less important to context sensitivity for languages like Java where methods tend to be small.

The fundamental question that we wish to ask in pointer-alias analysis is whether a given pair of pointers may be aliased. One way to answer this question is to compute for each pointer the answer to the question "what objects can this pointer point to?" If two pointers can point to the same object, then the pointers may be aliased.

12.4.1 Why is Pointer Analysis Difficult

Pointer-alias analysis for C programs is particularly difficult, because C programs can perform arbitrary computations on pointers. In fact, one can read in an integer and assign it to a pointer, which would render this pointer a potential alias of all other pointer variables in the program. Pointers in Java, known as references, are much simpler. No arithmetic is allowed, and pointers can only point to the beginning of an object.

Pointer-alias analysis must be interprocedural. Without interprocedural analysis, one must assume that any method called can change the contents of all accessible pointer variables, thus rendering any intraprocedural pointer-alias analysis ineffective.

Languages allowing indirect function calls present an additional challenge for pointer-alias analysis. In C, one can call a function indirectly by calling a dereferenced function pointer. We need to know what the function pointer can point to before we can analyze the function called. And clearly, after analyzing the function called, one may discover more functions that the function pointer can point to, and therefore the process needs to be iterated.

While most functions are called directly in C, virtual methods in Java cause many invocations to be indirect. Given an invocation `x.m()` in a Java program, there may be many classes to which object x might belong and that have a method named m. The more precise our knowledge of the actual type of x, the more precise our call graph is. Ideally, we can determine at compile time the exact class of x and thus know exactly which method m refers to.

Example 12.21: Consider the following sequence of Java statements:

```
Object o;
o = new String();
n = o.hashCode();
```

Here o is declared to be an `Object`. Without analyzing what o refers to, all methods called "hashCode" declared for all classes must be considered as possible targets. Knowing that o points to a `String` will narrow interprocedural analysis to precisely the method declared for `String`. □

It is possible to apply approximations to reduce the number of targets. For example, statically we can determine what are all the types of objects created, and we can limit the analysis to those. But we can be more accurate if we can discover the call graph on the fly, based on the points-to analysis obtained at the same time. More accurate call graphs lead not only to more precise results but also can reduce greatly the analysis time otherwise needed.

Points-to analysis is complicated. It is not one of those "easy" data flow problems where we only need to simulate the effect of going around a loop of statements once. Rather, as we discover new targets for a pointer, all statements assigning the contents of that pointer to another pointer need to be re-analyzed.

For simplicity, we shall focus mainly on Java. We shall start with flow-insensitive and context-insensitive analysis, assuming for now that no methods are called in the program. Then, we describe how we can discover the call graph on the fly as the points-to results are computed. Finally, we describe one way of handling context sensitivity.

12.4.2 A Model for Pointers and References

Let us suppose that our language has the following ways to represent and manipulate references:

1. Certain program variables are of type "pointer to T" or "reference to T," where T is a type. These variables are either static or live on the run-time stack. We call them simply *variables*.

2. There is a heap of objects. All variables point to heap objects, not to other variables. These objects will be referred to as *heap objects*.

3. A heap object can have *fields*, and the value of a field can be a reference to a heap object (but not to a variable).

Java is modeled well by this structure, and we shall use Java syntax in examples. Note that C is modeled less well, since pointer variables can point to other pointer variables in C, and in principle, any C value can be coerced into a pointer.

Since we are performing an insensitive analysis, we only need to assert that a given variable v can point to a given heap object h; we do not have to address the issue of where in the program v can point to h, or in what contexts v can point to h. Note, however, that variables can be named by their full name. In Java, this name can incorporate the module, class, method, and block within a method, as well as the variable name itself. Thus, we can distinguish many variables that have the same identifier.

Heap objects do not have names. Approximation often is used to name the objects, because an unbounded number of objects may be created dynamically. One convention is to refer to objects by the statement at which they are created. As a statement can be executed many times and create a new object each time, an assertion like "v can point to h" really means "v can point to one or more of the objects created at the statement labeled h."

The goal of the analysis is to determine what each variable and each field of each heap object can point to. We refer to this as a *points-to analysis*; two pointers are aliased if their points-to sets intersect. We describe here an *inclusion-based* analysis; that is, a statement such as `v = w` causes variable v to point to all the objects w points to, but not vice versa. While this approach may seem obvious, there are other alternatives to how we define points-to analysis. For example, we can define an *equivalence-based* analysis such that a statement like `v = w` would turn variables v and w into one equivalence class, pointing to all the variables that each can point to. While this formulation does not approximate aliases well, it provides a quick, and often good, answer to the question of which variables point to the same kind of objects.

12.4.3 Flow Insensitivity

We start by showing a very simple example to illustrate the effect of ignoring control flow in points-to analysis.

Example 12.22: In Fig. 12.20, three objects, h, i, and j, are created and assigned to variables a, b, and c, respectively. Thus, surely a points to h, b points to i, and c points to j by the end of line (3).

```
1)  h:  a = new Object();
2)  i:  b = new Object();
3)  j:  c = new Object();
4)      a = b;
5)      b = c;
6)      c = a;
```

Figure 12.20: Java code for Example 12.22

If you follow the statements (4) through (6), you discover that after line (4) a points only to i. After line (5), b points only to j, and after line (6), c points only to i. □

The above analysis is flow sensitive because we follow the control flow and compute what each variable can point to after each statement. In other words, in addition to considering what points-to information each statement "generates," we also account for what points-to information each statement "kills." For instance, the statement `b = c;` kills the previous fact "b points to i" and generates the new relationship "b points to what c points to."

A flow-insensitive analysis ignores the control flow, which essentially assumes that every statement in the program can be executed in any order. It computes only one global points-to map indicating what each variable can possibly point to at any point of the program execution. If a variable can point to two different objects after two different statements in a program, we simply record that it can point to both objects. In other words, in flow-insensitive analysis, an assignment does not "kill" any points-to relations but can only "generate" more points-to relations. To compute the flow-insensitive results, we repeatedly add the points-to effects of each statement on the points-to relationships until no new relations are found. Clearly, lack of flow sensitivity weakens the analysis results greatly, but it tends to reduce the size of the representation of the results and make the algorithm converge faster.

Example 12.23: Returning to Example 12.22, lines (1) through (3) again tell us a can point to h; b can point to i, and c can point to j. With lines (4) and (5), a can point to both h and i, and b can point to both i and j. With line (6), c can point to h, i, and j. This information affects line (5), which in turn affects line (4), In the end, we are left with the useless conclusion that anything can point to anything. □

12.4.4 The Formulation in Datalog

Let us now formalize a flow-insensitive pointer-alias analysis based on the discussion above. We shall ignore procedure calls for now and concentrate on the four kinds of statements that can affect pointers:

1. *Object creation.* `h: T v = new T();` This statement creates a new heap object, and variable v can point to it.

2. *Copy statement.* `v = w;` Here, v and w are variables. The statement makes v point to whatever heap object w currently points to; i.e., w is copied into v.

3. *Field store.* `v.f = w;` The type of object that v points to must have a field f, and this field must be of some reference type. Let v point to heap object h, and let w point to g. This statement makes the field f, in h now point to g. Note that the variable v is unchanged.

4. *Field load.* `v = w.f;` Here, w is a variable pointing to some heap object that has a field f, and f points to some heap object h. The statement makes variable v point to h.

Note that compound field accesses in the source code such as `v = w.f.g` will be broken down into two primitive field-load statements:

```
v1 = w.f;
v  = v1.g;
```

Let us now express the analysis formally in Datalog rules. First, there are only two IDB predicates we need to compute:

1. $pts(V, H)$ means that variable V can point to heap object H.

2. $hpts(H, F, G)$ means that field F of heap object H can point to heap object G.

The EDB relations are constructed from the program itself. Since the location of statements in a program is irrelevant when the analysis is flow-insensitive, we only have to assert in the EDB the existence of statements that have certain forms. In what follows, we shall make a convenient simplification. Instead of defining EDB relations to hold the information garnered from the program, we shall use a quoted statement form to suggest the EDB relation or relations that represent the existence of such a statement. For example, "$H : T\ V\ = \text{new}\ T$" is an EDB fact asserting that at statement H there is an assignment that makes variable V point to a new object of type T. We assume that in practice, there would be a corresponding EDB relation that would be populated with ground atoms, one for each statement of this form in the program.

With this convention, all we need to write the Datalog program is one rule for each of the four types of statements. The program is shown in Fig. 12.21. Rule (1) says that variable V can point to heap object H if statement H is an assignment of a new object to V. Rule (2) says that if there is a copy statement `V = W`, and W can point to H, then V can point to H.

$$
\begin{array}{lrcl}
1) & pts(V,H) & \texttt{:-} & \text{“}H : T\ V\ = \texttt{new}\ T\text{”} \\
 & & & \\
2) & pts(V,H) & \texttt{:-} & \text{“}V = W\text{”}\ \& \\
 & & & pts(W,H) \\
 & & & \\
3) & hpts(H,F,G) & \texttt{:-} & \text{“}V.F = W\text{”}\ \& \\
 & & & pts(W,G)\ \& \\
 & & & pts(V,H) \\
 & & & \\
4) & pts(V,H) & \texttt{:-} & \text{“}V = W.F\text{”}\ \& \\
 & & & pts(W,G)\ \& \\
 & & & hpts(G,F,H)
\end{array}
$$

Figure 12.21: Datalog program for flow-insensitive pointer analysis

Rule (3) says that if there is a statement of the form `V.F = W`, W can point to G, and V can point to H, then the F field of H can point to G. Finally, rule (4) says that if there is a statement of the form `V = W.F`, W can point to G, and the F field of G can point to H, then V can point to H. Notice that *pts* and *hpts* are mutually recursive, but this Datalog program can be evaluated by either of the iterative algorithms discussed in Section 12.3.4.

12.4.5 Using Type Information

Because Java is type safe, variables can only point to types that are compatible to the declared types. For example, assigning an object belonging to a superclass of the declared type of a variable would raise a run-time exception. Consider the simple example in Fig. 12.22, where S is a subclass of T. This program will generate a run-time exception if p is true, because a cannot be assigned an object of class T. Thus, statically we can conclude that because of the type restriction, a can only point to h and not g.

```
      S a;
      T b;
      if (p) {
g:       b = new T();
      } else
h:       b = new S();
      }
      a = b;
```

Figure 12.22: Java program with a type error

Thus, we introduce to our analysis three EDB predicates that reflect important type information in the code being analyzed. We shall use the following:

1. *vType*(V, T) says that variable V is declared to have type T.

2. *hType*(H, T) says that heap object H is allocated with type T. The type of a created object may not be known precisely if, for example, the object is returned by a native method. Such types are modeled conservatively as all possible types.

3. *assignable*(T, S) means that an object of type S can be assigned to a variable with the type T. This information is generally gathered from the declaration of subtypes in the program, but also incorporates information about the predefined classes of the language. *assignable*(T, T) is always true.

We can modify the rules from Fig. 12.21 to allow inferences only if the variable assigned gets a heap object of an assignable type. The rules are shown in Fig. 12.23.

The first modification is to rule (2). The last three subgoals say that we can only conclude that V can point to H if there are types T and S that variable V and heap object H may respectively have, such that objects of type S can be assigned to variables that are references to type T. A similar additional restriction has been added to rule (4). Notice that there is no additional restriction in rule (3) because all stores must go through a variable whose type already has been checked. Any type restriction would only catch one extra case, when the base object is a null constant.

12.4.6 Exercises for Section 12.4

Exercise 12.4.1: In Fig. 12.24, h and g are labels used to represent newly created objects, and are not part of the code. You may assume that objects of type T have a field f. Use the Datalog rules of this section to infer all possible *pts* and *hpts* facts.

1)	$pts(V, H)$	:-	"$H :\ T\ V\ = \texttt{new}\ T$"
2)	$pts(V, H)$	:-	"$V = W$" &
			$pts(W, H)$ &
			$vType(V, T)$ &
			$hType(H, S)$ &
			$assignable(T, S)$
3)	$hpts(H, F, G)$	:-	"$V.F = W$" &
			$pts(W, G)$ &
			$pts(V, H)$
4)	$pts(V, H)$	:-	"$V = W.F$" &
			$pts(W, G)$ &
			$hpts(G, F, H)$ &
			$vType(V, T)$ &
			$hType(H, S)$ &
			$assignable(T, S)$

Figure 12.23: Adding type restrictions to flow-insensitive pointer analysis

```
h: T a = new T();
g: T b = new T();
   T c = a;
   a.f = b;
   b.f = c;
   T d = c.f;
```

Figure 12.24: Code for Exercise 12.4.1

! Exercise 12.4.2: Applying the algorithm of this section to the code

```
g: T a = new T();
h:   a = new T();
   T c = a;
```

would infer that both a and b can point to g and h. Had the code been written

```
g: T a = new T();
h: T b = new T();
   T c = b;
```

we would infer accurately that a can point to g, and b and c can point to h. Suggest an intraprocedural data-flow analysis that can avoid this kind of inaccuracy.

```
t p(t x) {
    h: T a = new T;
       a.f = x;
       return a;
}

void main() {
    g: T b = new T;
         b = p(b);
         b = b.f;
}
```

Figure 12.25: Example code for pointer analysis

! **Exercise 12.4.3:** We can extend the analysis of this section to be interprocedural if we simulate call and return as if they were copy operations, as in rule (2) of Fig. 12.21. That is, a call copies the actuals to their corresponding formals, and the return copies the variable that holds the return value to the variable that is assigned the result of the call. Consider the program of Fig. 12.25.

a) Perform an insensitive analysis on this code.

b) Some of the inferences made in (a) are actually "bogus," in the sense that they do not represent any event that can occur at run-time. The problem can be traced to the multiple assignments to variable b. Rewrite the code of Fig. 12.25 so that no variable is assigned more than once. Rerun the analysis and show that each inferred *pts* and *hpts* fact can occur at run time.

12.5 Context-Insensitive Interprocedural Analysis

We now consider method invocations. We first explain how points-to analysis can be used to compute a precise call graph, which is useful in computing precise points-to results. We then formalize on-the-fly call-graph discovery and show how Datalog can be used to describe the analysis succinctly.

12.5.1 Effects of a Method Invocation

The effects of a method call such as `x = y.n(z)` in Java on the points-to relations can be computed as follows:

1. Determine the type of the receiver object, which is the object that y points to. Suppose its type is t. Let m be the method named n in the narrowest

superclass of t that has a method named n. Note that, in general, which method is invoked can only be determined dynamically.

2. The formal parameters of m are assigned the objects pointed to by the actual parameters. The actual parameters include not just the parameters passed in directly, but also the receiver object itself. Every method invocation assigns the receiver object to the `this` variable.[3] We refer to the `this` variables as the 0th formal parameters of methods. In `x = y.n(z)`, there are two formal parameters: the object pointed to by y is assigned to variable `this`, and the object pointed to by z is assigned to the first declared formal parameter of m.

3. The returned object of m is assigned to the left-hand-side variable of the assignment statement.

In context-insensitive analysis, parameters and returned values are modeled by copy statements. The interesting question that remains is how to determine the type of the receiver object. We can conservatively determine the type according to the declaration of the variable; for example, if the declared variable has type t, then only methods named n in subtypes of t can be invoked. Unfortunately, if the declared variable has type `Object`, then all methods with name n are all potential targets. In real-life programs that use object hierarchies extensively and include many large libraries, such an approach can result in many spurious call targets, making the analysis both slow and imprecise.

We need to know what the variables can point to in order to compute the call targets; but unless we know the call targets, we cannot find out what all the variables can point to. This recursive relationship requires that we discover the call targets on the fly as we compute the points-to set. The analysis continues until no new call targets and no new points-to relations are found.

Example 12.24: In the code in Fig. 12.26, r is a subtype of s, which itself is a subtype of t. Using only the declared type information, `a.n()` may invoke any of the three declared methods with name n since s and r are both subtypes of a's declared type, t. Furthermore, it appears that a may point to objects g, h, and i after line (5).

By analyzing the points-to relationships, we first determine that a can point to j, an object of type t. Thus, the method declared in line (1) is a call target. Analyzing line (1), we determine that a also can point to g, an object of type r. Thus, the method declared in line (3) may also be a call target, and a can now also point to i, another object of type r. Since there are no more new call targets, the analysis terminates without analyzing the method declared in line (2) and without concluding that a can point to h. □

[3] Remember that variables are distinguished by the method to which they belong, so there is not just one variable named `this`, but rather one such variable for each method in the program.

```
        class t {
1) g:      t n() { return new r(); }
        }
        class s extends t {
2) h:      t n() { return new s(); }
        }
        class r extends s {
3) i:      t n() { return new r(); }
        }

        main () {
4) j:      t a = new t();
5)           a = a.n();
        }
```

Figure 12.26: A virtual method invocation

12.5.2 Call Graph Discovery in Datalog

To formulate the Datalog rules for context-insensitive interprocedural analysis, we introduce three EDB predicates, each of which is obtainable easily from the source code:

1. $actual(S, I, V)$ says V is the Ith actual parameter used in call site S.
2. $formal(M, I, V)$ says that V is Ith formal parameter declared in method M.
3. $cha(T, N, M)$ says that M is the method called when N is invoked on a receiver object of type T. (*cha* stands for class hierarchy analysis).

Each edge of the call graph is represented by an IDB predicate *invokes*. As we discover more call-graph edges, more points-to relations are created as the parameters are passed in and returned values are passed out. This effect is summarized by the rules shown in Figure 12.27.

The first rule computes the call target of the call site. That is, "$S : V.N(...)$" says that there is a call site labeled S that invokes method named N on the receiver object pointed to by V. The subgoals say that if V can point to heap object H, which is allocated as type T, and M is the method used when N is invoked on objects of type T, then call site S may invoke method M.

The second rule says that if site S can call method M, then each formal parameter of M can point to whatever the corresponding actual parameter of the call can point to. The rule for handling returned values is left as an exercise.

Combining these two rules with those explained in Section 12.4 create a context-insensitive points-to analysis that uses a call graph that is computed on the fly. This analysis has the side effect of creating a call graph using a

$$\begin{array}{lrcl}
1) & invokes(S,M) & \text{:-} & \text{“}S:\ V.N(...)\text{”}\ \texttt{\&} \\
 & & & pts(V,H)\ \texttt{\&} \\
 & & & hType(H,T)\ \texttt{\&} \\
 & & & cha(T,N,M) \\
 & & & \\
2) & pts(V,H) & \text{:-} & invokes(S,M)\ \texttt{\&} \\
 & & & formal(M,I,V)\ \texttt{\&} \\
 & & & actual(S,I,W)\ \texttt{\&} \\
 & & & pts(W,H)
\end{array}$$

Figure 12.27: Datalog program for call-graph discovery

context-insensitive and flow-insensitive points-to analysis. This call graph is significantly more accurate than one computed based only on type declarations and syntactic analysis.

12.5.3 Dynamic Loading and Reflection

Languages like Java allow dynamic loading of classes. It is impossible to analyze all the possible code executed by a program, and hence impossible to provide any conservative approximation of call graphs or pointer aliases statically. Static analysis can only provide an approximation based on the code analyzed. Remember that all the analyses described here can be applied at the Java bytecode level, and thus it is not necessary to examine the source code. This option is especially significant because Java programs tend to use many libraries.

Even if we assume that all the code to be executed is analyzed, there is one more complication that makes conservative analysis impossible: reflection. Reflection allows a program to determine dynamically the types of objects to be created, the names of methods invoked, as well as the names of the fields accessed. The type, method, and field names can be computed or derived from user input, so in general the only possible approximation is to assume the universe.

Example 12.25 : The code below shows a common use of reflection:

```
String className = ...;
Class c = Class.forName(className);
Object o = c.newInstance();
T t = (T) o;
```

The `forName` method in the `Class` library takes a string containing the class name and returns the class. The method `newInstance` returns an instance of that class. Instead of leaving the object o with type `Object`, this object is cast to a superclass T of all the expected classes. □

While many large Java applications use reflection, they tend to use common idioms, such as the one shown in Example 12.25. As long as the application does not redefine the class loader, we can tell the class of the object if we know the value of `className`. If the value of `className` is defined in the program, because strings are immutable in Java, knowing what `className` points to will provide the name of the class. This technique is another use of points-to analysis. If the value of `className` is based on user input, then the points-to analysis can help locate where the value is entered, and the developer may be able to limit the scope of its value.

Similarly, we can exploit the typecast statement, line (4) in Example 12.25, to approximate the type of dynamically created objects. Assuming that the typecast exception handler has not been redefined, the object must belong to a subclass of the class T.

12.5.4 Exercises for Section 12.5

Exercise 12.5.1: For the code of Fig. 12.26

a) Construct the EDB relations *actual*, *formal*, and *cha*.

b) Make all possible inferences of *pts* and *hpts* facts.

! **Exercise 12.5.2:** How would you add to the EDB predicates and rules of Section 12.5.2 additional predicates and rules to take into account the fact that if a method call returns an object, then the variable to which the result of the call is assigned can point to whatever the variable holding the return value can point to?

12.6 Context-Sensitive Pointer Analysis

As discussed in Section 12.1.2, context sensitivity can improve greatly the precision of interprocedural analysis. We talked about two approaches to interprocedural analysis, one based on cloning (Section 12.1.4) and one on summaries (Section 12.1.5). Which one should we use?

There are several difficulties in computing the summaries of points-to information. First, the summaries are large. Each method's summary must include the effect of all the updates that the function and all its callees can make, in terms of the incoming parameters. That is, a method can change the points-to sets of all data reachable through static variables, incoming parameters and all objects created by the method and its callees. While complicated schemes have been proposed, there is no known solution that can scale to large programs. Even if the summaries can be computed in a bottom-up pass, computing the points-to sets for all the exponentially many contexts in a typical top-down pass presents an even greater problem. Such information is necessary for global queries like finding all points in the code that touch a certain object.

In this section, we discuss a cloning-based context-sensitive analysis. A cloning-based analysis simply clones the methods, one for each context of interest. We then apply the context-insensitive analysis to the cloned call graph. While this approach seems simple, the devil is in the details of handling the large number of clones. How many contexts are there? Even if we use the idea of collapsing all recursive cycles, as discussed in Section 12.1.3, it is not uncommon to find 10^{14} contexts in a Java application. Representing the results of these many contexts is the challenge.

We separate the discussion of context sensitivity into two parts:

1. How to handle context sensitivity logically? This part is easy, because we simply apply the context-insensitive algorithm to the cloned call graph.

2. How to represent the exponentially many contexts? One way is to represent the information as binary decision diagrams (BDD's), a highly-optimized data structure that has been used for many other applications.

This approach to context sensitivity is an excellent example of the importance of abstraction. As we are going to show, we eliminate algorithmic complexity by leveraging the years of work that went into the BDD abstraction. We can specify a context-sensitive points-to analysis in just a few lines of Datalog, which in turn takes advantage of many thousands of lines of existing code for BDD manipulation. This approach has several important advantages. First, it makes possible the easy expression of further analyses that use the results of the points-to analysis. After all, the points-to results on their own are not interesting. Second, it makes it much easier to write the analysis correctly, as it leverages many lines of well-debugged code.

12.6.1 Contexts and Call Strings

The context-sensitive points-to analysis described below assumes that a call graph has been already computed. This step helps make possible a compact representation of the many calling contexts. To get the call graph, we first run a context-insensitive points-to analysis that computes the call graph on the fly, as discussed in Section 12.5. We now describe how to create a cloned call graph.

A context is a representation of the call string that forms the history of the active function calls. Another way to look at the context is that it is a summary of the sequence of calls whose activation records are currently on the run-time stack. If there are no recursive functions on the stack, then the call string — the sequence of locations from which the calls on the stack were made — is a complete representation. It is also an acceptable representation, in the sense that there is only a finite number of different contexts, although that number may be exponential in the number of functions in the program.

However, if there are recursive functions in the program, then the number of possible call strings is infinite, and we cannot allow all possible call strings to represent distinct contexts. There are various ways we can limit the number of distinct contexts. For example, we can write a regular expression that describes all possible call strings and convert that regular expression to a deterministic finite automaton, using the methods of Section 3.7. The contexts can then be identified with the states of this automaton.

Here, we shall adopt a simpler scheme that captures the history of nonrecursive calls but considers recursive calls to be "too hard to unravel." We begin by finding all the mutually recursive sets of functions in the program. The process is simple and will not be elaborated in detail here. Think of a graph whose nodes are the functions, with an edge from p to q if function p calls function q. The strongly connected components (SCC's) of this graph are the sets of mutually recursive functions. As a common special case, a function p that calls itself, but is not in an SCC with any other function is an SCC by itself. The nonrecursive functions are also SCC's by themselves. Call an SCC *nontrivial* if it either has more than one member (the mutually recursive case), or it has a single, recursive member. The SCC's that are single, nonrecursive functions are *trivial* SCC's.

Our modification of the rule that any call string is a context is as follows. Given a call string, delete the occurrence of a call site s if

1. s is in a function p.

2. Function q is called at site s ($q = p$ is possible).

3. p and q are in the same strong component (i.e., p and q are mutually recursive, or $p = q$ and p is recursive).

The result is that when a member of a nontrivial SCC S is called, the call site for that call becomes part of the context, but calls within S to other functions in the same SCC are not part of the context. Finally, when a call outside S is made, we record that call site as part of the context.

Example 12.26 : In Fig. 12.28 is a sketch of five methods with some call sites and calls among them. An examination of the calls shows that q and r are mutually recursive. However, p, s, and t are not recursive at all. Thus, our contexts will be lists of all the call sites except `s3` and `s5`, where the recursive calls between q and r take place.

Let us consider all the ways we could get from p to t, that is, all the contexts in which calls to t occur:

1. p could call s at `s2`, and then s could call t at either `s7` or `s8`. Thus, two possible call strings are (`s2`, `s7`) and (`s2`, `s8`).

2. p could call q at `s1`. Then, q and r could call each other recursively some number of times. We could break the cycle:

   (a) At `s4`, where t is called directly by q. This choice leads to only one context, (`s1`, `s4`).

```
 void p() {
      h: T a = new T();
     s1: T b = a.q();
     s2:       b.s();
  }

  T   q() {
     s3: T c = this.r();
      i: T d = new T();
     s4:       d.t();
         return d;
  }

  T   r() {
     s5: T e = this.q();
     s6:       e.s();
         return e;
  }

  void s() {
     s7: T f = this.t();
     s8:   f = f.t();
  }

  T   t() {
      j: T g = new T();
         return g;
  }
```

Figure 12.28: Methods and call sites for a running example

(b) At s6, where r calls s. Here, we can reach t either by the call at s7 or the call at s8. That gives us two more contexts, $(\texttt{s1}, \texttt{s6}, \texttt{s7})$ and $(\texttt{s1}, \texttt{s6}, \texttt{s8})$.

There are thus five different contexts in which t can be called. Notice that all these contexts omit the recursive call sites, s3 and s5. For example, the context $(\texttt{s1}, \texttt{s4})$ actually represents the infinite set of call strings $(\texttt{s1}, \texttt{s3}, (\texttt{s5}, \texttt{s3})^n, \texttt{s4})$ for all $n \geq 0$. □

We now describe how we derive the cloned call graph. Each cloned method is identified by the method M in the program and a context C. Edges can be derived by adding the corresponding contexts to each of the edges in the original call graph. Recall that there is an edge in the original call graph linking call site S with method M if the predicate $invokes(S, M)$ is true. To add contexts to identify the methods in the cloned call graph, we can define a corresponding *CSinvokes* predicate such that $CSinvokes(S, C, M, D)$ is true if the call site S in context C calls the D context of method M.

12.6.2 Adding Context to Datalog Rules

To find context-sensitive points-to relations, we can simply apply the same context-insensitive points-to analysis to the cloned call graph. Since a method in the cloned call graph is represented by the original method and its context, we revise all the Datalog rules accordingly. For simplicity, the rules below do not include the type restriction, and the _'s are any new variables.

$$\begin{array}{llll}
1) & pts(V,C,H) & \texttt{:-} & \text{“}H:\ T\ V\ = \texttt{new}\ T()\text{”}\ \texttt{\&} \\
 & & & CSinvokes(H,C,\_,\_) \\[1em]
2) & pts(V,C,H) & \texttt{:-} & \text{“}V = W\text{”}\ \texttt{\&} \\
 & & & pts(W,C,H) \\[1em]
3) & hpts(H,F,G) & \texttt{:-} & \text{“}V.F = W\text{”}\ \texttt{\&} \\
 & & & pts(W,C,G)\ \texttt{\&} \\
 & & & pts(V,C,H) \\[1em]
4) & pts(V,C,H) & \texttt{:-} & \text{“}V = W.F\text{”}\ \texttt{\&} \\
 & & & pts(W,C,G)\ \texttt{\&} \\
 & & & hpts(G,F,H) \\[1em]
5) & pts(V,D,H) & \texttt{:-} & CSinvokes(S,C,M,D)\ \texttt{\&} \\
 & & & formal(M,I,V)\ \texttt{\&} \\
 & & & actual(S,I,W)\ \texttt{\&} \\
 & & & pts(W,C,H)
\end{array}$$

Figure 12.29: Datalog program for context-sensitive points-to analysis

An additional argument, representing the context, must be given to the IDB predicate *pts*. $pts(V, C, H)$ says that variable V in context C can point to heap object H. All the rules are self-explanatory, perhaps with the exception of Rule 5. Rule 5 says that if the call site S in context C calls method M of context D, then the formal parameters in method M of context D can point to the objects pointed to by the corresponding actual parameters in context C.

12.6.3 Additional Observations About Sensitivity

What we have described is one formulation of context sensitivity that has been shown to be practical enough to handle many large real-life Java programs, using the tricks described briefly in the next section. Nonetheless, this algorithm cannot yet handle the largest of Java applications.

The heap objects in this formulation are named by their call site, but without context sensitivity. That simplification can cause problems. Consider the object-factory idiom where, all objects of the same type are allocated by the same routine. The current scheme would make all objects of that class share the same name. It is relatively simple to handle such cases by essentially inlining the allocation code. In general, it is desirable to increase the context sensitivity in the naming of objects. While it is easy to add context sensitivity of objects to the Datalog formulation, getting the analysis to scale to large programs is another matter.

Another important form of sensitivity is object sensitivity. An object-sensitive technique can distinguish between methods invoked on different receiver objects. Consider the scenario of a call site in a calling context where a variable is found to point to two different receiver objects of the same class. Their fields may point to different objects. Without distinguishing between the objects, a copy among fields of the `this` object reference will create spurious relationships unless we separate the analysis according to the receiver objects. Object sensitivity is more useful than context sensitivity for some analyses.

12.6.4 Exercises for Section 12.6

```
void p() {
      h: T a = new T();
      i: T b = new T();
     c1: T c = a.q(b);
}

T    q(T y) {
      j: T d = new T();
     c2:   d = this.q(d);
     c3:   d = d.q(y);
     c4:   d = d.r();
           return d;
}

T    r() {
           return this;
}
```

Figure 12.30: Code for Exercises 12.6.1 and 12.6.2

Exercise 12.6.1: What are all the contexts that would be distinguished if we apply the methods of this section to the code in Fig. 12.30?

! **Exercise 12.6.2 :** Perform a context sensitive analysis of the code in Fig. 12.30.

! **Exercise 12.6.3 :** Extend the Datalog rules of this section to incorporate the type and subtype information, following the approach of Section 12.5.

12.7 Datalog Implementation by BDD's

Binary Decision Diagrams (BDD's) are a method for representing boolean functions by graphs. Since there are 2^{2^n} boolean functions of n variables, no representation method is going to be very succinct on all boolean functions. However, the boolean functions that appear in practice tend to have a lot of regularity. It is thus common that one can find a succinct BDD for functions that one really wants to represent.

It turns out that the boolean functions that are described by the Datalog programs that we have developed to analyze programs are no exception. While succinct BDD's representing information about a program often must be found using heuristics and/or techniques used in commercial BDD-manipulating packages, the BDD approach has been quite successful in practice. In particular, it outperforms methods based on conventional database-management systems, because the latter are designed for the more irregular data patterns that appear in typical commercial data.

It is beyond the scope of this book to cover all of the BDD technology that has been developed over the years. We shall here introduce you to the BDD notation. We then suggest how one represents relational data as BDD's and how one could manipulate BDD's to reflect the operations that are performed to execute Datalog programs by algorithms such as Algorithm 12.18. Finally, we describe how to represent the exponentially many contexts in BDD's, the key to the success of the use of BDD's in context-sensitive analysis.

12.7.1 Binary Decision Diagrams

A BDD represents a boolean function by a rooted DAG. The interior nodes of the DAG are each labeled by one of the variables of the represented function. At the bottom are two leaves, one labeled 0 the other labeled 1. Each interior node has two edges to children; these edges are called "low" and "high." The low edge is associated with the case that the variable at the node has value 0, and the high edge is associated with the case where the variable has value 1.

Given a truth assignment for the variables, we can start at the root, and at each node, say a node labeled x, follow the low or high edge, depending on whether the truth value for x is 0 or 1, respectively. If we arrive at the leaf labeled 1, then the represented function is true for this truth assignment; otherwise it is false.

Example 12.27 : In Fig. 12.31 we see a BDD. We shall see the function it represents shortly. Notice that we have labeled all the "low" edges with 0 and

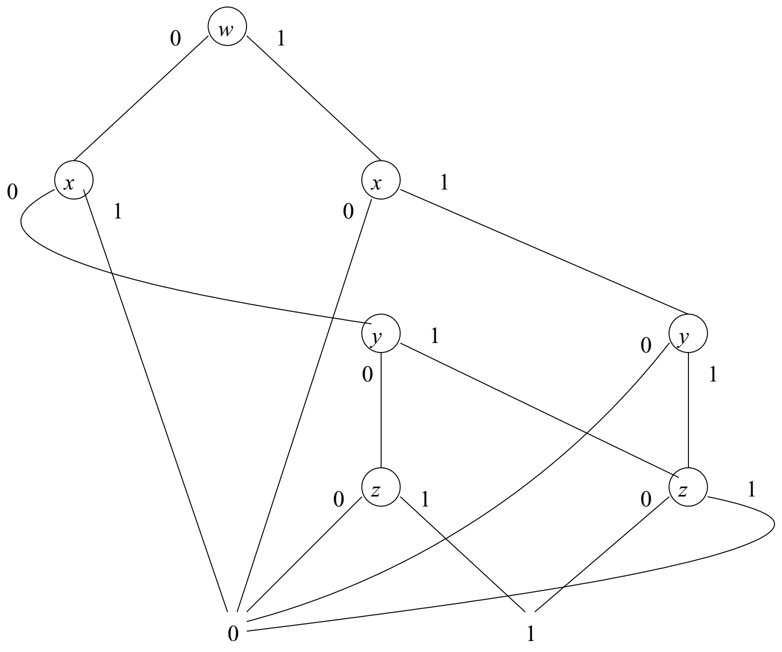

Figure 12.31: A binary decision diagram

all the "high" edges by 1. Consider the truth assignment for variables $wxyz$ that sets $w = x = y = 0$ and $z = 1$. Starting at the root, since $w = 0$ we take the low edge, which gets us to the leftmost of the nodes labeled x. Since $x = 0$, we again follow the low edge from this node, which takes us to the leftmost of the nodes labeled y. Since $y = 0$ we next move to the leftmost of the nodes labeled z. Now, since $z = 1$, we take the high edge and wind up at the leaf labeled 1. Our conclusion is that the function is true for this truth assignment.

Now, consider the truth assignment $wxyz = 0101$, that is, $w = y = 0$ and $x = z = 1$. We again start at the root. Since $w = 0$ we again move to the leftmost of the nodes labeled x. But now, since $x = 1$, we follow the high edge, which jumps to the 0 leaf. That is, we know not only that truth assignment 0101 makes the function false, but since we never even looked at y or z, any truth assignment of the form $01yz$ will also make the function have value 0. This "short-circuiting" ability is one of the reasons BDD's tend to be succinct representations of boolean functions. □

In Fig. 12.31 the interior nodes are in ranks — each rank having nodes with a particular variable as label. Although it is not an absolute requirement, it is convenient to restrict ourselves to *ordered BDD's*. In an ordered BDD, there is an order $x_1, x_2, \ldots, x_n$ to the variables, and whenever there is an edge from a parent node labeled x_i to a child labeled x_j, then $i < j$. We shall see that it is easier to operate on ordered BDD's, and from here we assume all BDD's are ordered.

Notice also that BDD's are DAG's, not trees. Not only will the leaves 0 and 1 typically have many parents, but interior nodes also may have several parents. For example, the rightmost of the nodes labeled z in Fig. 12.31 has two parents. This combination of nodes that would result in the same decision is another reason that BDD's tend to be succinct.

12.7.2 Transformations on BDD's

We alluded, in the discussion above, to two simplifications on BDD's that help make them more succinct:

1. *Short-Circuiting*: If a node N has both its high and low edges go to the same node M, then we may eliminate N. Edges entering N go to M instead.

2. *Node-Merging*: If two nodes N and M have low edges that go to the same node and also have high edges that go to the same node, then we may merge N with M. Edges entering either N or M go to the merged node.

It is also possible to run these transformations in the opposite direction. In particular, we can introduce a node along an edge from N to M. Both edges from the introduced node go to M, and the edge from N now goes to the introduced node. Note, however, that the variable assigned to the new node must be one of those that lies between the variables of N and M in the order. Figure 12.32 shows the two transformations schematically.

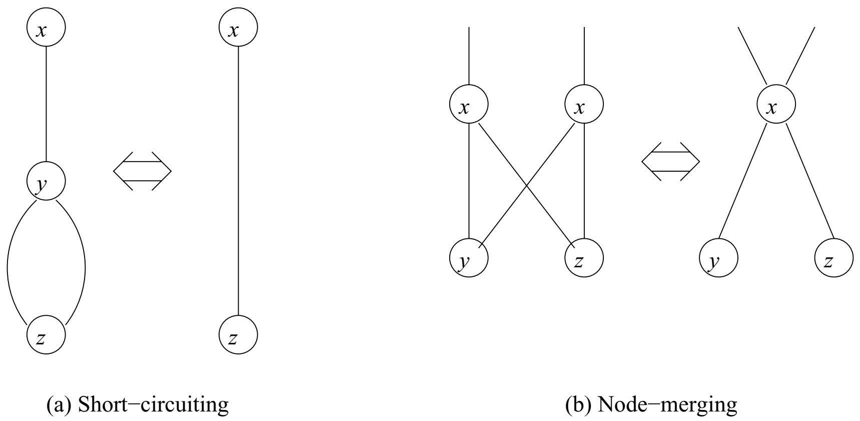

Figure 12.32: Transformations on BDD's

12.7.3 Representing Relations by BDD's

The relations with which we have been dealing have components that are taken from "domains." A domain for a component of a relation is the set of possible values that tuples can have in that component. For example, the relation $pts(V, H)$ has the domain of all program variables for its first component and the domain of all object-creating statements for the second component. If a domain has more than 2^{n-1} possible values but no more than 2^n values, then it requires n bits or boolean variables to represent values in that domain.

A tuple in a relation may thus be viewed as a truth assignment to the variables that represent values in the domains for each of the components of the tuple. We may see a relation as a boolean function that returns the value true for all and only those truth assignments that represent tuples in the relation. An example should make these ideas clear.

Example 12.28: Consider a relation $r(A, B)$ such that the domains of both A and B are $\{a, b, c, d\}$. We shall encode a by bits 00, b by 01, c by 10, and d by 11. Let the tuples of relation r be:

A	B
a	b
a	c
d	c

Let us use boolean variables wx to encode the first (A) component and variables yz to encode the second (B) component. Then the relation r becomes:

w	x	y	z
0	0	0	1
0	0	1	0
1	1	1	0

That is, the relation r has been converted into the boolean function that is true for the three truth-assignments $wxyz = 0001, 0010$, and 1110. Notice that these three sequences of bits are exactly those that label the paths from the root to the leaf 1 in Fig. 12.31. That is, the BDD in that figure represents this relation r, if the encoding described above is used. □

12.7.4 Relational Operations as BDD Operations

Now we see how to represent relations as BDD's. But to implement an algorithm like Algorithm 12.18 (incremental evaluation of Datalog programs), we need to manipulate BDD's in a way that reflects how the relations themselves are manipulated. Here are the principal operations on relations that we need to perform:

1. *Initialization*: We need to create a BDD that represents a single tuple of a relation. We'll assemble these into BDD's that represent large relations by taking the union.

2. *Union*: To take the union of relations, we take the logical OR of the boolean functions that represent the relations. This operation is needed not only to construct initial relations, but also to combine the results of several rules for the same head predicate, and to accumulate new facts into the set of old facts, as in the incremental Algorithm 12.18.

3. *Projection*: When we evaluate a rule body, we need to construct the head relation that is implied by the true tuples of the body. In terms of the BDD that represents the relation, we need to eliminate the nodes that are labeled by those boolean variables that do not represent components of the head. We may also need to rename the variables in the BDD to correspond to the boolean variables for the components of the head relation.

4. *Join*: To find the assignments of values to variables that make a rule body true, we need to "join" the relations corresponding to each of the subgoals. For example, suppose we have two subgoals $r(A, B)$ & $s(B, C)$. The join of the relations for these subgoals is the set of (a, b, c) triples such that (a, b) is a tuple in the relation for r, and (b, c) is a tuple in the relation for s. We shall see that, after renaming boolean variables in BDD's so the components for the two B's agree in variable names, the operation on BDD's is similar to the logical AND, which in turn is similar to the OR operation on BDD's that implements the union.

BDD's for Single Tuples

To initialize a relation, we need to have a way to construct a BDD for the function that is true for a single truth assignment. Suppose the boolean variables are $x_1, x_2, \ldots, x_n$, and the truth assignment is $a_1 a_2 \cdots a_n$, where each a_i is either 0 or 1. The BDD will have one node N_i for each x_i. If $a_i = 0$, then the high edge from N_i leads to the leaf 0, and the low edge leads to N_{i+1}, or to the leaf 1 if $i = n$. If $a_i = 1$, then we do the same, but the high and low edges are reversed.

This strategy gives us a BDD that checks whether each x_i has the correct value, for $i = 1, 2, \ldots, n$. As soon as we find an incorrect value, we jump directly to the 0 leaf. We only wind up at the 1 leaf if all variables have their correct value.

As an example, look ahead to Fig. 12.33(b). This BDD represents the function that is true if and only if $x = y = 0$, i.e., the truth assignment 00.

Union

We shall give in detail an algorithm for taking the logical OR of BDD's, that is, the union of the relations represented by the BDD's.

Algorithm 12.29: Union of BDD's.

INPUT: Two ordered BDD's with the same set of variables, in the same order.

OUTPUT: A BDD representing the function that is the logical OR of the two boolean functions represented by the input BDD's.

METHOD: We shall describe a recursive procedure for combining two BDD's. The induction is on the size of the set of variables appearing in the BDD's.

BASIS: Zero variables. The BDD's must both be leaves, labeled either 0 or 1. The output is the leaf labeled 1 if either input is 1, or the leaf labeled 0 if both are 0.

INDUCTION: Suppose there are k variables, $y_1, y_2, \ldots, y_k$ found among the two BDD's. Do the following:

1. If necessary, use inverse short-circuiting to add a new root so that both BDD's have a root labeled y_1.

2. Let the two roots be N and M; let their low children be N_0 and M_0, and let their high children be N_1 and M_1. Recursively apply this algorithm to the BDD's rooted at N_0 and M_0. Also, recursively apply this algorithm to the BDD's rooted at N_1 and M_1. The first of these BDD's represents the function that is true for all truth assignments that have $y_1 = 0$ and that make one or both of the given BDD's true. The second represents the same for the truth assignments with $y_1 = 1$.

3. Create a new root node labeled y_1. Its low child is the root of the first recursively constructed BDD, and its high child is the root of the second BDD.

4. Merge the two leaves labeled 0 and the two leaves labeled 1 in the combined BDD just constructed.

5. Apply merging and short-circuiting where possible to simplify the BDD.

□

Example 12.30: In Fig. 12.33(a) and (b) are two simple BDD's. The first represents the function x OR y, and the second represents the function

$$\texttt{NOT}\ x\ \text{AND}\ \texttt{NOT}\ y$$

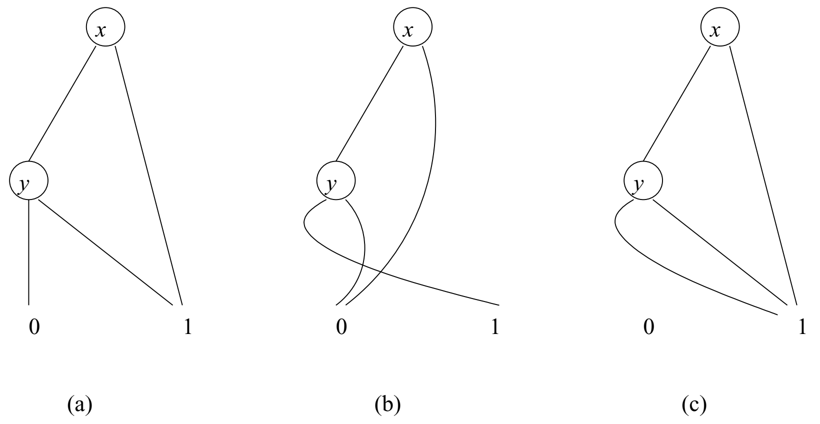

Figure 12.33: Constructing the BDD for a logical OR

Notice that their logical OR is the function 1 that is always true. To apply Algorithm 12.29 to these two BDD's, we consider the low children of the two roots and the high children of the two roots; let us take up the latter first.

The high child of the root in Fig. 12.33(a) is 1, and in Fig. 12.33(b) it is 0. Since these children are both at the leaf level, we do not have to insert nodes labeled y along each edge, although the result would be the same had we chosen to do so. The basis case for the union of 0 and 1 is to produce a leaf labeled 1 that will become the high child of the new root.

The low children of the roots in Fig. 12.33(a) and (b) are both labeled y, so we can compute their union BDD recursively. These two nodes have low children labeled 0 and 1, so the combination of their low children is the leaf labeled 1. Likewise, their high children are 1 and 0, so the combination is again the leaf 1. When we add a new root labeled x, we have the BDD seen in Fig. 12.33(c).

We are not done, since Fig. 12.33(c) can be simplified. The node labeled y has both children the node 1, so we can delete the node y and have the leaf 1 be the low child of the root. Now, both children of the root are the leaf 1, so we can eliminate the root. That is, the simplest BDD for the union is the leaf 1, all by itself. $\square$

12.7.5 Using BDD's for Points-to Analysis

Getting context-insensitive points-to analysis to work is already nontrivial. The ordering of the BDD variables can greatly change the size of the representation. Many considerations, as well as trial and error, are needed to come up with an ordering that allows the analysis to complete quickly.

It is even harder to get context-sensitive points-to analysis to execute because of the exponentially many contexts in the program. In particular, if we arbitrarily assign numbers to represent contexts in a call graph, we cannot handle even small Java programs. It is important that the contexts be numbered so that the binary encoding of the points-to analysis can be made very compact. Two contexts of the same method with similar call paths share a lot of commonalities, so it is desirable to number the n contexts of a method consecutively. Similarly, because pairs of caller-callees for the same call site share many similarities, we wish to number the contexts such that the numeric difference between each caller-callee pair of a call site is always a constant.

Even with a clever numbering scheme for the calling contexts, it is still hard to analyze large Java programs efficiently. Active machine learning has been found useful in deriving a variable ordering efficient enough to handle large applications.

12.7.6 Exercises for Section 12.7

Exercise 12.7.1: Using the encoding of symbols in Example 12.28, develop a BDD that represents the relation consisting of the tuples (b,b), (c,a), and (b,a). You may order the boolean variables in whatever way gives you the most succinct BDD.

! **Exercise 12.7.2:** As a function of n, how many nodes are there in the most succinct BDD that represents the exclusive-or function on n variables. That is, the function is true if an odd number of the n variables are true and false if an even number are true.

Exercise 12.7.3: Modify Algorithm 12.29 so it produces the intersection (logical AND) of two BDD's.

!! **Exercise 12.7.4:** Find algorithms to perform the following relational operations on the ordered BDD's that represent them:

a) Project out some of the boolean variables. That is, the function represented should be true for a given truth assignment α if there was any truth assignment for the missing variables that, together with α made the original function true.

b) Join two relations r and s, by combining a tuple from r with one from s whenever these tuples agree on the attributes that r and s have in common. It is really sufficient to consider the case where the relations have only two components, and one from each relation matches; that is, the relations are $r(A,B)$ and $s(B,C)$.

12.8 Summary of Chapter 12

✦ *Interprocedural Analysis*: A data-flow analysis that tracks information across procedure boundaries is said to be interprocedural. Many analyses, such as points-to analysis, can only be done in a meaningful way if they are interprocedural.

✦ *Call Sites*: Programs call procedures at certain points referred to as call sites. The procedure called at a site may be obvious, or it may be ambiguous, should the call be indirect through a pointer or a call of a virtual method that has several implementations.

✦ *Call Graphs*: A call graph for a program is a bipartite graph with nodes for call sites and nodes for procedures. An edge goes from a call-site node to a procedure node if that procedure may be called at the site.

✦ *Inlining*: As long as there is no recursion in a program, we can in principle replace all procedure calls by copies of their code, and use intraprocedural analysis on the resulting program. This analysis is in effect, interprocedural.

✦ *Flow Sensitivity and Context-Sensitivity*: A data-flow analysis that produces facts that depend on location in the program is said to be flow-sensitive. If the analysis produces facts that depend on the history of procedure calls is said to be context-sensitive. A data-flow analysis can be either flow- or context-sensitive, both, or neither.

✦ *Cloning-Based Context-Sensitive Analysis*: In principle, once we establish the different contexts in which a procedure can be called, we can imagine that there is a clone of each procedure for each context. In that way, a context-insensitive analysis serves as a context-sensitive analysis.

✦ *Summary-Based Context-Sensitive Analysis*: Another approach to interprocedural analysis extends the region-based analysis technique that was described for intraprocedural analysis. Each procedure has a transfer function and is treated as a region at each place where that procedure is called.

✦ *Applications of Interprocedural Analysis*: An important application requiring interprocedural analysis is the detection of software vulnerabilities. These are often characterized by having data read from an untrusted input source by one procedure and used in an exploitable way by another procedure.

✦ *Datalog*: The language Datalog is a simple notation for if-then rules that can be used to describe data-flow analyses at a high level. Collections of Datalog rules, or Datalog programs, can be evaluated using one of several standard algorithms.

✦ *Datalog Rules*: A Datalog rule consists of a body (antecedent) and head (consequent). The body is one or more atoms, and the head is an atom. Atoms are predicates applied to arguments that are variables or constants.

The atoms of the body are connected by logical AND, and an atom in the body may be negated.

✦ *IDB and EDB Predicates*: EDB predicates in a Datalog program have their true facts given a-priori. In a data-flow analysis, these predicates correspond to the facts that can be obtained from the code being analyzed. IDB predicates are defined by the rules themselves and correspond in a data-flow analysis to the information we are trying to extract from the code being analyzed.

✦ *Evaluation of Datalog programs*: We apply rules by substituting constants for variables that make the body true. Whenever we do so, we infer that the head, with the same substitution for variables, is also true. This operation is repeated, until no more facts can be inferred.

✦ *Incremental Evaluation of Datalog Programs*: An efficiency improvement is obtained by doing incremental evaluation. We perform a series of rounds. In one round, we consider only substitutions of constants for variables that make at least one atom of the body be a fact that was just discovered on the previous round.

✦ *Java Pointer Analysis*: We can model pointer analysis in Java by a framework in which there are reference variables that point to heap objects, which may have fields that point to other heap objects. An insensitive pointer analysis can be written as a Datalog program that infers two kinds of facts: a variable can point to a heap object, or a field of a heap object can point to another heap object.

✦ *Type Information to Improve Pointer Analysis*: We can get more precise pointer analysis if we take advantage of the fact that reference variables can only point to heap objects that are of the same type as the variable or a subtype.

✦ *Interprocedural Pointer Analysis*: To make the analysis interprocedural, we must add rules that reflect how parameters are passed and return values assigned to variables. These rules are essentially the same as the rules for copying one reference variable to another.

✦ *Call-Graph Discovery*: Since Java has virtual methods, interprocedural analysis requires that we first limit what procedures can be called at a given call site. The principal way to discover limits on what can be called where is to analyze the types of objects and take advantage of the fact that the actual method referred to by a virtual method call must belong to an appropriate class.

✦ *Context-Sensitive Analysis*: When procedures are recursive, we must condense the information contained in call strings into a finite number of contexts. An effective way to do so is to drop from the call string any call site where a procedure calls another procedure (perhaps itself) with which it is mutually recursive. Using this representation, we can modify the rules for intraprocedural pointer analysis so the context is carried along in predicates; this approach simulates cloning-based analysis.

✦ *Binary Decision Diagrams*: BDD's are a succinct representation of boolean functions by rooted DAG's. The interior nodes correspond to boolean variables and have two children, low (representing truth value 0) and high (representing 1). There are two leaves labeled 0 and 1. A truth assignment makes the represented function true if and only if the path from the root in which we go to the low child if the variable at a node is 0 and to the high child otherwise, leads to the 1 leaf.

✦ *BDD's and Relations*: A BDD can serve as a succinct representation of one of the predicates in a Datalog program. Constants are encoded as truth assignments to a collection of boolean variables, and the function represented by the BDD is true if an only if the boolean variables represent a true fact for that predicate.

✦ *Implementing Data-Flow Analysis by BDD's*: Any data-flow analysis that can be expressed as Datalog rules can be implemented by manipulations on the BDD's that represent the predicates involved in those rules. Often, this representation leads to a more efficient implementation of the data-flow analysis than any other known approach.

12.9 References for Chapter 12

Some of the basic concepts in interprocedural analysis can be found in [1, 6, 7, and 21]. Callahan et al. [11] describe an interprocedural constant-propagation algorithm.

Steensgaard [22] published the first scalable pointer-alias analysis. It is context-insensitive, flow-insensitive, and equivalence-based. A context-insensitive version of the inclusion-based points-to analysis was derived by Andersen [2]. Later, Heintze and Tardieu [15] described an efficient algorithm for this analysis. Fähndrich, Rehof, and Das [14] presented a context-sensitive, flow-insensitive, equivalence-based analysis that scales to large programs like `gcc`. Notable among previous attempts to create a context-sensitive, inclusion-based points-to analysis is Emami, Ghiya, and Hendren [13], which is a cloning-based context-sensitive, flow-sensitive, inclusion-based, points-to algorithm.

Binary decision diagrams (BDD's) first appeared in Bryant [9]. Their use for data-flow analysis was by Ball and Rajamani [4]. The application of BDD's to insensitive pointer analysis is reported by Zhu [25] and Berndl et al. [8]. Whaley and Lam [24] describe the first context-sensitive, flow-insensitive, inclusion-based algorithm that has been shown to apply to real-life applications. The paper describes a tool called `bddbddb` that automatically translates analysis described in Datalog into BDD code. Object-sensitivity was introduced by Milanova, Rountev, and Ryder [18].

For a discussion of Datalog, see Ullman and Widom [23]. Also see Lam et al. [16] for a discussion of the connection of data-flow analysis to Datalog.

The Metal code checker is described by Engler et al. [12] and the PREfix checker was created by Bush, Pincus, and Sielaff [10]. Ball and Rajamani [4] developed a program analysis engine called SLAM using model checking and symbolic execution to simulate all possible behaviors of a system. Ball et al. [5] have created a static analysis tool called SDV based on SLAM to find API usage errors in C device-driver programs by applying BDD's to model checking.

Livshits and Lam [17] describe how context-sensitive points-to analysis can be used to find SQL vulnerabilities in Java web applications. Ruwase and Lam [20] describe how to keep track of array extents and insert dynamic bounds checks automatically. Rinard et al. [19] describe how to extend arrays dynamically to accommodate for the overflowed contents. Avots et al. [3] extend the context-sensitive Java points-to analysis to C and show how it can be used to reduce the cost of dynamic detection of buffer overflows.

1. Allen, F. E., "Interprocedural data flow analysis," *Proc. IFIP Congress 1974*, pp. 398–402, North Holland, Amsterdam, 1974.

2. Andersen, L., *Program Analysis and Specialization for the C Programming Language*, Ph.D. thesis, DIKU, Univ. of Copenhagen, Denmark, 1994.

3. Avots, D., M. Dalton, V. B. Livshits, and M. S. Lam, "Improving software security with a C pointer analysis," *ICSE 2005: Proc. 27th International Conference on Software Engineering*, pp. 332–341.

4. Ball, T. and S. K. Rajamani, "A symbolic model checker for boolean programs," *Proc. SPIN 2000 Workshop on Model Checking of Software*, pp. 113–130.

5. Ball, T., E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. Rajamani, and A. Ustuner, "Thorough static analysis of device drivers," *EuroSys* (2006), pp. 73–85.

6. Banning, J. P., "An efficient way to find the side effects of procedural calls and the aliases of variables," *Proc. Sixth Annual Symposium on Principles of Programming Languages* (1979), pp. 29–41.

7. Barth, J. M., "A practical interprocedural data flow analysis algorithm," *Comm. ACM* **21**:9 (1978), pp. 724–736.

8. Berndl, M., O. Lohtak, F. Qian, L. Hendren, and N. Umanee, "Points-to analysis using BDD's," *Proc. ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pp. 103–114.

9. Bryant, R. E., "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. on Computers* **C-35**:8 (1986), pp. 677–691.

10. Bush, W. R., J. D. Pincus, and D. J. Sielaff, "A static analyzer for finding dynamic programming errors," *Software–Practice and Experience*, **30**:7 (2000), pp. 775–802.

11. Callahan, D., K. D. Cooper, K. Kennedy, and L. Torczon, "Interprocedural constant propagation," *Proc. SIGPLAN 1986 Symposium on Compiler Construction, SIGPLAN Notices*, **21**:7 (1986), pp. 152–161.

12. Engler, D., B. Chelf, A. Chou, and S. Hallem, "Checking system rules using system-specific, programmer-written compiler extensions," *Proc. Sixth USENIX Conference on Operating Systems Design and Implementation* (2000). pp. 1–16.

13. Emami, M., R. Ghiya, and L. J. Hendren, "Context-sensitive interprocedural points-to analysis in the presence of function pointers," *Proc. SIGPLAN Conference on Programming Language Design and Implementation* (1994), pp. 224–256.

14. Fähndrich, M., J. Rehof, and M. Das, "Scalable context-sensitive flow analysis using instantiation constraints," *Proc. SIGPLAN Conference on Programming Language Design and Implementation* (2000), pp. 253–263.

15. Heintze, N. and O. Tardieu, ""Ultra-fast aliasing analysis using CLA: a million lines of C code in a second," *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (2001), pp. 254–263.

16. Lam, M. S., J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel, "Context-sensitive program analysis as database queries," *Proc. 2005 ACM Symposium on Principles of Database Systems*, pp. 1–12.

17. Livshits, V. B. and M. S. Lam, "Finding security vulnerabilities in Java applications using static analysis," *Proc. 14th USENIX Security Symposium* (2005), pp. 271–286.

18. Milanova, A., A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to and side-effect analyses for Java," *Proc. 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1–11.

19. Rinard, M., C. Cadar, D. Dumitran, D. Roy, and T. Leu, "A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors)," *Proc. 2004 Annual Computer Security Applications Conference*, pp. 82–90.

20. Ruwase, O. and M. S. Lam, "A practical dynamic buffer overflow detector," *Proc. 11th Annual Network and Distributed System Security Symposium* (2004), pp. 159–169.

21. Sharir, M. and A. Pnueli, "Two approaches to interprocedural data flow analysis," in S. Muchnick and N. Jones (eds.) *Program Flow Analysis: Theory and Applications*, Chapter 7, pp. 189–234. Prentice-Hall, Upper Saddle River NJ, 1981.

22. Steensgaard, B., "Points-to analysis in linear time," *Twenty-Third ACM Symposium on Principles of Programming Languages* (1996).

23. Ullman, J. D. and J. Widom, *A First Course in Database Systems*, Prentice-Hall, Upper Saddle River NJ, 2002.

24. Whaley, J. and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," *Proc. ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pp. 131–144.

25. Zhu, J., "Symbolic Pointer Analysis," *Proc. International Conference in Computer-Aided Design* (2002), pp. 150–157.