Chapter 6

Dataflow

There is some evidence that the next big wave of change to wash over programming languages will be concurrency. Both architectures and languages for concurrency have been around for some time. In this chapter, I will discuss the dataflow architectural concept and the languages that have been designed to conform to it. Dataflow is one way to achieve concurrency, particularly at the fine-grain level: It finds multiple operations that can be undertaken concurrently within the evaluation of a single expression. Ideas from dataflow have found their way into parallelizing compilers for more conventional architectures, as well. The ideas here in some ways prepare for Chapter 7, which deals with concurrent programming languages that work at a coarser level of granularity.

Sequential execution is an essential characteristic of the von Neumann computer architecture, in which programs and data are stored in a central memory. The concepts embodied by classical architecture have not been directly applicable to the domain of parallel computation. Most programming languages have evolved from von Neumann languages, designed specifically for the von Neumann architecture, so programmers have been conditioned to analyze problems and write programs in sequential fashion.

The dataflow approach was first suggested by Karp and Miller [Karp 66] as an alternative to the von Neumann architectural and linguistic concepts. Consider computation of the series of statements in Figure 6.1.

Figure 6.1

```
A := B*C + D/F;
G := H**2 + A;
```

A data-dependency graph called a **dataflow graph** represents the ordering of evaluation imposed by data dependencies. It encodes the fact that an expression can't be evaluated before its operands are evaluated. The dataflow graph for Figure 6.1 appears in Figure 6.2.

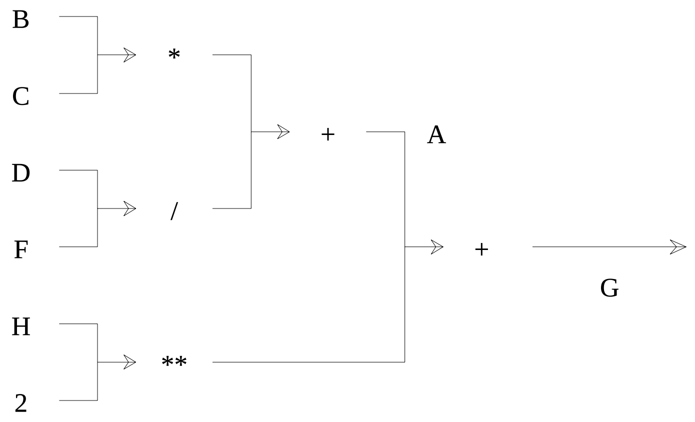

Figure 6.2 Dataflow graph

This graph represents a partial order on the evaluation sequence. In contrast, a typical von Neumann language will create a totally ordered instruction sequence to evaluate these statements, but such an order loses a significant amount of potential concurrency. If more than one set of operands is available, more than one expression should be evaluated concurrently.

In a dataflow computer, a program isn't represented by a linear instruction sequence, but by a dataflow graph. Moreover, no single thread of control moves from instruction to instruction demanding data, operating on it, and producing new data. Rather, data flows to instructions, causing evaluation to occur as soon as all operands are available. Data is sent along the arcs of the dataflow graph in the form of **tokens**, which are created by computational nodes and placed on output arcs. They are removed from the arcs when they are accessed as input by other computational nodes. Concurrent execution is a natural result of the fact that many tokens can be on the dataflow graph at any time; the only constraint on evaluation order is the presence of tokens on arcs in the graph.

Most computational nodes in a dataflow graph compute arithmetic results. However, some sort of conditional structure is necessary. Loops are accommodated in dataflow graphs by introducing nodes called **valves** that control the flow of tokens within the graph. Two kinds of valves are commonly built: distributors and selectors. A distributor takes an input token and a Boolean control token. It distributes the input token to one of two output arcs (labeled T and F), as shown graphically in Figure 6.3.

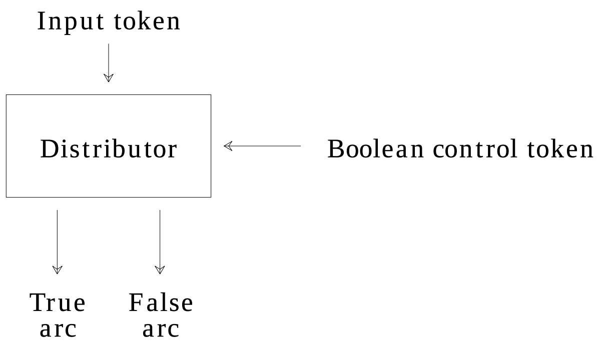

Figure 6.3 Distributor node

A selector uses a Boolean control token to accept one of two input tokens and passes the selected token onto its output arc, as shown in Figure 6.4.

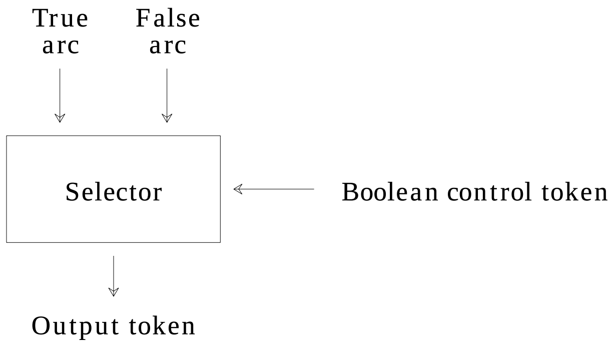

Figure 6.4 Selector node

Viewing a computation as a dataflow graph leads directly to a functional view of programming. Dataflow graphs do not include the notion of variables, since there are no named memory cells holding values. Computation does not produce side effects. Functional programming languages lend themselves to various evaluation orders. Dataflow evaluators are typically speculative evaluators, since they are data-driven. Computations are triggered not by a demand for values or data, but rather by their availability.

However, there are some important differences between dataflow and functional programming. First, dataflow graphs have no simple concept of a function that returns a value. One could surround part of a dataflow graph with a boundary and call it a function, where all inbound arcs to the region would be the parameters and all outgoing arcs would be the results. Such an organization could lead to recursively defined dataflow graphs. Second, as you saw in Chapter 4, functional programming languages rely heavily on recursion, because they do not support iteration. However, dataflow manages to support recursion by building cyclic graphs; an initial token may be placed inside a cycle in order to allow the first iteration to proceed. Third, the values placed on arcs are all simple values. It is easy to understand integers, reals, and even Boolean values moving along the arcs, but values of structured types such as records and arrays might take more underlying machinery. Pointer types are most likely out of the question. Most important, function types, which are so useful in functional programming, are not supported by the dataflow graph formalism.

1 ◆ DATAFLOW COMPUTERS

There are two classes of dataflow architectures. The first class is called "static," since such architectures do not support reentrant dataflow code (that is, code that is used simultaneously in multiple places in the dataflow graph) and recursion. The simple dataflow computer model introduced above is static, as is the machine proposed by Dennis [Dennis 77].

The second class is called "dynamic." Such machines support simultaneous multiple incarnations of an activity, recursion, and loop unfolding. In a dynamic dataflow architecture, an arc may carry multiple tokens, and care is taken to ensure that activities fire only upon receipt of matching tokens along their input arcs. Tokens are labeled to distinguish values arising in different contexts or from different incarnations of an activity. Two tokens match only if their activity labels match. For example, I might wish to perform the computation specified by a dataflow graph on each element of a vector of 1000 values. With a dynamic dataflow architecture, I can place 1000 tokens on the input arc. Tokens are labeled to insure that values produced during the computation can be ascribed to the appropriate input value.

The dynamic architecture outlined below is essentially that of the classical Manchester Dataflow Computer [Gurd 85]. Modern dynamic dataflow architectures may look different. Tokens are labeled, but I leave out the specific details of how the labels are generated or used, except that labels are matched to identify matching tokens.

The architecture is schematically depicted in Figure 6.5. The machine operates as a circular pipeline divided into four sections. The processing unit receives packets containing operands and an instruction. The instruction is executed on the accompanying operands, and the result tokens (after appropriate labeling) are placed back on the bus to be sent to the I/O switch. The I/O Switch is included in the pipeline to serve as an I/O port. The matching unit consists of associative token storage. When a token arrives at the matching unit, the storage is searched for any tokens with the same label and destination. If matches are discovered, these tokens are read out of the store, and a packet is formed of all these tokens to be sent on to the program store. The destination field of a token carries the address in the program store of the instruction to which the token is directed.

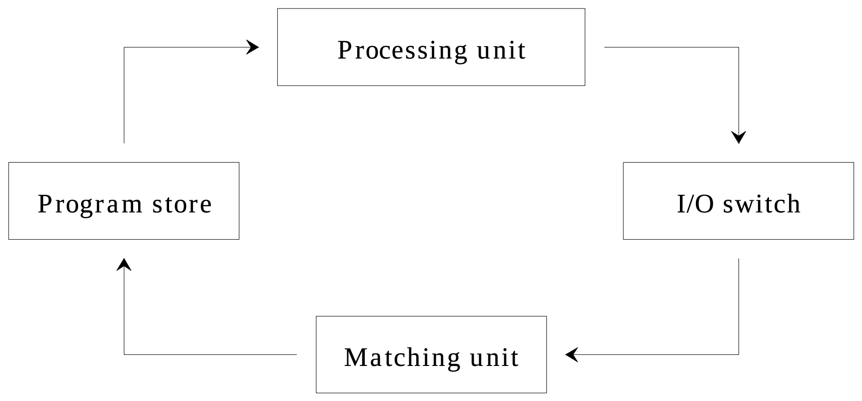

Figure 6.5 Dataflow architecture

If no match for the arriving token is found in the matching store, the token is written into storage and held in abeyance for matching tokens expected to arrive in the future. For unary operators, no matching is needed, and the token can bypass the matching unit and proceed directly to the program store. It is estimated that 55–70 percent of all tokens are able to bypass the matching section.

When a packet of tokens reaches the program store, the instruction to which the tokens are directed is accessed, and a packet is formed of the tokens and the instruction. This instruction is an opcode, the operands being already available in the packet. Opcodes are elementary operations like addition and multiplication. The program store effectively holds the dataflow graph. Instructions are vertices, and the arcs are represented by link fields associated with each instruction to indicate the successor nodes in the dataflow graph. The packet consisting of the operands, the opcode, and successor information is then sent to the processing unit, and the operation of the machine continues as described.

2 ◆ VAL

Val is a dataflow language developed at MIT by J. Dennis and others [McGraw 82]. This language was originally designed for the static dataflow architecture of Dennis, so it does not support dynamic dataflow features like recursion. Val has been meticulously defined both by an axiomatic [Ackerman 80] and a denotational [Gehani 80] semantics (these concepts are discussed in Chapter 10.) Val is functional in nature, so side effects are absent. However, Val is strongly typed (using structural equivalence) and intentionally looks more like conventional languages than LISP or FP does. In fact, it looks much like ML (described in Chapter 3). For example, the Val function in Figure 6.6 computes the mean and standard deviation for parameters X, Y, and Z.

Figure 6.6

```
function stats(X, Y, Z: real) : real, real;
    let
        Mean : real := (X+Y+Z)/3;
        SD : real := sqrt((X - Mean)**2 +
            (Y - Mean)**2 + (Z - Mean)**2)/3;
    in
        Mean, SD
    end
end
```

In Val, functions return tuples of values. This syntax simplifies composition of functions, as the values returned by one function can be immediately used as parameters to a calling function.

Val has the usual scalar types (integer, real, Boolean, and character) as well as arrays, records, and choice types. Arrays are all flexible, with only the index type declared. A variety of array operations such as concatenation, extension, and contraction are provided. During array construction, all elements of an array may be specified simultaneously, allowing all the evaluations of array entries to proceed concurrently. Record construction is patterned after array construction to promote concurrency. Val defines error values, discussed in Chapter 2, so that the result of all computations is well defined, no matter what the evaluation order.

Val appears to have an assignment statement, but this appearance is misleading; like ML, identifiers can be bound in a block to values, but the binding cannot be changed within that block. Such bindings get confusing when an iterative loop is employed, since the program may need to update values between iterations. Val views each iteration as creating a new name-value binding, with values from the previous iteration included in the computation of the new loop values.

Structured values are viewed as single values, so array and record components can never be individually modified. If you want to change an array element, you must create a new array differing from the original in only one element.

Val provides implicit concurrency. Operations that can execute independently are evident (once the dataflow graph is built by the compiler) without needing any explicit notation. Val achieves implicit concurrency by using functional language features, and makes use of the fact that evaluation of a function or expression has no side effects. If two operations do not depend on the outcomes of each other, they can execute simultaneously. A source of side effects in conventional languages is **aliasing**, whereby the same memory cell can be referenced by more than one name. Reference-mode parameters, pointers, and overlays can create aliases. All aliasing is forbidden in Val. The fact that Val relies on implicit concurrency is justified on the grounds that concurrency can be at a very low level (at the level, say, of individual arithmetical operations), and it is unreasonable to expect the programmer to specify concurrency details at this level.

In addition to implicit concurrency, Val provides explicit concurrency in the **forall** expression, which concurrently evaluates an expression for all values in a range or structure. The width of parallelism is specified as a control identifier that assumes all values in a given range. In addition to specifying the control identifier and introducing a new name scope, loops also specify how to merge the values from all the parallel streams into one result. There are two ways to generate results: **construct**, which allows each parallel execution path to generate a value that becomes an element of an array of results, and **accumulate**, in which values from all the result streams are merged into one result using one of a fixed set of associative binary operators like + . (The operation specified by **accumulate** can be computed by an implicit balanced binary tree, allowing the merged value to be produced in logarithmic time.) Figure 6.7 clarifies these notions.

Figure 6.7

```
forall i in [1, 100] do                                          1
     left: real     := point[i].x_low;                           2
     bottom: real   := point[i].y_low;                           3
     right: real    := point[i].x_high;                          4
     top: real      := point[i].y_high;                          5
     area: real     := (right-left) * (top-bottom);              6
     okay: Boolean  := acceptable(area);                         7
     abort: Boolean := erroneous(area);                          8

     accumulate + if okay then area else 0.0 end;                9
     accumulate or abort;                                        10
     construct if okay then area else 0.0 end;                   11
end                                                              12
```

In this program, the **forall** produces 100 concurrent streams of execution. Their results are merged using **accumulate** + to add all acceptable areas (line 9), **accumulate or** to determine if abort was true in any of the streams (line 10), and **construct** to create an array of elements calculated by a formula (line 11). The entire **forall** expression returns a 3-tuple comprising those three results.

The **for** expression implements loops that cannot execute in parallel because values produced in one iteration must be used in the next. The decision concerning whether to continue loop iteration occurs within the loop body as a conditional expression, as in Figure 6.8.

Figure 6.8

```
for                                                         1
     a: real := 0.0;                                        2
     b: real := 1.0;                                        3
do                                                          4
     let                                                    5
          c: real, done: Boolean := Compute(a, b);          6
     in                                                     7
          if done then                                      8
               c                                            9
          else                                              10
               iter                                         11
                    a := NewA(a, b);                        12
                    b := NewB(a, b);                        13
               end                                          14
          end                                               15
     end                                                    16
end                                                         17
```

The identifiers a and b (lines 2 and 3) are loop parameters. During the first iteration of the loop, the parameters have values 0.0 and 1.0. The Compute invocation (line 6) returns two values, which are bound to c and done. If **iter** is selected (lines 11–14), a new iteration is begun. New values for a and b are evaluated for the next iteration. The binding in line 13 uses the old value of a on the right-hand side. When done is true, the expression returns the value bound to c (line 9).

A choice type is built as shown in Figure 6.9.

Figure 6.9

```
type list =                                                 1
     choice [                                               2
          empty : void;                                     3
          nonempty : record [item : real; rest : list]      4
     ]                                                      5
```

Void (line 3) is a predeclared type with no values. This example defines list as an ordinary linked list, but with an interesting difference — it is recursively defined without using pointers. Val disallows pointers because of aliasing issues. A list is therefore a recursive data structure rather than a sequence of individual elements linked with pointers.

A value of a choice type is created by using a **make** constructor (Figure 6.10).

Figure 6.10

```
make list[empty : nil]
```

To guarantee type compatibility, the contents of a **choice** can only be accessed via **tagcase**, as in Figure 6.11.

Figure 6.11

```
function IsEmpty(L : list) : Boolean;
     tagcase L of
          when tag empty => true
          when nonempty => false
     end
end
```

Val also has conditional expressions.

Because Val lacks side effects and provides error values, it is an ideal candidate for speculative evaluation. In some places, however, Val inhibits speculative evaluation to reduce unneeded computations. In particular, in **if**, **tagcase** and **for** expressions, computations are not initiated until the controlling expression is computed and tested. Val uses a lazy evaluator for these constructs. In contrast, components of ordinary expressions are assumed to be data-driven, implying a speculative evaluator. Parameters to functions are always fully evaluated before the function is invoked.

Evaluation order in Val is primarily dictated by efficiency and simplicity concerns, allowing lazy, speculative, and strict evaluation to coexist. An interesting problem arises if a computation fails to return any value (even an error value), because it diverges (that is, loops forever). A lazy evaluator avoids a diverging subcomputation if its result isn't needed. A speculative evaluator tries to compute everything, and unnecessary diverging subcomputations proceed concurrently with other computations. A conventional evaluator does not proceed beyond the diverging subcomputation. Thus, Val's evaluation rules can affect the results computed by otherwise equivalent constructs. For example, an **if** expression can't be replaced by a call to an equivalent function, because **if** evaluates only some of its components, while a function evaluates all its parameters.

Val programs obey many of the laws of FP (Chapter 4), so many of the FP theorems can be used to transform (and optimize) Val programs. Val differs from FP in the way it handles error values, which can invalidate certain FP theorems. FP functions are bottom-preserving, and ⊥ represents "error." Val, on the other hand, allows error values to be detected, and further computation can repair or ignore the error and produce an ordinary value. For example, I might like to establish that in Val line 1 of Figure 6.12 is equivalent to line 2.

Figure 6.12

```
H(if p(...) then F(...) else G(...) end)                 1
if p(...) then H(F(...)) else H(G(...)) end              2
```

That is, that I can distribute a call of `H` into both arms of a conditional. This theorem is true in bottom-preserving FP environments, but it isn't true in Val, because `H(error_value)` need not be `error_value`.

3 ◆ SISAL

Val was one of the first serious attempts to produce a production-quality dataflow language. A descendent of Val called Sisal was created to exploit the capabilities of dynamic dataflow computers [McGraw 83]. A Sisal compiler exists, with code generators for Vax, Crays, HEP multiprocessors, and the Manchester dataflow computer.

The most obvious advantage of Sisal over Val is its support of recursion. Recursive functions are useful and natural, especially in functional languages. Val's rejection of recursion was a reflection of the design of early static dataflow machines. Sisal also supports streams, which are needed for ordinary sequential I/O and as a means of composing functions.

Sisal programs can be decomposed into distinct compilation units that explicitly import and export functions. Sisal also extends Val's iterative and parallel (**`forall`**) loop forms. They can return arrays or streams. Parallel loops can also define explicit inner and outer products, making array manipulation cleaner and potentially more efficient.

4 ◆ POST

Chinya V. Ravishankar developed Post as a Ph.D. thesis starting in 1981 [Ravishankar 89]. Post introduces several novel ideas. First, it lets the programmer determine the level of speculation in evaluation. As I mentioned earlier, speculative evaluation can lead to nontermination under certain circumstances, but strictly lazy evaluation reduces parallelism. A second novel concept is **polychronous data structures** that are partly **synchronous** (must be available before used) and partly **asynchronous** (parts can be used when ready). Third, Post provides communication between computational activities in order to terminate speculative computation that may turn out to be unnecessary. Communication between computations is not natural in purely functional programming languages. Much of their semantic elegance derives from their lack of side effects, so computations scheduled in parallel must not depend on each other's results. Further, a purely functional language permits only deterministic computations and prohibits history-sensitivity.

Post was never fully implemented, but a prototype compiler was built. It first builds a dataflow graph from the program and then converts that graph into instructions for a dynamic dataflow machine. Post needs a dataflow machine (never implemented) that has a few special features, such as a "hold" node for implementing lazy evaluation and a "terminate store" to help find and remove terminated tokens.

4.1 Data Types

Values can be either primitive (`integer`, `real`, `Boolean`, `char`) or structured. Structured values are constructed using the abstractions **stream** and **tuple**. Both are sequences of values, but a tuple may be heterogeneous and is of fixed length, while a stream is homogeneous and of unbounded length. All operators and functions are automatically overloaded to apply to streams; they create streams of results of pointwise application. Nested structures are allowed; an element of a stream may itself be a stream.

After a finite set of values, streams continue with an infinite number of **eos** (end of stream) values. The **eos** value may be used as an operand in arithmetic, logical, and comparison operations. It acts as an identity with arithmetic and logical operations. With comparison operations, **eos** may be made to behave either as the minimal or maximal element by using different comparison operations. Post has two predeclared functions that generate streams of values: (1) `stream(a,b)` generates a stream of integers ranging from `a` to `b`, followed by an infinite number of **eos** values; and (2) `const(a)` generates an infinite stream of integers with value `a`.

Values are implemented by the target dataflow architecture as tokens containing a name (identifying the arc in the dataflow graph), a value (typically, a real number), and a label. The label is unexpectedly complicated, containing fields describing scope, index, and program-defined information. Each of these three fields is a stack containing information pertaining to different dynamic scopes in the program. The scope field identifies those dynamic scopes. The index field distinguishes elements of a stream. The program-defined field permits the programmer to tag values so that computations tagged in a particular way can be terminated; this facility allows Post to manage speculative computation, as described below.

4.2 Programs

A program consists of a name, a parameter list (used as a pattern), and a target expression. The pattern is matched against input data and serves to bind formal parameters to values in the input. The target expression generates a value that is returned by the program. The target expression may introduce a new name scope, with type definitions and identifier declarations. Post is statically scoped. Figure 6.13 shows a simple program.

Figure 6.13

```
function AddNumbers{a,b,c};            1
     type a, b, c : int                2
     in a+b+c                          3
end;                                   4
```

The structure is much like a classical procedure declaration, with formal parameters (the pattern `{a,b,c}` in line 1), type declarations for the formal parameters (line 2), and a body (line 3).

The type declarations are separated from the pattern for clarity, because the pattern can become complex. For example, `{x,y,z,}*` is a pattern that repeats the template `{x,y,z}` indefinitely, matching three more values of the input stream each time. On the other hand, `{x,{y,}*,z}` is a 3-tuple with a stream as the second component. Figure 6.14 uses a pattern that matches a stream.

Figure 6.14

```
function AddPairs{x,y,}*;              1
     type x, y : int                   2
     in x+y                            3
end;                                   4
```

This program outputs a stream consisting of the sums of adjacent values in the input stream; the order of the output stream matches the order of the input stream. If the input stream consists of an odd number of elements, y in the final instance of the template matches an **eos** token, which is an identity for arithmetic operations.

4.3 Synchrony Control

By using connectors other than a comma, the programmer may specify the degree of synchrony required in pattern matching. The comma indicates completely asynchronous matching; the actual parameters may be accessed independently of each other. If the pattern uses only ˆ , matching is synchronous: all elements must be present before any element is made available to the target expression.[1] If the pattern uses only ˜ , matching is sequential: the *i*th element of the matched data structure is available only after all previous elements have arrived (even if they have not been accessed). Any or all of these combinators may occur within a pattern. If several of them occur, access is polychronous; precedence rules indicate how to group subpatterns in the absence of parentheses.

In Figure 6.14, I could change the input pattern to {xˆy,}*, forcing pairwise synchronization of input elements. The program would compute the same results, because + requires both operands before proceeding.

Sequential patterns are used for loops, as shown in the Figure 6.15.

Figure 6.15

```
function Largest{x˜}* init 0;                                   1
     type x : int                                               2
     in if Largest > x then Largest else x end                  3
end;                                                            4
```

This program finds the largest value in a stream of integers. It terminates when the last instance of the target expression terminates; the value generated in this last instance is returned as the value of the program. The program name is used to name the current value, initialized to 0 in line 1 and compared with the next value in line 3. The dataflow graph corresponding to this program has a cycle holding an initial token with value 0. Conditional expressions, like the one in line 3, are evaluated speculatively. The result from the branch that is not needed is discarded after evaluation is complete.[2]

An alternative syntax for conditionals is shown in Figure 6.16.

Figure 6.16

```
function OddSquares{x,}*;                                       1
     type x : int                                               2
     in [(x mod 2) ≠ 0] x*x                                     3
end;
```

Line 3 evaluates to x*x only if the condition in square brackets is true, that

[1] The concept of synchronization in concurrent programming languages is related; it is discussed in Chapter 7.

[2] If conditionals were evaluated lazily, the programmer could hoist both branches out of the conditional to force them to be evaluated speculatively.

is, when x is odd. For other instances, the expression evaluates to nil. All nil values are removed from the resulting stream before it is returned.

4.4 Guardians

Guardians implement shared data. They look like procedures and act like variables. In a sense, they are like objects in an object-oriented programming language (discussed in Chapter 5). Assignment to the guardian invokes the procedure with the given value as an actual parameter. The procedure computes a stored value, which can differ from the value of the actual parameter. When the program accesses the guardian's value, it gets a copy of the current stored value. Such access is lazy, in the sense that it occurs only after all nonguardian values used in the expression have arrived. The guardian only has one instance, which prevents simultaneous computations from attempting simultaneous assignment. For example, consider the program of Figure 6.17.

Figure 6.17

```
function LargestFactor{x,}*;                                    1
     type x : int                                               2
     guardian Largest{v} init 0;                                3
          type v : int                                          4
          in if Largest < v then v else Largest end             5
     end -- Largest                                             6
     in Largest := if (N mod x) = 0 then x end                  7
end; -- LargestFactor                                           8

LargestFactor(stream(2,99));                                    9
```

This program finds the largest factor of N (a nonlocal variable) smaller than 100. Even though a new instance of LargestFactor is created for each element of the input stream, there is only one instance of its local guardian, Largest (lines 3–6). Each element of the input stream is tested by an instance of line 7; the result is assigned into the guardian. The conditional expression in line 7 evaluates to nil if the Boolean is false; nil values are filtered out and are not passed to the guardian. Each assignment invokes Largest's target expression (line 5). This expression computes a new value for the guardian, namely, the largest value so far assigned to it. The value of a program containing guardians is a tuple of the guardians' final values; in this case, there is only one guardian, so a single value is returned.

Because guardians may have different values if they are evaluated at different times, it is necessary to permit lazy evaluation of actual parameters that are expressions involving guardians. Post allows parameters to be passed in value mode (the default) or lazy value mode.

4.5 Speculative Computation

Speculative computation can be terminated by a combination of program-defined labels and an explicit **terminate** statement. Any expression may be labeled, and the value resulting from that expression carries the label until it exits the scope in which the label is declared. An individual value may carry many labels, since it may be composed of many components, each of which may acquire multiple labels in different computations. Figure 6.18 is an example of labeling.

Figure 6.18

```
function ReportEvens{x,}*;                                 1
     type x : int;                                          2
     label Even;                                            3
     function AddLabel{y};                                  4
          type y : int                                      5
          in                                                6
               if (y mod 2) = 0 then                        7
                    tag y with Even                         8
               else                                         9
                    y                                       10
               end;                                         11
     end; -- AddLabel                                       12
     in [ AddLabel(x) haslabel Even ] x;                    13
end; -- ReportEvens                                         14
```

This program returns only the even elements of the input stream. AddLabel labels its parameter if the parameter is even (line 8). The body of ReportEvens in line 12 checks to see if its parameter, passed through AddLabel, is labeled. If so, the parameter is returned; otherwise, the value nil is returned (and then removed from the resulting stream).

If the program chooses to delete all tokens that have a particular label, any computation they are involved in is thereby terminated. Figure 6.19 demonstrates termination.

Figure 6.19

```
function SomeFactor{x,}*;                                  1
     type x : int;                                          2
     guardian Factor{y};                                    3
          type y : int                                      4
          in terminate y                                    5
     end; -- Factor                                         6
     in Factor := if (N mod x)=0 then x end;                7
end; -- SomeFactor                                          8

SomeFactor(stream(2,99));                                   9
```

The program returns the first factor of N assigned to the guardian Factor; it may return different values for different runs. For each input value, line 7 reports a factor, if any, to the guardian. The guardian Factor executes a **terminate** statement and returns the value y (line 5). The **terminate** statement does not specify a label in this case; all computation in the current scope (which is implicitly given a scope label) is canceled. There is also a syntax (not shown in this example) for specifying a program-defined label to control termination.

5 ◆ FINAL COMMENTS

Val and Sisal look, at first glance, like ordinary imperative languages. What makes them dataflow languages is that they are functional, so that speculative evaluation is possible, and they provide for explicitly concurrent loop executions.

Post was developed in reaction to this imperative appearance. It tries to give the programmer a feeling of labeled tokens being routed on arcs. The **terminate** statement only makes sense in such a context, for example. Although Post is a worthy attempt to mirror dataflow architectures better than Val or Sisal, the result is not particularly readable. Lack of clarity, in the sense introduced in Chapter 1, is its major weakness.

In a sense, all these languages are failures, because dataflow computing never became popular. Very few dataflow computers were ever built, and interest in this field has mostly subsided. Still, it is instructive to see how architectural design and programming language design influence each other. Not only did dataflow architecture lead to new languages, but those languages dictated enhancements to the architecture (such as multiple-field label stacks on tokens). A similar interplay is now taking place between architecture and languages as massively parallel and distributed computers are becoming available. That is the subject of Chapter 7.

Dataflow has had some successes. Optimizing compilers for vector machines build dataflow graphs in order to schedule computations effectively. The graphs indicate what dependencies constrain the order of evaluation.

From one point of view, you could say that dataflow has been quite successful and is widely used. Spreadsheets incorporate a form of data-driven computation to update values that depend on other values that may have changed. The internal representation of a spreadsheet is very like a dataflow graph. Strangely, the languages used in spreadsheet programming are quite different from any of the languages described here. First, they are not linear; that is, they are not organized as a text with a start, an ordered set of commands, and an end. Instead, each cell of a spreadsheet (typically, a two-dimensional grid of cells) is separately "programmed." For a cell acting as a leaf in the dataflow graph, the program indicates the value of that cell. For a cell acting as a computation node, the program indicates how to recompute the value of that cell based on the values of other cells. These other cells can be named explicitly, but accumulation operators such as summation and averaging can also specify a set of cells (generally, a contiguous one-dimensional subset). This organization is reminiscent of declarative programming (the subject of Chapter 8), in which there is no necessary order to the pieces that together make up a program.

A second difference in spreadsheet programming is that the user can often control to what extent computation is speculative. This control is specified as the number of times to reevaluate each computational cell when one of its inputs changes. Zero means do not update; an infinite value means to reevaluate until values do not change. In other words, the binding of evaluation strategies, which is usually done at language-design time, and only occasionally at compile time, can be deferred until runtime.

EXERCISES

Review Exercises

6.1 Draw a dataflow graph for the code of Figure 6.20.

Figure 6.20

```
(A + B) * (A + B + C)
```

6.2 Draw a dataflow graph for the code of Figure 6.21.

Figure 6.21

```
A := 0;
while A < 10 do
     A := A + 3;
end;
```

6.3 How do nondataflow languages allow the programmer to specify evaluation strategy?

Challenge Exercises

6.4 Draw a dataflow graph for the code of Figure 6.22.

Figure 6.22

```
procedure Orbit(A : integer) : integer;
begin
     if A = 1 then
          return 1;
     elsif even(A) then
          return Orbit(A/2);
     else
          return Orbit(3*A+1);
     end;
end;
```

6.5 In Figure 6.15 (page 180), which version of > is meant on line 3? That is, does **eos** act as the minimal or maximal element?

6.6 In Figure 6.18 (page 182), what would be the effect of changing line 13 as follows?

```
     in [ AddLabel(x) haslabel Even] AddLabel(x)
```

6.7 Modify Figure 6.19 (page 182) so that speculative computation is not terminated, but the first factor found is still returned.

6.8 In spreadsheets, how can reevaluating more than once have a different effect from evaluating only once?