3

Assignment and communication

There are two essentially different ways of communicating information from one part of a program to another. Just as in conventional programming languages, a process may modify the values of variables which are subsequently read by another process executed in sequence with the first. A process may also send values over channels to other processes executing in parallel with it.

Variables and assignment

A variable is a name with a specific type, associated with a value that can be changed by the program. At any time, the value of a variable is the last value assigned to it by the program, so communication through variables is essentially sequential.

A variable is introduced by a declaration consisting of its type, and its name, for example

```
INT Exe, Wye :
```

which could also be written as two separate declarations

```
INT Exe :
INT Wye :
```

declares two `INT` variables called *Exe*, and *Wye*.

```
[3][3][3]REAL64 tensor :
```

declares an `[3][3][3]REAL64` variable called *tensor*, or you could think of it as declaring twenty-seven component variables called *tensor*[0][0][0], *tensor*[0][0][1], *tensor*[0][0][2], *tensor*[0][1][0], and so on. Segments of an array of variables can also be used as arrays of variables. If *row* is an `[100]INT` variable, then `[row FROM 50 FOR 10]` is a `[10]INT` variable which consists of the ten variables *row*[50], *row*[51], to *row*[59].

Variables can be renamed by abbreviations, for example, if *Windscale* is an `INT` variable

```
INT Sellafield IS Windscale :
```

makes *Sellafield* a name for the same variable. Assignment to *Sellafield* changes the value of *Windscale*, but it is not permitted to use the name of *Windscale* in the scope of the declaration of *Sellafield*. It is a good idea to quote the type of a variable abbreviation, but it can be omitted in the same way as for value abbreviations.

Abbreviation of variables is most commonly used to give a short or efficient name for a component or segment of an array, for example

```
REAL64 middle IS tensor[1][1][1] :
```

Having made such an abbreviation, an implementation can be expected to be able to refer more efficiently to *middle* than to *tensor*[1][1][1], but the abbreviation is valid only if the selection is valid. Abbreviating a component of an array like this makes the name of the whole array inaccessible, except that the name can be used in other abbreviations which refer to other parts of the array. This concession means that it is possible to make disjoint abbreviations, for example in the scope of

```
left  IS [row FROM 0 FOR n] :
right IS [row FROM n FOR (SIZE row) - n] :
```

it is possible to refer to *left* and *right*, but not to *row*.

In the scope of its declaration, the name of a variable can be used in expressions, and its value is the last value that was assigned to it. That means, amongst other things, that a variable has no defined value until the first time that a value is assigned. The rules governing the use of variables in valid parallel programs mean that an expression cannot need to use the value of a variable while another process is able to change that value.

Just as in conventional languages, the values of variables can be changed by *assignment*. After the execution of

```
variable := expression
```

the value of the variable is the value which the expression had before its execution. For an assignment to be valid, the variable has to have the same type as the expression, and the expression has to be valid, as does any expression (for example, an array index) which occurs in the variable.

More generally, a multiple assignment

```
v1, v2, v3, ..., vn := e1, e2, e3, ..., en
```

is valid only if there are as many variables as expressions, and corresponding pairs have the same types. The effect is to assign to each variable in the list the value before all of the assignments of the corresponding expression. This means that, for example, you can write

```
yin, yang := yang, yin
```

to exchange the values of two variables *yin* and *yang*.

You may be wondering about assignments like

```
i, a[i] := p, q
```

where it is not plain whether the value of i is changed before or after it is used to decide which component of a is selected. In fact, in occam, such an assignment would be invalid. There is no implication that any part of the assignment executes from left to right: the expressions on the right may be evaluated in any order; the values of the variable on the left may be changed in any order. A multiple assignment behaves as though it were

```
SEQ
  PAR
    t1 := e1
    ...
    tn := en
  PAR
    v1 := t1
    ...
    vn := tn
```

where $t1$ up to tn are new variables of the right types.

It is invalid for any variable assigned to by an assignment to appear in an expression selecting any of the variables; it is also invalid for the same variable to be assigned to twice by an assignment,

```
x, x, x := hay, bee, sea
```

or as would be the case in

```
a[i], a[j] := a[j], a[i]
```

were i to be equal to j.

Communication by channels

In parallel programs, the analogue of the variable is the *channel*. You can think of a channel as being something like a wire between the processes; indeed, if two processes running on different processors in a multiprocessor computer communicate over a channel, then there will be a real physical wire implementing the channel. On the other hand, you should not think of occam communication as necessarily involving signals moving into or out of a computer. In occam, input and output are component processes just like assignment. An output process

```
channel ! expression
```

which sends the value of the expression over the channel, executed in parallel with a corresponding input process

```
channel ? variable
```

which reads a value from the channel and assigns it to the variable, is exactly the same as the assignment

```
variable := expression
```

In suitable circumstances – when both parties to a communication are implemented on a single processor – an input and an output reduce to just the corresponding assignment, and some synchronization to ensure that both parts of the communication happen at the same time.

A channel is used for communication in only one direction: if two concurrent processes (branches of a PAR) use a single channel, one of them must use it exclusively for input, the other exclusively for output. The synchronization of input and output is an important part of the communication. An input is not complete until the corresponding output is complete, and vice versa, so each partner in a communication has to wait for the other to be ready before the communication is complete.

Protocols and channel declarations

The equivalent, for channels, of the type of a variable is the *protocol* operated on that channel. Channels are declared, just like variables, by quoting a description when introducing the name. A protocol describes the values that are to be communicated, in the same way that the type of a variable constrains the values that can be stored.

The simplest protocols are just types; any type can be used as a simple protocol, and to declare a channel with such a protocol, say

```
CHAN OF REAL64 control :
```

is to say that it will only be used for outputs of a single value of that type,

```
control ! 1.0(REAL64) / (REAL64 ROUND 15)
```

and inputs to a single variable of that type,

```
[3][3][3]REAL64 tensor :
...
control ? tensor[0][0][0]
```

Of course, a single value of an array type is a whole array, so

```
CHAN OF [34]BYTE stream :
```

would be used for outputs of rows of thirty-four bytes, like

```
stream ! "pleated bagpipes without the pipes"
```

and inputs of rows of thirty-four bytes, like

```
[34]BYTE accordion :
stream ? accordion
```

This example illustrates another case where the precise sizing of arrays can be annoying, and there is a protocol for counted arrays which is like not quoting the length of an array when specifying an abbreviation.

Because it is important to be able to determine the sizes of arrays, in order to confirm that accesses to them are valid, a counted array consists of an integer describing the size of the array accompanied by that number of components of the array. Channels of counted array protocols are declared by quoting the type of the integer, and specifying the array type without giving an exact length

```
CHAN OF INT::[][3]INT vectors :
```

Outputs on such a channel have to specify the number of components being sent, so

```
vectors ! 2::[[1,1,1],[1,2,3],[1,4,9],[1,8,27]]
```

will send $[[1,1,1],[1,2,3]]$. Inputs receive the length and a number of components, so

```
INT d :
[100][3]INT buff :
...
vectors ? d::buff
```

together with the above output would have the effect of setting d to 2, and setting

```
[buff FROM 0 FOR d] :=
        [[[1,1,1],[1,2,3],[1,4,9],[1,8,27]] FROM 0 FOR 2]
```

Of course, neither the input nor the output is valid if the count which is sent or received is greater than the number of components in the array of which a segment is sent, or the array to a segment of which an assignment is made.

Counted array protocols are often used with strings of characters, and it is quite usual to use

```
c ! SIZE v :: v
```

for moving strings around.

Protocols can be given names, for example

```
PROTOCOL BIT IS BOOL :
PROTOCOL STRING IS INT::[]BYTE :
```

and in the scope of such a declaration the name can be used in a channel declaration

```
CHAN OF STRING file :
```

to have exactly the same meaning as declaring the channel with the corresponding protocol

```
CHAN OF INT::[]BYTE file :
```

In fact, for more complicated protocols you have to give a name to the protocol and use that name in declaring channels.

Sequential protocols

A sequential protocol is a sequence of (possibly different) simple protocols – types and counted arrays – and a channel declared to have a sequential protocol can be used for communicating sequences of values, each value conforming to the corresponding component of the protocol. In the scope of

```
PROTOCOL COMPLEX IS REAL64; REAL64 :
```

a channel declared by

```
CHAN OF COMPLEX complex :
```

can only be used for outputs of pairs of `REAL64`s

```
complex ! r * sin.th; r * cos.th
```

and inputs to pairs of `REAL64` variables

```
complex ? x; y
```

In this case there is a defined left-to-right sequence, so it is perfectly in order to use a channel with a sequential protocol

```
PROTOCOL RECORD IS INT; INT::[]BYTE :
CHAN OF RECORD update :
```

for inputs like

```
update ? i; length[i]::data[i]
```

The input to i happens first; then, using the new value of i variables are selected for the input of the counted array.

There is no significance to our use of an upper-case name for the protocol, except that this is a convention which we will adopt to make protocol names look like type names.

Discriminated protocols

In general, the communications between a pair of processes will at different times conform to different protocols. In the same way that in some programming languages you can store values of different types in a variable, and keep track of which type of value is stored, so in occam 2 values of different types can be sent over a channel by indicating with *tags* which type of value is to come. The declaration

```
PROTOCOL NUMBER
  CASE
    whole; INT64
    floating; REAL64
    complex; REAL64; REAL64
:
```

introduces a protocol called `NUMBER`, but it also declares tags called *whole*, *floating*, and *complex* with the same scope. The protocols which appear in the component lists of a discriminated protocol have to be simple protocols – types, and counted arrays.

If a channel is declared to use such a protocol,

```
CHAN OF NUMBER c :
```

the only outputs which are legal have the forms

```
c ! whole; n
```

where n is an `INT64`,

```
c ! floating; r
```

where r is a REAL64, or

```
c ! complex; r; theta
```

where r and *theta* are both REAL64s.

The corresponding discriminated input has the form of a case discrimination on the tag of the communicated value,

```
c ? CASE
  whole; n
    wurble.int64(n)
  floating; r
    wurble.real64(r)
  complex; r; theta
    wurble.complex64(r, theta)
```

This process accepts a tag, and depending on which of the tags is received completes the corresponding input. It then proceeds to execute the process that appears in that branch of the case discrimination.

Notice that the first thing to appear in the semicolon-separated list in each branch of a discriminated input is the tag. It is a constant, not a variable into which the tag is read! There is no way of accepting a tag into a variable and then testing the value of the tag: for what type would the variable have; what would be the number and types of the other variables in the input process?

In fact, the discriminated input is very much more like an alternative process

```
CHAN OF INT64 c.whole :
CHAN OF REAL64 c.floating :
CHAN OF COMPLEX c.complex :
...
ALT
  c.whole ? n
    wurble.int64(n)
  c.floating ? r
    wurble.real64(r)
  c.complex ? r; theta
    wurble.complex64(r, theta)
```

over a number of new channels with different protocols.

If some of the tags are not listed in a discriminated input, it is an error for that input to match with an output of one of the missing tags, and the whole input stops. A discriminated input with only one branch can be abbreviated to one line, for example

```
c ? CASE complex; r; theta
```

which is the same as

```
c ? CASE
  complex; r; theta
    SKIP
```

and it is of course wrong for the received tag to be different from the one tag in such an input. In that case, the input fails to terminate, like `STOP`.

Arrays of channels

Just as arrays of variables of the same type can be declared, so can arrays of channels with the same protocol, and arrays of arrays of channels, and so on. The need for arrays of channels arises naturally when one process needs to communicate with an array of processes.

Individual components and complete segments can be selected from arrays of channels in exactly the same way as from arrays of variables, and channels and arrays of channels can be abbreviated.

```
[100]CHAN OF NUMBER c :
[]CHAN OF NUMBER left   IS [c FROM 0 FOR n] :
CHAN OF NUMBER middle   IS c[n] :
[]CHAN OF NUMBER right  IS [c FROM n+1 FOR (SIZE c) - (n+1)] :
```

The same rules as for arrays of variables apply to the use of names of arrays of channels that have been abbreviated.