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

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

The components are as follows:

- H and $B_1, B_2, \ldots, B_n$ are literals — either atoms or negated atoms.

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 this 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:

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

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. *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 *def*$(b_1, 1, x)$ is true, *def*$(b_1, 3, x)$ is true, and *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. $succ(B, N, C)$ 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 $succ(b_2, 5, b_1)$ is true.

There is one IDB predicate, $rd(B, N, C, M, X)$. 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.

$$
\begin{array}{lll}
1) & rd(B,N,B,N,X) & \text{:-} \quad def(B,N,X) \\
 & & \\
2) & rd(B,N,C,M,X) & \text{:-} \quad rd(B,N-1,C,M,X) \ \& \\
 & & \phantom{\text{:-}} \quad def(B,N,Y) \ \& \\
 & & \phantom{\text{:-}} \quad X \neq Y \\
 & & \\
3) & rd(B,0,C,M,X) & \text{:-} \quad rd(D,N,C,M,X) \ \& \\
 & & \phantom{\text{:-}} \quad succ(D,N,B)
\end{array}
$$

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. □

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 $B = b_1$, $N = 2$, $P = p$, $T =$ 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. □

```
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

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

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:

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

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.length();
```

Here o is declared to be an `Object`. Without analyzing what o refers to, all possible methods called "length" 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 j" 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 = 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.

```
1)      pts(V,H)   :-  "H : T V = new T"

2)      pts(V,H)   :-  "V = W" &
                       pts(W,H)

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)
```

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 variables. 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 = 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

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

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

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

we would infer accurately that a can point to h, and b and c can point to g. 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

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

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.