CHAPTER

Deducing Errors

7

In this chapter, we begin exploring the techniques for creating hypotheses introduced in Chapter 6. We start with *deduction* techniques—reasoning from the *abstract* program code to the *concrete* program run. In particular, we present *program slicing*, an automated means of determing possible origins of a variable value. Using program slicing, one can effectively narrow down the number of possible infection sites.

7.1 ISOLATING VALUE ORIGINS

Oops! We have observed something that should not happen—the program has reached some state it never should have reached. How did it get into this state? Kernighan and Pike (1999) give a hint:

> *Something impossible occurred, and the only solid information is that it really did occur. So we must think backwards from the result to discover the reasons.*

What does "thinking backwards" mean here? One of the main applications of program code during debugging is to identify *relevant statements* that *could* have caused the failure—and, in the same step, to identify the *irrelevant statements* that *could not* have caused the failure in any way. This allows the programmer to neglect the irrelevant statements—and to focus on the relevant ones instead. These relevant statements are found by following back the *possible origins* of the result—that is, "thinking backward." As an example of relevant and irrelevant statements, consider the following piece of BASIC code.

```
10 INPUT X
20 Y = 0
30 X = Y
40 PRINT "X = ", X
```

This piece of code outputs the value of `X`, which is always a zero value. Where does this value come from? We can trace our way backward from the printing statement in line 40 and find that `X`'s value was assigned from `Y` (line 30), which in turn got its zero value in line 20. The input to `X` in line 10 (and anything else that might be inserted before line 20) is irrelevant for the value of `X` in line 40.

Applying this relevant/irrelevant scheme, we can effectively narrow down our search space—simply by focusing on the relevant values and neglecting the irrelevant ones. This is shown in Figure 1.5. During the execution of a program, only a few values (marked with !) can possibly influence the failing state. Knowing these relevant values can be crucial for effective debugging.

How does one determine whether a value (or a statement that generates this value) is relevant for a failure or not? To do so, we need not execute the program. We can do so by pure *deduction*—that is, reasoning from the abstract (program code) to what might happen in the concrete (run). By deducing from the code, we can abstract over *all* (or at least several) runs to determine properties that hold for all runs (for instance, properties about relevant and irrelevant values). The key question is:

WHAT GENERAL ERRORS CAN WE DEDUCE FROM CODE ALONE?

7.2 UNDERSTANDING CONTROL FLOW

Deducing from program code is more than just sitting in front of the code and trying to understand it. A few basic principles can effectively guide the search for relevant values—and incidentally, these principles are the same for seasoned programmers as for automated analysis tools.

As an ongoing example, consider the `fibo.c` program shown in Example 7.1. This program displays the first nine members of the *Fibonacci sequence* 1, 1, 2, 3, 5, 8, . . . , in which each element is the sum of its two predecessors. Formally, the nth element of the Fibonacci sequence is defined as

$$\text{fib}(n) = \begin{cases} 1 & \text{for } n = 0 \vee n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{otherwise.} \end{cases}$$

Unfortunately, the implementation in Example 7.1 has a defect. Its output is:

```
$ gcc -o fibo fibo.c
$ ./fibo
fib(9)=55
fib(8)=34
fib(7)=21
fib(6)=13
fib(5)=8
fib(4)=5
fib(3)=3
fib(2)=2
fib(1)=134513905
$ _
```

As we see, the value of `fib(1)` is wrong. `fib(1)` should be 1 instead of the arbitrary value reported here.

EXAMPLE 7.1: `fibo.c` prints out Fibonacci numbers—except for 1

```
 1  /* fibo.c -- Fibonacci C program to be debugged */
 2
 3  #include <stdio.h>
 4
 5  int fib(int n)
 6  {
 7      int f, f0 = 1, f1 = 1;
 8
 9      while (n > 1) {
10           n = n - 1;
11          f = f0 + f1;
12          f0 = f1;
13          f1 = f;
14      }
15
16      return f;
17  }
18
19  int main()
20  {
21      int n = 9;
22
23      while (n > 0)
24      {
25          printf("fib(%d)=%dN", n, fib(n));
26          n = n - 1;
27      }
28
29      return 0;
30  }
```

How does the bad return value of `fib(1)` come to be? As an experienced programmer, you can probably identify the problem in half a minute or less (just read the source code in Example 7.1). Let's try, though, to do this in a little more systematic fashion—after all, we want our process to scale and we eventually want to automate parts of it.

The first thing to reason about when tracking value origins through source code is to identify those regions of code that could have influenced the value *simply because they were executed.* In our example, this is particularly easy. We only need to consider the code of the `fib()` function, as the defect occurs between its call and its return.

Because earlier statements may influence later statements (but not vice versa), we must now examine the *order* in which the statements were executed. We end up in a *control flow graph*, as shown in Figure 7.1. Such a graph is built as follows:

- Each statement of a program is mapped to a *node.* (In compiler construction—the origin of control flow graphs—statements that must follow each other are

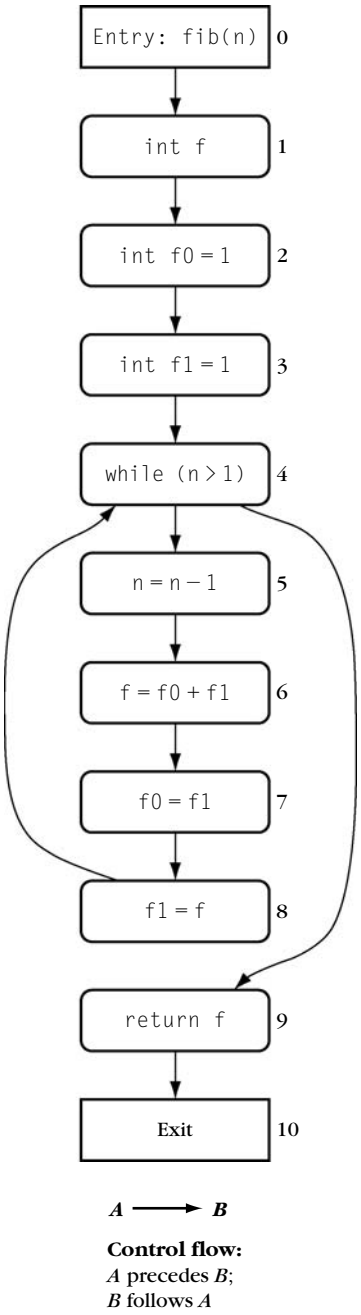

FIGURE 7.1

The `fib()` control flow graph.

combined into nodes called *basic blocks*. In Figure 7.1, for instance, nodes 1–3 and 5–8 would form basic blocks.)

- Edges connecting the nodes represent the possible *control flow* between the statements—a possible execution sequence of statements. An edge from a statement *A* to a statement *B* means that during execution statement *B* may immediately be executed after statement *A*.
- An entry and exit node represent the beginning and the end of the program or function.

In Figure 7.1, for instance, you can see that after `f1 = f` (statement 8); we always have to check the loop condition (statement 4) before possibly returning from `fib()` (statements 9 and 10).

For structured programming languages such as C, a control flow graph (such as that shown in Figure 7.1) is straightforward to produce. All one needs is a *pattern* for each control structure, as sketched in Figure 7.2. The actual control flow graph for a program is composed from such patterns.

There are situations, though, where the control flow cannot be determined from such patterns. All of these impose difficult situations for debugging. List 7.1 outlines the most important caveats you should be aware of.

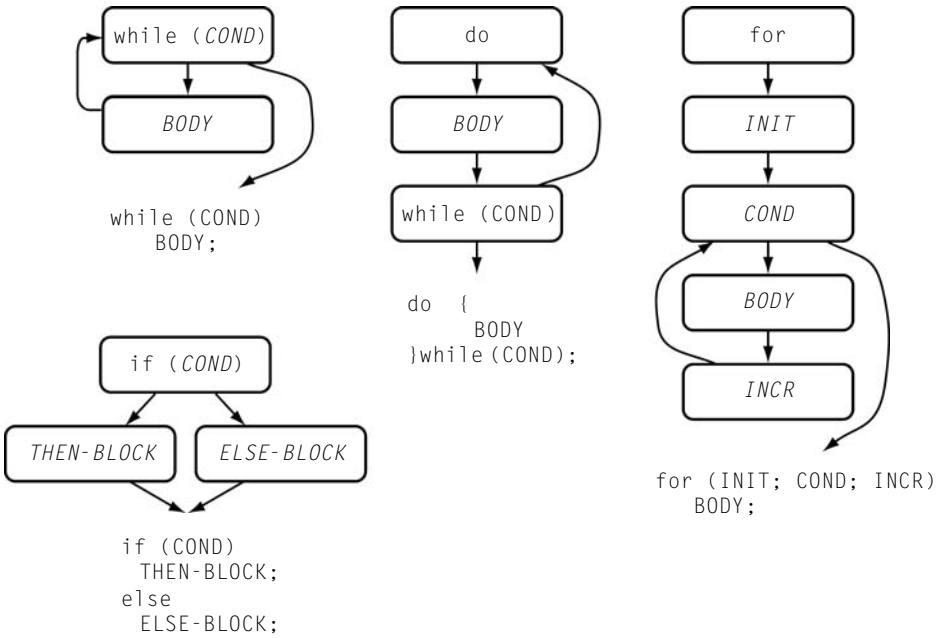

FIGURE 7.2

Some common control flow patterns.

LIST 7.1: Control Flow Caveats

Jumps and gotos. A jump or *goto* is an unconditional transfer of control; `goto 50` means to resume execution at line or label 50. Unconstrained gotos can make reasoning about programs much more difficult—in particular, if they involve jumps into loop or function bodies. (In technical terms, this may make the control flow graph *unstructured* or *irreducible*.) Fortunately, most programmers (and languages) avoid `goto` statements as a whole or use them only to jump to the end of a block.

Indirect jumps. Even more complicated than a `goto` to a specific line is a *computed* `goto`, also known as an indirect jump. A statement such as `goto X` transfers control to the statement or address as stored in the variable `X`.

Unconstrained indirect jumps make reasoning about control flow very difficult because in principle they can be followed by an arbitrary statement. Fortunately, indirect jumps are almost exclusively used for dynamic function dispatch. The address of a function is taken from some table and then used for invocation. Languages such as C and C++ provide such mechanisms as *function pointers*.

Dynamic dispatch. A very constrained form of indirect jumps is found in object-oriented languages. A call such as `shape.draw()` invokes the `draw()` method of the object referenced by `shape`. The actual destination of the call is resolved at runtime, depending on the class of the object. If the object is a rectangle, then `Rectangle.draw()` is called. If it is a circle, then `Circle.draw()` is called.

Dynamic dispatch is a powerful tool, but also a frequent source of misunderstanding when reasoning about program code. For every method call, one must be aware of the possible destinations.

Exceptions. By throwing an *exception*, a function can transfer control back to its caller, which either must handle the exception or rethrow it to its respective caller. (Instead of the caller, a surrounding block may also handle or rethrow the exception.)

In the presence of exceptions, one must be aware that control may never reach the "official" end of a function but be transferred directly to the caller. Be sure that an exception does not go by unnoticed, such that you know that it has occurred.

7.3 TRACKING DEPENDENCES

The control flow graph is the basis for all deduction about programs, as it shows how information propagates along the sequence of statements. Let's go a little more into detail here. Exactly how do individual statements affect the information flow? And how are statements affected by the information flow?

7.3.1 Effects of Statements

To contribute to the computation, every statement of the program must (at least potentially) affect the information flow in some way. We distinguish the following two types of effects.

Write

A statement can *change the program state* (i.e., assign a value to a variable). For example, the statement `v1 = 1` writes a value to the variable `v1`.

The "program state" considered here is a very general term. For instance, printing some text on an output device changes the state of the device. Sending a message across the network changes the state of attached devices. To some extent, the "program state" thus becomes the state of the world. Therefore, it is useful to limit the considered state—for instance, to the hardware boundaries.

Control

A statement may *change the program counter*—that is, determine which statement is to be executed next. In Figure 7.1, the `while` statement determines whether the next statement is either 5 or 9. Obviously, we are only talking about *conditional* changes to the program counter here—that is, statements that have at least two possible successors in the control flow graph, dependent on the program state.

In principle, one may consider the program counter as part of the program state. In practice, though, locations in the program state, and locations in the program code, are treated conceptually as separate dimensions of space and time. Figure 1.1 uses this distinction to represent the intuition about what is happening in a program run.

7.3.2 Affected Statements

Affecting the information flow by writing state or controlling execution represents the *active* side of how a statement affects the information flow. However, statements are also *passively* affected by other statements. For example:

- *Read.* A statement can *read the program state* (i.e., read a value from a variable). For example, the statement `v2 = v1 + 1` reads a value from the variable `v1`. Consequently, the effect of the statement is affected by the state of `v1`.

  Just as in the writing state, the "program state" considered here is a very general term. For instance, reading some text from an input device reads the state of the device. Reading a message across the network reads the state of attached devices.
- *Execution.* To have any effect, a statement must be *executed.* Consequently, if the execution of a statement *B* is potentially controlled by another statement *A*, then *B* is affected by *A*.

For each statement *S* of a program, we can determine what part of the state is being read or written by *S* (as deduced from the actual program code), and which other statements are controlled by *S* (as deduced from the control flow graph). As an example, consider Table 7.1, which lists the actions of the statements in the `fib()` program.

Table 7.1 Effects of the `fib()` Statements

	Statement	Reads	Writes	Controls
0	`fib(n)`		`n`	1–10
1	`int f`		`f`	
2	`f0 = 1`		`f0`	
3	`f1 = 1`		`f1`	
4	`while (n > 1)`	`n`		5–8
5	`n = n - 1`	`n`	`n`	
6	`f = f0 + f1`	`f0, f1`	`f`	
7	`f0 = f1`	`f1`	`f0`	
8	`f1 = f`	`f`	`f1`	
9	`return f`	`f`	⟨return value⟩	

Note: Each statement reads or writes a variable, or controls whether other statements are executed.

7.3.3 Statement Dependences

Given the effects of statements, as well as the statements thereby affected, we can construct *dependences* between statements, showing how they influence each other. We distinguish the following two types of dependences.

Data dependence

A statement B is *data dependent* on a statement A if

- A writes some variable V (or more generally, part of the program state) that is being read by B, and
- there is at least one path in the control flow graph from A to B in which V is not being written by some other statement.

In other words, the outcome of A can influence the data read by B.

Figure 7.3 shows the data dependences in the `fib()` program. By following the dashed arrows on the right side, one can determine where the data being written by some statement are being read by another statement. For instance, the variable `f0` being written by statement 2, `f0 = 1`, is read in statement 6, `f = f0 + f1`.

Control dependence

A statement B is *control dependent* on a statement A if B's execution is potentially controlled by A. In other words, the outcome of A determines whether B is executed.

The dotted arrows on the left side of Figure 7.3 show the control dependences in `fib()`. Each statement of the body of the `while` loop is dependent on entering

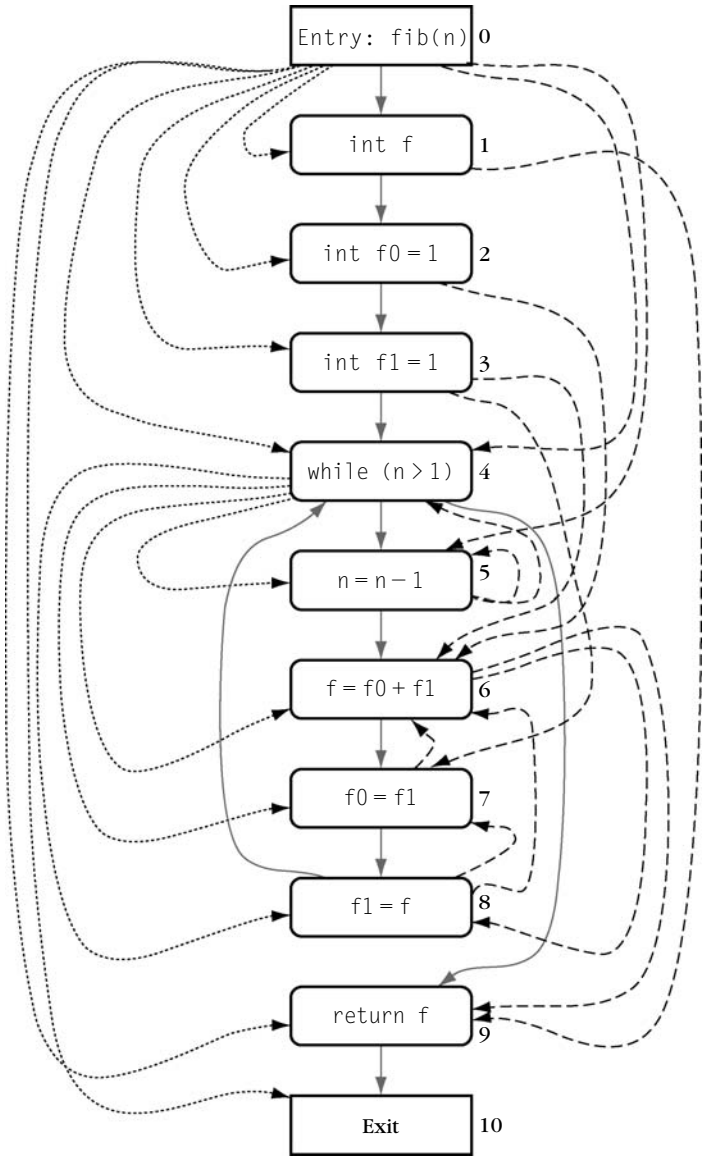

FIGURE 7.3

`fib()` dependence graph.

the loop (and thus dependent on the head of the `while` loop). All other statements are dependent on the entry of the function (as it determines whether the body is actually executed).

The control and data dependences of a program, as shown in Figure 7.3, form a graph—the *program-dependence graph.* This graph is the basis for a number of program-analysis techniques, as it reflects all influences within a program.

7.3.4 Following Dependences

Following the control and data dependences in the program-dependence graph, one can determine which statements influence which other statements—in terms of data or control, or both. In particular, one can answer two important questions:

1. *Where does this value go to?* Given a statement *S* writing a variable *V*, we can determine the impact of *S* by checking which other statements are dependent on *S*—and which other statements are dependent on these. Let's follow the dependences to see what happens when we call `fib()`. In Figure 7.3, the value of `n` is being used in the `while` head (statement 4) as well as in the `while` body (statement 5). Because the `while` head also controls the assignments to `f`, `f0`, and `f1` (statements 6–8), the value of `n` also determines the values of `f`, `f0`, and `f1`—and eventually, the returned value `f`. This is how `fib()` is supposed to work.
2. *Where does this value come from?* Given a statement *S* reading a variable *V*, we can determine the statements that possibly influenced *V* by following back the dependences of *S*. Let's now follow the dependences to see where the arbitrary value returned by `fib(1)` comes from. In Figure 7.3, consider the return value `f` in statement 9. The value of `f` can come from two sources. It can be computed in statement 6 from `f0` and `f1` (and, following their control dependences, eventually `n`). However, it also can come from statement 1, which is the declaration of `f`. In the C language, a local variable is not initialized and thus may hold an arbitrary value. It is exactly this value that is returned if the `while` body is not executed. In other words, this is the arbitrary value returned by `fib(1)`.

7.3.5 Leveraging Dependences

Following dependences through programs is a common technique for finding out the origins of values. But from where does one get the dependences?

Typically, programmers *implicitly* determine dependences while reading the code. Assessing the effect of each statement is part of understanding the program. Studies have shown that programmers effectively follow dependences while debugging, either in a *forward* fashion to assess the impact of a statement or in a *backward* fashion to find out which other parts of the program might have influenced a statement. As Weiser (1982) puts it: "When debugging, programmers view programs in ways that need not conform to the programs' textual or modular structures." Thus, dependences become an important guide for *navigating through the code.*

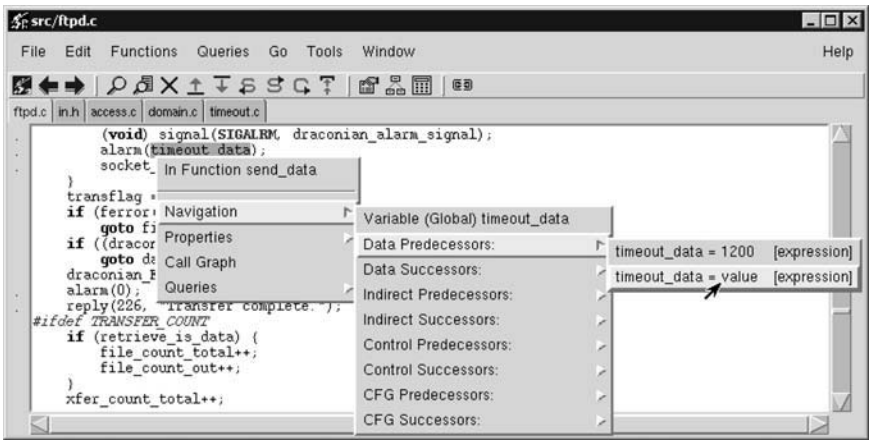

FIGURE 7.4

Following dependences in CODESURFER. For each variable, one can query the predecessors and successors along the dependence graph.

Obtaining *explicit* dependences such that they can be leveraged in tools is also feasible and is part of several advanced program-analysis tools. Figure 7.4 shows a screenshot of the CODESURFER tool, one of the most advanced program-analysis tools available. Rather than visualizing dependences as in Figure 7.3, CODESURFER allows programmers to explore the dependences interactively by navigating to predecessors and successors according to data and control dependences.

7.4 SLICING PROGRAMS

Using dependences, one can check for specific defect patterns and focus on specific *subsets* of the program being debugged (the subset that may have influenced a specific statement or the subset that may be influenced by a specific statement). Such a subset is called a *slice,* and the corresponding operation is called *slicing.*

7.4.1 Forward Slices

By following all dependences from a given statement A, one eventually reaches *all statements of which the read variables or execution could ever be influenced by* A. This set of statements is called a *program slice*, or more specifically the *forward slice* of $S^F(A)$. Formally, it consists of all statements that (transitively) depend on A:

$$S^F(A) = \{B | A \rightarrow^+ B\}$$

In a slice $S^F(A)$, the originating statement A is called the *slicing criterion.*

As an example for a forward slice, consider Figure 7.3. The forward slice originating at statement 2, `f0 = 1`, first includes statement 6, `f = f0 + f1`. Via `f`, the

slice also includes statement 8, `f1 = f`, and statement 9, `return f`. Via `f1`, the slice finally also includes statement 7, `f0 = f1`. Overall, the forward slice $S^F(2)$ is thus $S^F(2) = \{2, 6, 7, 8, 9\}$.

More important than the statements included in a slice are the statements *not* included in a slice, in that these can be in no way affected by the original statement. In our case, the statements excluded are not just the statements 0 and 1 (hardly surprising, as they are always executed *before* statement 2) but statements 4 and 5—the head of the `while` loop. In other words, the execution of the `while` loop is independent of the initial value of `f0`.

7.4.2 Backward Slices

The term *forward slice* implies that there is also a backward slice. To compute the backward slice of B, one proceeds *backward* along the dependences. Thus, we can determine all statements that could have influenced B. This is most useful in determining where the program state at execution of B could have come from. Formally, the backward slice $S^B(B)$ is computed as

$$S^B(B) = \{A | A \rightarrow^* B\}$$

Again, B is called the *slicing criterion* of $S^B(B)$.

As an example for a backward slice, again consider Figure 7.3. The backward slice of statement 9, `return f`, first includes statement 1, `int f`, and statement 6, `f = f0 + f1`. Because statement 6 is control dependent on the `while` loop, the slice also includes statement 4, `while (n > 1)`, and statement 5, `n = n + 1`, on which statement 4 is data dependent. Because `f0` and `f1` are computed in statements 7 and 8 and are initialized in statements 2 and 3, all of these statements also become part of the backward slice—which means that the slice includes *all statements* of `fib()`, or $S^B(9) = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

Having *all* statements being included in the backward slice of a returned value is quite typical. After all, if some statement would *not* contribute to the computation this would be a code smell, as discussed in Section 7.5. However, if a function computes *multiple values*, a backward slice for one of these values may well result in a true subset. As an example, consider the program in Example 7.2(a), which computes the sum and the product of the range of integers $[a, b]$. The backward slice of `write(mul)`, shown in Example 7.2(b) slices away all those parts that compute `sum` and cannot influence the computation or output of `mul` in any way.

7.4.3 Slice Operations

To further focus on specific behavior, one can *combine* several slices. Typical operations on slices include the following:

Chops. The intersection between a forward and a backward slice is called a *chop.* Chops are useful for finding out *how* some statement A (originating the forward slice) influences another statement B (originating the backward slice).

EXAMPLE 7.2: Slicing away irrelevant program parts: (a) entire program, (b)backward slice $S^B(13)$, (c) backbone $S^B(12) \cap S^B(13)$, and (d) dice $S^B \times S^B(12)$

```
 1  int main() {
 2    int a, b, sum, mul;
 3    sum = 0;
 4    mul = 1;
 5    a = read();
 6    b = read();
 7    while (a <= b) {
 8      sum = sum + a;
 9      mul = mul * a;
10      a = a + 1;
11    }
12    write(sum);
13    write(mul);
14  }
```

(a)

```
 1  int main() {
 2
 3
 4    mul = 1;
 5    a = read();
 6    b = read();
 7    while (a <= b) {
 8
 9      mul = mul * a;
10      a = a + 1;
11    }
12
13    write(mul);
14  }
```

(b)

```
 1  int main() {
 2
 3
 4
 5    a = read();
 6    b = read();
 7    while (a <= b) {
 8
 9
10      a = a + 1;
11    }
12
13
14  }
```

(c)

```
 1  int main() {
 2
 3
 4    mul = 1;
 5
 6
 7
 8
 9      mul = mul * a;
10
11
12
13    write(mul);
14  }
```

(d)

In the `fib()` program from Figure 7.3, for instance, a chop from statements 3–7 also includes statements 6 and 8, thus denoting all possible paths by which the initial value of `f1` could have influenced `f0`.

Backbones. The *intersection* between two slices is called a *backbone slice*, or backbone for short. A backbone is useful for finding out those parts of an application that contribute to the computation of several values.

As an example, consider the program shown in Example 7.2. The backbone of the two backward slices of `write(sum)` and `write(mul)` consists of those statements included in both slices—namely, `a = read()`, `b = read()`, `while (a <= b)`, and `a = a + 1`.

As the name suggests, backbones are central parts of the computation. In debugging, finding a backbone is most useful if one has multiple infected values at different places and wants to determine a possible common origin.

Dices. The *difference* between two slices is called a *dice.* A dice is useful for finding out how the backward slice of some variable differs from the backward slice of some other variable.

Again, consider the program shown in Example 7.2. If we subtract the backward slice of `write(sum)` from the backward slice of `write(mul)` (shown in the figure), all that remains is the initialization `mul = 1` and the assignment `mul = mul * a`.

Dices are most useful if one knows that a program is "largely correct"—that is, most of the values it computes are correct but some are not. By subtracting the backward slices of the correct variables from the backward slices of the infected variables one can focus on those statements that only contribute to the infected values—that is, those statements likely to cause the failure.

7.4.4 Leveraging Slices

Just like dependences, slices can be leveraged in programming environments, allowing programmers to explore slices and to explicitly ignore those parts of a program that are irrelevant for a specific behavior. As an example, consider the CODESURFER screenshot shown in Figure 7.5, showing the program from Example 7.2(a). The programmer has selected statement 13, `write(mul)`, as the slicing criterion and has chosen to view its backward slice. CODESURFER highlights all statements that are part of the backward slice. As in Example 7.2(b), it turns out that the computation of `sum` has no influence whatsoever on the computation of `mul`.

In addition to displaying slices, CODESURFER can perform slice operations (as discussed in Section 7.4.3). This allows the programmer to further focus on possible failure origins.

7.4.5 Executable Slices

Comparing the backward slice as determined by CODESURFER with the slice as shown in Example 7.2(b), you may notice a small difference: the CODESURFER slice also includes the declarations of `a`, `b`, and `mul`, whereas Example 7.2(b) does not. Could it be that CODESURFER determines dependences we do not know about?

The reason CODESURFER (Figure 7.5) includes the declarations in the slice is not that anything would depend on these declarations. (If something would, this would be an error, as discussed in Section 7.5.) CODESURFER includes these declarations because it attempts to make the slice *executable.* That is, the slice should be an "independent program guaranteed to represent faithfully the original program within the domain of the specified subset of behavior" (i.e., the state of the program as read by the slicing criterion). Because the program needs to be executable, all

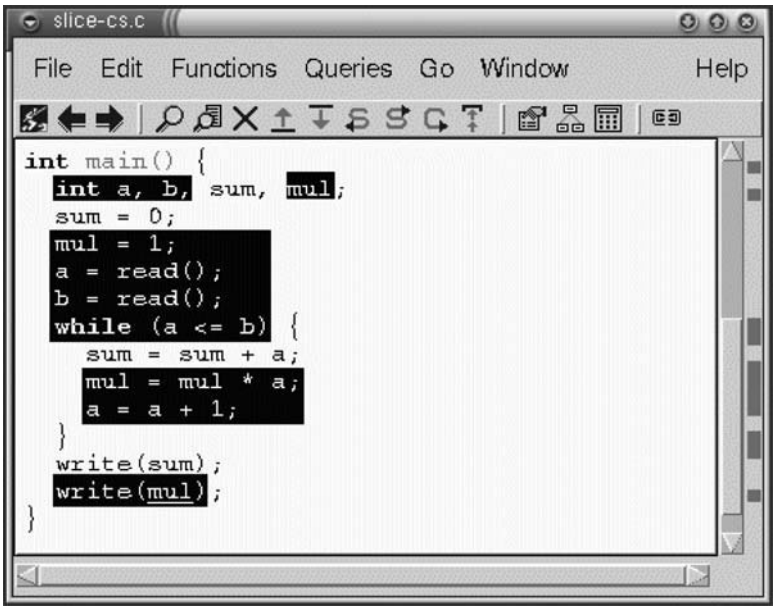

FIGURE 7.5

A program slice in CODESURFER. All statements that are part of the slice are highlighted.

variables have to be declared, regardless of whether the declarations are actually part of some dependence or not. In our examples, though, we do not require the slice to be executable. Thus, we omit declarations if they are not involved in any dependence.

7.5 DEDUCING CODE SMELLS

In Section 7.3.4 we saw how a data dependence from the uninitialized variable `f` caused the `fib()` program to fail. In general, we can assume that *any* read from an uninitialized variable is a bad idea, and we may thus easily qualify any such attempt as an error. In fact, a number of common errors can be directly detected from the dependence graph—that is, deduced from the program code alone. Examples include the following.

7.5.1 Reading Uninitialized Variables

Uninitialized variables, such as `f` in `fib()`, are a common source of errors. In terms of dependences, declarations such as `int f` should have no influence on any other statement. If they do, this should be considered an error.

Compilers routinely determine variable usage and dependences when they *optimize* the generated code. Thus, they can easily report if some variable appears to be used although not initialized:

```
$ gcc -Wall -O fibo.c
fibo.c: In function ‘fib’:
fibo.c:7: warning: ‘f’ might be used uninitialized
          in this function
$ _
```

(The `-O` option turns on optimization, and the `-Wall` option turns on almost all warnings.) As the wording suggests, the compiler may err in reporting a variable as being used. This is illustrated by the following example.

```
int go;
switch (color) {
    case RED:
    case AMBER:
        go = 0;
        break;
    case GREEN:
        go = 1;
        break;
}
if (go) { ... }
```

Here, `go` is initialized if `color` is one of `RED`, `AMBER`, or `GREEN`. If `color` has another value, though, `go` will remain uninitialized. The compiler is unable to determine automatically whether `color` may take another value. Nonetheless, the compiler emits a warning such that the programmer can take a look at the code.

7.5.2 Unused Values

If some variable is written, but never read, this is likely to be an error. In the dependence graph, a write into such a variable translates into a statement on which no other statement is data dependent—in other words, a statement without any effect.

Compilers (and derived program-analysis tools) can also warn against unused values. However, as there are many ways to access a variable that may go unnoticed by the compiler (including access of other modules to global variables, access via pointers, and so on) the feature is typically limited to local variables.

7.5.3 Unreachable Code

If some code is never executed, this is likely to be an error. In the dependence graph, this translates into a statement that is not control dependent on any other statement.

In many simple cases, compilers can warn against unreachable code. Consider the following example.

```
if (w >= 0)
    printf("w is non-negative\n");
else if (w > 0)
    printf("w is positive\n");
```

The second `printf()` will never be executed because its condition is subsumed by the first condition. Its execution is dependent on no other statement. The compiler, being smart enough to notice the subsumption, issues a warning such as:

```
$ gcc -Wunreachable-code -O noop.c
noop.c:4: warning: will never be executed
$ _
```

Why do we have to enable warnings about unreachable code explicitly? The reason is that during debugging programmers frequently insert statements to observe the behavior (see Chapter 8) or to check their expectations (see Chapter 10). Such statements may be written in such a way that executing them would show the presence of a failure. The following is an example.

```
switch (color) {
    case RED:
    case AMBER:
        go = 0;
        break;
    case GREEN:
        go = 1;
        break;
    default:
        printf("This can't happen\n");
        exit(1);
}
```

If the compiler reports that the `printf()` statement is unreachable, this is actually a good sign. (At the same time, the warning about `go` used before initialization should also go away.)

If one of the preceding conditions occurs, this typically is an error in the program in question. At the least it is a *code smell* that should be verified before the code goes into production.

So far, the code smells we have seen are all related to dependences concerning the usage of variables. In addition to these *general* dependences there are dependences that are specific to some language feature or runtime library—and again, such dependences can be leveraged to detect errors.

Memory leaks. In languages without garbage collection, such as C or C++, the programmer is responsible for deallocating dynamic memory. If the last reference to a chunk of dynamic memory is lost, the chunk can no longer be deallocated (a *memory leak* occurs). Example 7.3 shows a C function that has a memory leak at `return 0`. The reference `p` to the memory allocated in line 4 is lost.

EXAMPLE 7.3: A potential memory leak. On premature return, memory pointed to by `p` is not deallocated

```
 1  /* Allocate and read in SIZE integers */
 2  int *readbuf(int size)
 3  {
 4      int *p = malloc(size * sizeof(int));
 5      for (int i = 0; i < size; i++)
 6      {
 7          p[i] = readint();
 8          if (p[i] == 0)
 9              return 0;  // end-of-file
10      }
11
12      return p;
13  }
```

Just as we tracked the effects of statements on variables in Table 7.1, we can track the effects of statements on dynamic memory. For each statement, we check whether it allocates, uses, or deallocates a chunk. We also check whether the reference to the chunk is still *live*—that is, accessible by other statements—and we can identify statements where a reference becomes *lost*, such as overwriting an existing reference or returning from a function in which the reference was declared a local variable. If there is a path in the control flow graph from an allocation to a statement where a chunk becomes dead without going through a deallocation first, this is a memory leak. In Example 7.3, such a path goes from line 4 to line 9.

Interface misuse. In addition to memory, one can think of other resources that must be explicitly deallocated when they are no longer in use. As an example, consider *streams*. An input/output stream is first opened, but must be closed when it is no longer used. Checking for such conditions uses the same mechanisms as memory leaks. If there is a path in the control flow graph from a stream opening to a statement where the stream reference becomes dead without going through a closing first, this is an error. Similar techniques apply to resources such as locks, sockets, devices, and so on.

Null pointers. In the same style as memory leaks, we can check whether a pointer being null may be accidentally referenced. This happens if there is a path from a statement in which a null pointer `p` is being initialized to a statement in which `p` is dereferenced without going through some assignment to `p`.

In Example 7.3, for instance, the `malloc()` function may return a null pointer if no more memory is available. Consequently, in the expression `p[i]` the pointer `p` may be null, resulting in a potential runtime failure. Therefore, this error can be detected automatically. The code should be changed such that `malloc()` returning a null pointer ends in a user-friendly diagnosis.

Given a control flow graph, and basic data and control dependences, a tool that checks for such common errors is not too difficult to build. Some advanced compilers even have such built-in functionality. However, there are also external tools that are especially built for detecting code smells. As an example, consider the FINDBUGS tool for JAVA programs. FINDBUGS scans JAVA bytecode for *defect patterns*—that is, common programming errors (as those listed previously)—and highlights potential problems, as shown in Figure 7.6. Table 7.2 lists some of the most common defect patterns detected by FINDBUGS.

Tools such as FINDBUGS are highly useful in detecting code smells before they end up in production code. One should keep in mind, though, that these tools can report *false positives*—that is, they can report possible influences where indeed there are not. The FINDBUGS authors, for instance, list a false positive rate of 50 percent, meaning that only every second smell reported by FINDBUGS is indeed an error. Programmers are well advised, though, to rewrite even those smells that are not errors—simply because this way they will not shown up in the next diagnostic.

In general, whenever a failure occurs it is good practice to use a static checker such as FINDBUGS (or the compiler with all warnings enabled) to rule out common

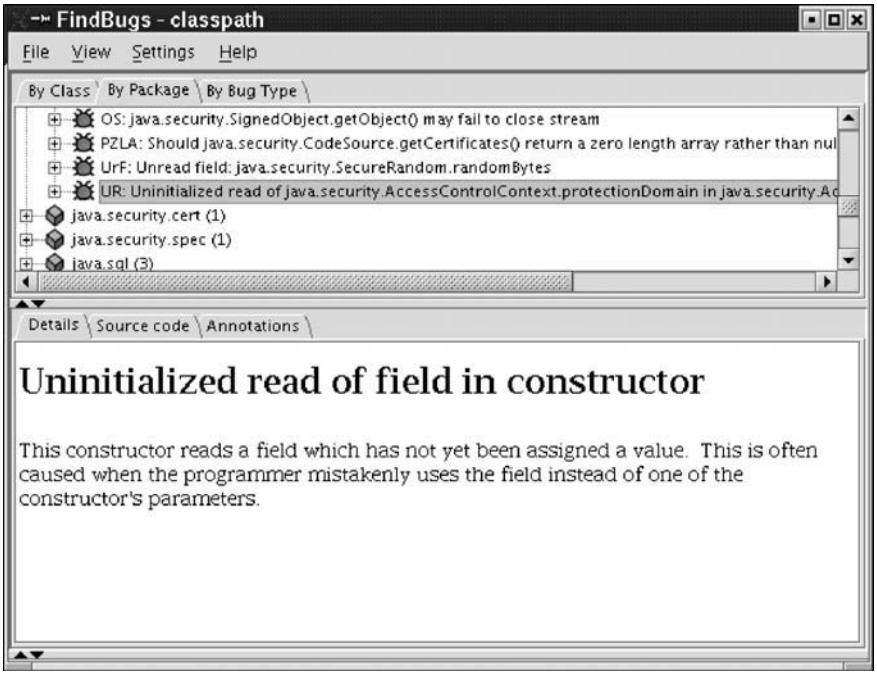

FIGURE 7.6

Detecting defect patterns with FINDBUGS.

Table 7.2 Some Defect Patterns Detected by FINDBUGS
Class implements `Cloneable` but does not define or use `clone` method
Method might ignore exception
Null pointer dereference in method
Class defines `equal()`; should it be `equals()`?
Method may fail to close database resource
Method may fail to close stream
Method ignores return value
Unread field
Unused field
Unwritten field
Private method is never called

defect patterns as a failure cause. In addition, if one has a concrete failure to deal with one can also apply checking tools that search for common issues in this particular run—such as memory issues or violation of invariants. These dynamic tools are discussed in Chapter 10.

7.6 LIMITS OF STATIC ANALYSIS

As discussed in Section 7.5, a tool such as FINDBUGS has a false positive rate of 50 percent. Why can't we rewrite tools such as FINDBUGS to have *no* false positives at all?

The reason for the imprecision of FINDBUGS (or CODESURFER, or any tool using static analysis) is that there are a number of language constructs that make computing precise data dependences difficult, if not impossible. For instance, given a statement *A* that writes `a[i]`, and a following statement *B* that reads `a[j]`, how can one know whether *A* may influence *B*? To answer this question requires finding out whether `i = j` can ever hold. Although it may be possible to answer a *specific* question such as this, in general such questions are instances of Turing's halting problem and thus undecidable.

Note that this limitation is not necessarily prone to machines. Humans suffer the very same problem. The following is an example showing the limits of static analysis. In the following piece of code, is `x` being used uninitialized or not?

```
int x;
for(i=j=k=1;--j||k;k=j?i%j?k:k-j:(j=i+=2));
write(x);
```

The answer is: nobody knows. The `for` loop terminates if and only if `i` holds an odd *perfect number*, an integer that is the sum of its proper positive divisors (28 is a perfect number because $28 = 1 + 2 + 4 + 7 + 14$). In that nobody knows today whether odd perfect numbers exist, it is unknown whether the `write(x)` statement will ever be executed—and neither humans nor machines have a straightforward way of deciding this.

In practice, though, we may well treat `write(x)` as unreachable. It is known that an odd perfect number must be greater than 10^{300}, and thus (assuming sufficiently long integers `i`, `j`, and `k`), we must prepare for at least 10^{600} loop iterations. In addition, `write(x)` is either unreachable or uses an uninitialized variable, and thus an error anyway—and this ambiguity is something a tool *could* determine.

As a consequence, static analysis tools must resort to *conservative approximations*—for instance, an approximation saying that *any* write to the array `a[]` may influence *any* read from the array `a[]`. Although this approximation may result in more data dependences than strictly necessary, it at least ensures that no data dependence is falsely omitted. List 7.2 outlines the most important language constructs that require approximations.

LIST 7.2: Data Flow Caveats

Indirect access. To determine that a write to a variable V influences a later read of this variable requires that V be precisely determined. If V is a location that is determined at runtime, however, one has to resort to *approximations*.

As a typical example, consider *array accesses*. If some statement writes into `a[i]`, one has to determine the possible values of `i` in order to track precise dependences. These possible values can be approximated by following back the dependences of `i` (assuming they all lead to constants) or by *symbolic evaluation* of `i`. (Humans, of course, may have better means of abstracting possible values of `i`.)

Pointers. Writing to a location referenced by a pointer or reference P requires that one know the locations P may point to or reference. This type of analysis is known as *points-to analysis*, which is a common part of optimizing compilers. A simple and common automated strategy is to assume that a pointer P may point to all objects the addresses of which have been taken in the program code (again, human reasoning is usually more precise).

Functions. Including function calls in the analysis results in dependences between the arguments at the call sites (the *context*) and the formal parameters at the function entries. If a function is called from multiple sites in a program, one can choose to inline the function body at each call site, resulting in precise dependences. This approach fails, though, for large numbers of functions. It is also infeasible for recursive functions.

A viable alternative is to introduce *summary edges* at call sites that represent the transitive dependences of the function in question. Such summaries also introduce imprecision.

More features. Other features that make computing dependences difficult include *object orientation* and *concurrency*.

The drawback of such approximations is that dependences are difficult to track. In the `sample` program discussed in Chapters 1 and 6, tracking the origins of `a[0]` stops short at the `shell_sort()` function, which writes into every element of `a[]`. Therefore, all one can deduce at this point is that the content of `a[]` was responsible for the value of `a[0]`. Consequently, any member of `a[]` may influence `a[0]`.

If one is really paranoid about conservative approximation, static analysis results in virtually no results for many real-life programs. If a stray pointer may access noninitialized memory, and if an array index may go out of bounds, anything can happen—meaning that conservative approximation returns "I don't know" for any program property. Likewise, if a function is unavailable to analysis (due to lack of source code, for instance), calling such a function also stops static analysis on the spot. Anything can happen after the function is done. However, even if there were

LIST 7.3: Source Code Caveats

Source mismatch. Whenever processing source code, one must make sure that the source code being read actually corresponds to the program being executed. For released programs, this means to use version control, as discussed in Section 2.9 in Chapter 2. For a local program, be sure not to confound the locations (Bug Story 8). Using incremental construction (using tools such as *make*), be sure that all compilations are up to date.

Macros and preprocessors. A *preprocessor* is a program that manipulates program text before it is being fed to the compiler. In Example 7.1, for instance, the `#include` statement in line 3 makes the C preprocessor insert the contents of the `stdio.h` header.

Preprocessors can be tricky because they may introduce uncommon behavior. For instance, a *macro definition* such as `#define int long` causes all subsequent `int` types to be read in as `long` types. Programmers typically make macros explicit to ease understanding. Modern programming languages avoid the usage of preprocessors.

Undefined behavior. Some programming languages deliberately do not specify the semantics of some constructs. Instead, each compiler can choose its own implementation. In C, for instance, the value range of a `char` is not defined. It may be anything from an 8-bit to a 128-bit value or even larger.

Issues with undefined behavior typically arise when a program is ported to a new environment or a new compiler. Being aware of undefined behavior helps to identify errors quickly.

Aspects. An *aspect* is a piece of code that is added (or *woven*) to specific parts of a program—for instance, an aspect that prints out "`set()` has been called" at the beginning of every `set()` method.

Aspects are great tools for logging and debugging. We will cover these uses in Section 8.2.3 in Chapter 8. However, as adding an aspect to a program can cause arbitrary changes in behavior, aspects can also seriously hamper our ability to understand the code.

BUG STORY 8

Stubborn Hello

In the beginning of my programming career, I was writing a simple program called `hello` that would output `Hello, world!` to the UNIX console. My program worked fine, but as I changed the text to `Bonjour, monde!` and compiled it, `hello` would still output `Hello, world!` Regardless of what I did, the text would remain fixed.

A friend then explained to me that by typing `hello` at the prompt, I was invoking the preinstalled GNU `hello` program instead of my own. He reported similar problems with a program of his own called `test`—conflicting with the built-in `test` command. I quickly learned to type `./test`, `./sort`, `./hello`, and so on to start my own programs from my directory.

no conservative approximation deduction brings a number of risks simply by being based on *abstraction*, including the following.

- *Risk of code mismatch.* Using source code to deduce facts in a concrete program run requires that the run actually be created from this very source code. Otherwise, bad mismatches can happen (see List 7.3).
- *Risk of abstracting away.* To actually execute the source code, one requires a compiler, an operating system, a runtime library, or other tools. When deducing from source code, one cannot take all of this "real world" into account. One assumes that the environment of the source code operates properly. More precisely, one assumes semantics of the program code that hold regardless of the environment. In rare instances, though, failures can be caused by a defect in the environment—and therefore, deducing an error from source code will be impossible.
- *Risk of imprecision.* In Figure 7.3 we saw that an ordinary program already has much data and many control dependences, such that any slice quickly encompasses large parts of the program. In the presence of data flow caveats (see List 7.2), slices become even larger, as conservative approximation is required to make sure no dependence is lost. On average, a static slice encompasses about 30 percent of the program code, which is a significant reduction but still a huge amount of code.

The risk of code mismatch can be easily taken care of by establishing precise configuration management. Abstracting away is a risk inherent to any type of pure deduction. The risk of imprecision, though, can be addressed by two mechanisms:

1. *Verification.* If one can constrain the possible program states, it is possible to increase the precision of deduction. As an example, consider the following code.

```
p = &y;
if (x > 0)
    y = x;
if (y > 0)
    p = &x;
```

Where can the pointer `p` point to after this code? It is trivial to prove that the condition `x > 0` and the assignment `y = x` imply `y > 0`. Thus, if we know that `x > 0` holds we can ensure that `p` points to `x`. Such constraints can be computed, accumulated, and resolved across the code, thereby increasing precision. In Chapter 10, we see how such conditions can be expressed as assertions and verified at runtime as well as deduced at compile time.

2. *Observation.* Rather than deducing facts from source code that hold for *all* runs, one can combine deduction with facts observed from concrete program runs—notably from the one run that fails. Not only does this give concrete findings about the failure in question, as a side effect observation also removes the risk of abstracting away.

7.7 CONCEPTS

How To

To isolate value origins, follow back the dependences from the statement in question (Section 7.3).

Dependences can uncover *code smells*—in particular common errors such as use of uninitialized variables, unused values, or unreachable code.

Before debugging, get rid of code smells reported by automated detection tools (such as the compiler).

To slice a program, follow dependences from a statement *S* to determine all statements that:

- Could be influenced by *S* (*forward slice*)
- Could influence *S* (*backward slice*)

Using deduction alone includes a number of risks, including the risk of code mismatch, the risk of abstracting away relevant details, and the risk of imprecision.

Any type of deduction is limited by the halting problem and must thus resort to conservative approximation.

7.8 TOOLS

CODESURFER. CODESURFER is considered among the most advanced static-analysis tools available. It is available free of charge to faculty members (if you are a student, ask your advisor). All others must purchase a license. CODESURFER is available at *http://www.codesurfer.com/*.

FINDBUGS. The FINDBUGS tool was developed by Hovemeyer and Pugh (2004). It is open source. Its project page is found at *http://findbugs.sourceforge.net/*.

7.9 FURTHER READING

Weiser (1982) was the first to discover that programmers mentally ignore statements that cannot have an influence on a statement in which an erroneous state is discovered. In this paper, Weiser also coined the term *program slicing*.

The original approach by Weiser (1984) was based on data flow equations. The same year, Ottenstein and Ottenstein (1984) introduced the notion of the program-dependence graph. Indeed, all later slicing approaches used a graph-based representation of program dependences.

Since these pioneer works, several researchers have extended the concept. Tip (1995) still summarizes today's state of the art in slicing. Regarding the usefulness of slices, Binkley and Harman (2003) examined slice sizes in 43 ℂ programs and found that the average slice size was under 30 percent of the original program.

Besides Hovemeyer and Pugh, several researchers have worked on using static analysis to detect defect patterns. I specifically recommend the work of Dawson Engler's group on analyzing the Linux kernel. Chelf (2004) gives a survey.

The basic techniques for analyzing source code—especially scanning, parsing, and detecting the effects of statements—are all part of compiler construction. As an introduction, I recommend Aho et al. (1986) as well as the series of *Modern Compiler Implementation* by Andrew Appel. Advanced readers may like to look at Muchnik (1997).

Christian Morgenstern's poem "The Impossible Fact" is taken from Morgenstern (1964).

EXERCISES

7.1 For the program shown in Example 7.2(a), write down:

(a) The control flow graph, as in Figure 7.1.
(b) The effects of the statements, as in Table 7.1.
(c) The control dependences, as in Figure 7.3.
(d) The data dependences, as in Figure 7.3.

7.2 Sketch a mechanism based on the control flow graph and dependences that ensures that after a call to `free(x)` the value `x` is no longer used.

7.3 For the defect patterns in Table 7.2, explain what type of program representation (call flow graph, data-dependence graph, source code) is needed to compute these smells.

7.4 Xie and Engler (2002) describe an analysis technique for catching defects in code. The idea is that *redundant operations* commonly flag correctness errors. Xie and Engler applied their technique on the source code of the Linux kernel and found errors such as the following:

- *Idempotent operations:* Such as when a variable is assigned to itself—for instance, in the following code, where the programmer makes a mistake while copying the structure `sa` to the structure `da`.

```
/* 2.4.1/net/appletalk/aarp.c:aarp_rcv() */
/* We need to make a copy of the entry. */
da.s_node = sa.s_node;
da.s_net = da.s_net;
```

- *Redundant assignments:* Where a value assigned to a variable is not subsequently used—such as the value assigned to `err` in the following code.

```
/* 2.4.1/net/decnet/af_decnet.c:dn_wait_run() */
do {
    ...
    if (signal_pending(current)) {
        err = -ERESTARTSYS;
        break;
    }
    SOCK_SLEEP_PRE(sk);
    if (scp->state != DN_RUN)
        schedule();
    SOCK_SLEEP_POST(sk);
} while (scp->state != DN_RUN);
return 0;
```

- *Dead code:* Which is never executed—such as the following code, where the insertion of a logging statement causes the function to always return (note the misleading indentation).

```
/* 2.4.1/drivers/net/arcnet/arc-rimi.c:arcrimi_found() */
/* reserve the irq */
if (request_irq(dev->irq, &arcnet_interrupt ...))
                BUGMSG(D_NORMAL,
                       "Can't get IRQ %d!\n", dev->irq);
    return -ENODEV;

⟨Following code is never executed⟩
```

(a) For each of the previous categories, sketch how dependences can be used to detect them.

(b) Are these defects still present in the current Linux kernel? When were they fixed?

7.5 What problems can you imagine that arise for users of deduction from code mismatch?

7.6 A *dice* can highlight those program statements computing an infected variable that cannot have an influence on a correct variable. How should a conservative approximation of indirect access look for a slice of a correct and a slice of an infected variable?

Palmström, old, an aimless rover,
walking in the wrong direction
at a busy intersection
is run over.

"How," he says, his life restoring
and with pluck his death ignoring,
"can an accident like this
ever happen? What's amiss?"

"Did the state administration
fail in motor transportation?
Did police ignore the need
for reducing driving speed?"

"Isn't there a prohibition,
barring motorized transmission
of the living to the dead?
Was the driver right who sped …?"

Tightly swathed in dampened tissues
he explores the legal issues,
and it soon is clear as air:
Cars were not permitted there!

And he comes to the conclusion:
His mishap was an illusion,
for, he reasons pointedly,
that which must not, can not be.

– Christian Morgenstern
The Impossible Fact (1905)