### **Control Flow**

### 6.7 Nondeterminacy

In Algol 68, the lack of ordering among expression operands was explicitly defined as an example of nondeterminacy, which the language designers called *collateral execution*. Several other built-in constructs in Algol 68 were nondeterministic as well, and an explicit *collateral statement* allowed the programmer to specify nondeterminacy in the evaluation of arbitrary expressions when desired.

Among his many contributions to the art of programming, Dijkstra [Dij75] advocated the use of nondeterminacy for selection and logically controlled loops. His *guarded command* notation has been adopted by several languages. One of these is SR, a pedagogical language of the 1980s, which we will mention again in Chapter 13. Imagine for a moment that we are writing a function to return the maximum of two integers. In C, we would probably employ a code fragment something like this:

if (a > b) max = a; else max = b;

Of course, we could also write

if (a >= b) max = a; else max = b;

These fragments differ in their behavior when a = b: the first sets max = b; the second sets max = a. As a practical matter the difference is irrelevant, since a and b are equal, but it is in some sense aesthetically unpleasant to have to make an arbitrary choice between the two. More important, the arbitrariness of the choice makes it more difficult to reason about the code formally, or to prove it is correct. In a language with guarded commands (the example here is in SR), one could write the following:

if a >= b -> max := a
[] b >= a -> max := b
fi

EXAMPLE 6.89

Avoiding asymmetry with nondeterminism

example 6.90

Selection with guarded commands

c-131

The general form of this construct is

```
if condition -> stmt_list
[] condition -> stmt_list
[] condition -> stmt_list
[] condition -> stmt_list
...
fi
```

Each of the conditions in this construct is known as a *guard*. The guard and its following statement, together, are called a *guarded command*. When control reaches an *if* statement in a language with guarded commands, a nondeterministic choice is made among the guards that evaluate to true, and the statement list following the chosen guard is executed. In SR, the final condition may optionally be else. If none of the conditions evaluates to true, the statement list following the else, if any, is executed. If there is no else, the *if* statement as a whole has no effect. (In Dijkstra's original proposal, there was no else guard option, and it was a dynamic semantic error for none of the guards to be true.) Interestingly, SR provides no separate case construct: the SR compiler detects when the conditions of an *if* statement test the same expression against a nonoverlapping set of compile-time constants, and generates table-lookup code as appropriate.

SR uses guarded commands for several purposes in addition to selection. Its logically controlled looping construct (again patterned on Dijkstra's proposal) looks very much like the if statement:

```
do condition -> stmt_list
[] condition -> stmt_list
[] condition -> stmt_list
[] condition -> stmt_list
...
od
```

For each iteration of the loop, a nondeterministic choice is made among the guards that evaluate to true, and the statement list following the chosen one is executed. The loop terminates when none of the guards is true (there is no else guard option for loops). Using this notation, we can write Euclid's greatest common divisor algorithm as follows:

```
do a > b -> a := a - b
[] b > a -> b := b - a
od
gcd := a
```

#### Nondeterministic Concurrency

While nondeterministic constructs have a certain appeal from an aesthetic and formal semantics point of view, their most compelling advantages arise in concurrent programs, for which they can affect correctness. Imagine, for example, that we are writing a simple dictionary program to support computer-aided design on a

example 6.91

Looping with guarded commands

example 6.92

Nondeterministic message receipt

```
process client:

loop

toss coin

if heads, send read request to server

wait for response

if tails, send write request to server

wait for response

process server:

loop

receive read request

reply with data

OR

receive write request

update data and reply
```

**Figure 6.7** Example of a concurrent program that requires nondeterminacy. The server must be able to accept either a read or a write request, whichever is available at the moment. If it insists on receiving them in any particular order, deadlock may result.

network of personal computers. The dictionary keeps a mapping from part names to their specifications. A dictionary server process handles requests from clients on other workstations on the network. Each request may be either a read (return me the current specification for part X) or a write (define part Y as follows).<sup>1</sup> Clients send requests at unpredictable times. As a result, the server cannot tell at any given time whether it should try to receive a read or a write request. If it makes the wrong choice the entire system may deadlock (see Figure C-6.7).

Most message-based concurrent languages provide at least one mechanism to specify nondeterministic choice among potential communication partners. These mechanisms do not all look like guarded commands, but they have similar semantics. In SR, one could write our dictionary server as follows:

Nondeterministic server in SR

EXAMPLE 6.93

```
# declarations of request types:
op read_data(n : name) returns d : description
op write_data(n : name; d : description)
# local subroutines:
proc lookup ... # find info in dictionary
proc update ... # change info in dictionary
```

I This is of course an oversimplified example. Among other things, any real system of this sort would need a mechanism to lock parts in the dictionary, so that no two clients would ever end up designing new specifications for the same part concurrently.

Here in is a nondeterministic construct whose guards can contain communication statements. The guard write\_data(n, d) will evaluate to true if and only if some client is attempting to send a request containing a new specification for a part. We shall see in Section C-13.5.3 that more elaborate guards can allow a server to constrain the types of requests that it is willing to receive at a given point in time, or even to "peek" inside a message to see if it is acceptable. If none of the guards of an in statement is true, the server waits until one is.

#### Choosing among Guards

What happens if two or more guards evaluate to true? How does the language implementation choose among them? We have glossed over this issue so far. The most naive implementation would treat a guarded command construct like a conventional if...then...else:

```
server:
loop
if read_data request available
...
elsif write_data request available
...
else wait until some request is available
```

The problem with this implementation is that it always favors one type of request over another; if read\_data requests are always available, write\_data requests will never be received.

A slightly more sophisticated implementation would maintain a circular list of the guards in each set of guarded commands. Each time it encounters the construct in which these commands appear, it would check guards beginning with the one after the one that succeeded last time. This technique works well in many cases, but can fail consistently in others. In the following, for example (again in SR), the guard of the first in statement combines a communication test with a Boolean condition:

EXAMPLE 6.94

Naive (unfair) implementation of nondeterminism

#### example 6.95

"Gotcha" in round-robin implementation of nondeterminism

```
process silly
var count : int := 0
    do true ->
        in A() st count % 2 = 1 -> ...
        [] B() -> ...
        [] C() -> ...
        ni
        count++
        od
```

This example is somewhat contrived, but illustrates the problem. The st ("such that") clause in the first guard indicates that it can be chosen only on odd iterations of the loop. Now imagine that A, B, and C requests are always available. If we always check guards starting with the one after the one that succeeded last time (beginning at first with the initial guard), then B will be chosen in the first iteration (because count mod  $2 \neq 1$ ), C will be chosen in the second iteration (when count = 2), B will be chosen again in the third iteration (because again count mod  $2 \neq 1$ ), and so forth. A will never be chosen. The lesson to be learned from this example is that no deterministic algorithm will provide a truly satisfactory implementation of a nondeterministic construct (see Sidebar C-6.10).

One final issue has to do with side effects. Guarded command constructs make a nondeterministic choice among the guards that evaluate to true. They do *not*, however, guarantee that all guards will be evaluated before the choice is made; the implementation is free to ignore the rest of the guards once it has chosen one that is true. A program may therefore produce unexpected or even unpredictable

#### **DESIGN & IMPLEMENTATION**

#### 6.10 Nondeterminacy and fairness

Ideally, what we should like in a nondeterministic construct is a guarantee of fairness. This turns out to be trickier than one might expect: there are several plausible ways that "fair" might be defined. Certainly we should like to guarantee that no guard that is always true is always skipped. Probably, we should like to guarantee that no guard that is true infinitely often (in a hypothetical infinite sequence of iterations) is always skipped. Better, we might ask that any guard that is true infinitely often be chosen infinitely often. This stronger notion of fairness will obtain if the choice among true guards is genuinely random. Unfortunately, good pseudorandom number generators are expensive enough that we may not want to use them to choose among guards. As a result, most implementations of guarded commands are not provably fair. Many simply employ the circular list technique. Others use somewhat "more random" heuristics. Many machines, for example, provide a fast-running clock register that can be read efficiently in user-level code. A reasonable "random" choice of the guard to evaluate first can be made by interpreting this clock as an integer, and computing its remainder modulo the number of guards.

#### c-136 Chapter 6 Control Flow

results if any of the guards have side effects. This problem is the programmer's responsibility in SR. An alternative would have been to prohibit side effects and have the compiler verify their absence.

#### CHECK YOUR UNDERSTANDING

**45**. What is a *guarded command*?

- **46**. Explain why nondeterminacy is particularly important for concurrent programs.
- 47. Give three alternative definitions of *fairness* in the context of nondeterminacy.
- **48**. Describe three possible ways of implementing the choice among guards that evaluate to true. What are the tradeoffs among these?

# **Control Flow**

### 6.9 Exercises

6.38 Explain why the following guarded commands in SR are *not* equivalent:

if a < b -> c := a	if a < b -> c := a
[] b < c -> c := b	[] b < c -> c := b
[] else -> c := d	[] true -> c := d
fi	fi

- **6.39** The astute reader may have noticed that the final line of the code in Example C-6.91 embodies an arbitrary choice. It could just as easily have said gcd := b. Show how to use a guarded command to restore the symmetry of the program.
- 6.40 Write, in SR or pseudocode, a function that returns
  - (a) an arbitrary nonzero element of a given array
  - (b) an arbitrary permutation of a given array

In each case, write your code in such a way that if the implementation of nondeterminism were truly random, all correct answers would be equally likely.

## **Control Flow**

# 6.10 Explorations

- **6.48** Learn about the select routine in the Unix (POSIX) library. How does it deal with the need for nondeterministic receipt from multiple communication partners? How would you use this routine to achieve the effect of the SR code in Example C-6.93?
- 6.49 Explain how to use threads in Java to achieve the effect of Example C-6.93.