PROGRAM CORRECTNESS PROOFS

C.A.R. Hoare

Rapporteurs: Dr. T. Anderson
Mr. R.B. Gimson
Mr. R.M. Simpson

Introduction

This short series of three connected lectures, on the topic of parallel programming, tries to illustrate how a certain approach to the problems of organising programs to run in a parallel environment can be used to evade (rather than solve) most of the fascinating problems involved. I also want to illustrate how this approach has had an influence on my ability to organise non-parallel programs. As Dijkstra has pointed out, when we know how to solve the problems of parallel programming, perhaps we will be able to write sequential programs much better.

So in fact, the history of the development of these ideas is exactly the reverse of the order in which I will present them to you. I shall start with a treatment of sequential programming and proceed in my last lecture to deal with parallel programming. In fact it was the parallel programming idea that I came to first, and then gradually worked backwards to apply the same concepts, perhaps even more fruitfully, to sequential programs.

Lecture 1 Abstraction and Representation of Data Structures

My first talk is on the representation of data; and I would like to pick up a remark made by Professor Hopcroft yesterday on how it is possible to clarify one's thinking about complex programming problems by separating out questions of the algorithm being used to solve the problem from questions of how to represent the data involved. I'll assume, for the time being, that we have used a top-down approach, and that we know what sort of data we need in the abstract domain. Our task now is to code the representation of the data on to our machine.

Our purpose will be to enable the programmer to separate the algorithmic aspects of his program from those of data representation, and to develop and prove these aspects independently. The method that we will adopt is to allow the programmer (conceptually or in practice) to extend the range of types available in his programming language, by coding representation of new types.

The idea of extending the range of types in a programming language is now a familiar one. It has been included in Algol 68 and in Pascal, for example, but neither of these languages have exactly the right mechanism or structure for what I want. The reason why these languages do not fully satisfy me is because they fail to recognise what is, I think, the essential aspect of a type. These languages recognise a type as denumerable or finite collection of values of that type, but fail to associate a type with the set of primitive operations which are defined over that type. In order to say what we mean by the Boolean type (to take the simplest example) it is not enough to say that it just consists of the two values 'true' and 'false'. You should say that it consists of these values together with (say) the operations 'and', 'or' and 'not'. It is the association of the undifferentiated carriers of values together with the structure imposed by a set of operations that expresses the essence of a type. Those of you familiar with algebra will immediately recognise this as being an abstract algebra.

The first language to recognise this essential aspect of a type was SIMULA 67, which uses the word class for a programmer-defined type, to differentiate it from the built-in types real and integer of a programming language.

Example: Sets of Integers

I shall work almost wholly by examples. Suppose we wish to represent sets of integers, and that we have written some algorithm, possibly some combinatorial algorithm, which operates on sets of integers and uses certain operations to manipulate them. First we need to be able to declare variables of this new type or class, so I introduce the word 'integerset' to stand for this new type or class. For variable declarations I have used the notation of Pascal, or indeed mathematics:

```
s1, s2: integerset;
```

which declares variables of name s1 and s2 to be of type integerset. This sort of declaration will appear in the abstract program in the

same way as declarations of real or integer variables will appear in a concrete program. Now obviously we need to initialise these variables to some value, and a very common initial value for sets is the empty set, so we initialise s1, s2 to empty:

```
s1, s2:= { };
```

These sets will start empty, but will gradually be filled up by inserting numbers into these sets. This is done by forming the union of the current value of s1 with the unit set of some integer i and assigning the result back to s1:

```
s1:= s1 ∪ {i};
```

which in normal programming parlance we would read as 'insert i into s1'. We also need an operation of removal, to remove i from a set s1; i.e., take the set difference between s1 and the unitset i, and assign the value back to s1 again.

```
s1:= s1 - {i};
```

Finally we need a function which tests whether an integer i is a member of a set s1:

```
i ∈ s1
```

One of the important things about practical programming, particularly when operating on large values like arrays, sets and multi-length integers, for example, is that it is more efficient to use a two-address operation and update these values by partial updating operations in situ, rather than to implement generalised set union or difference as functions. Indeed to perform the operation s1:= s2 ∪ s3 is quite a major computational task and involves considerable problems of storage allocation, among other things. Provided we confine ourselves to updating in situ, we can usually obtain a significant boost of efficiency, sometimes one or even two orders of magnitude. For this reason I will sometimes use an alternative notation for selective updating in which I put the

operator directly against a colon to mean "apply this operation to the left-hand side, and the right-hand side and assign the result back to the left-hand side variable"; for example:

s1: ∪ {i}; s2:- {i};

Having written an abstract program, we often find that we don't need every operation on sets, but for this particular program we need a subset of them. It is part of the skill of the programmer to choose representations which are going to be suitable and efficient for particular applications. Why do we need to ask the programmer to program this particular type? Why didn't we build the concept of an integerset into the language to begin with? The answer is purely a matter of engineering feasibility. There are so many different ways of representing sets that there is no one way that you could build into a programming language that would be uniformly satisfactory. Of course, if the ranges of integers are small, say ranging between 1 and 32, we all know there is a good representation of these sets as bit patterns, and a few languages like PASCAL have built-in facilities to deal with sets of such small integers. But in general, and certainly in the problem we are going to deal with, the range of integers is as large as the range of integers on the machine, which is considerably larger than the number of bits you would be willing to allocate to any one of the sets.

So the choice between different representations of sets must, I think, be left to the programmer, at least for the foreseeable future, because he must choose a representation which depends on knowledge that only he has.

Example: Flight Booking

I will make my example even more specific and choose an example, not of a combinatorial problem, but a problem of aircraft flight booking.

A flight booking program maintains for each flight the set of ticket numbers of passengers booked on that flight. The capacity of the planes places an upper limit C on the size of each set. Therefore the sort of representation that we may be able to get away with is considerably simpler than the representation most general-purpose set manipulation packages would offer us. We do not need a great deal of sophisticated chaining because the capacity of the planes places a rather strict upper limit on the size of each set that we are interested in. Taking advantage of this we will represent the set as an integer array:

```
A: array [1..C ] of integer;
```

together with a variable

```
size: integer;
```

indicating the current size of the set.

What do we do next? I have described to you in ordinary words the representation I am going to use, and I think perhaps all of you are saying "Ah yes, that is fairly simple. I can see what is going on. He is now going to draw a picture so that we can see what it looks like." Well I have drawn a picture here (see Figure 1), because a picture is what you would expect to see at this stage in the program documentation, is it not?

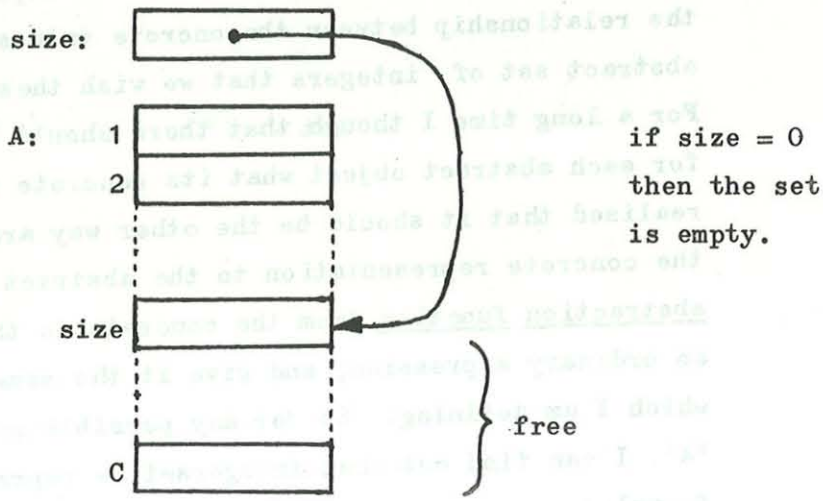


Figure 1

We have drawn the array with subscripts going between 1 and C, and 'size' points to the last used element of the array, the rest of the array being free. Then we realise the picture is not quite good enough so we add the note that if the size is zero then the set is empty.

Now I like pictures, especially nice simple ones like that, but they have their limitations. I would like to suggest to you that we ought to supplement our use of pictures by the rigorous logical formulation of assertions, using logical and mathematical notations to specify exactly the information that is intuitively conveyed by that picture.

Having shown how you can supplement this picture by rigorous assertions, I would like to persuade you to regard them as paramount. It is the assertions which really specify what the representation is; and the picture is just a useful confirmation of our understanding of the assertions, rather than the other way round.

Assertions

The first thing that we need in order to make precise the decisions that we have taken about the representation is to specify the relationship between theconcrete values of 'A' and 'size' and the abstract set of integers that we wish these values to stand for. For a long time I thought that there should be a mapping which specified for each abstract object what its concrete representation was. Then I realised that it should be the other way around; we need a mapping from the concrete representation to the abstract object. I will specify an abstraction function from the concrete to the abstract in the form of an ordinary expression, and give it the same name as that of the class which I am defining. So for any possible pair of values of 'size' and 'A', I can find out what integerset is represented by computing the formula:

$$\text{integerset} = \{i \mid \exists\, j : 1..\text{size},\ A[j]=i\}$$

Of course, I will never actually compute this function; though its arguments are concrete objects, which I can indeed compute on, the result is an abstract object which never exists within the store of the computer.

Notice that it is a many-one function. There are many different values of 'A' which mean the same integerset. There is a great deal of redundancy in this representation. In particular, the values of elements of 'A' whose subscripts are greater than 'size' are totally redundant; whatever their value, it does not change the set denoted by the representation. Furthermore, the ordering of the elements inside the array is not relevant, and any two elements in the subscript range 1....size could be swapped and it would still be the same set.

We also need to know certain facts about the concrete representation which remain true throughout the lifetime of the object; information which we need in order to preserve the validity of the representation. Before carrying out any operations on the concrete representation it must satisfy certain constraints, which I call <u>invariants</u>. These invariants will be made true when the value of the variable is initialised, and will remain true before and after every operation on the variable. Every operation on the integerset may assume the invariant is true beforehand, but in return must guarantee that it is true afterwards; so it is invariant in the sense that it is true between any two operations on this representation, but not necessarily in the middle of an operation.

The invariant in this case has to state that the value of variable 'size' remains within the range 1..C, otherwise the abstraction function becomes undefined. Also we must state that there are no repetitions within the first part of the array, in other words that the same value does not occur twice. So the invariant is defined as:

$$size = \#integerset \ \& \ size \leq C.$$

I have used the # sign to stand for the size (cardinality) of the set. I maintain that these two assertions contain all the information contained in the picture, and perhaps more.

Professor Randell "The invariant has to be true in order that the abstraction function is defined, but in defining the invariant you have used the abstraction function because you are talking about cardinality. Isn't there a circularity there?"

Professor Hoare "In practice there is no need to be quite so precise since these things won't have to be evaluated."

Dr. Burstall "Isn't the second part of that invariant, 'size $\leq$ C' a restriction on the domain of the abstraction function, and the first part, 'size= #integerset', merely a property of the abstraction function?"

Professor Hoare "Yes, the second part is a restriction, but the first part is not a property of the abstraction function."

Dr. Burstall "The way you have defined the set it must always be true."

Professor Dijkstra "You could have circumvented it by saying $0 \leq size \leq C$ and $\forall i,j:1..C(i \neq j \Rightarrow A[i] \neq A[j])$"

Professor Hoare "Yes, thank you. In fact 'size= # integerset' can be expressed:

$$\forall i,j:1..C\ (A[i] = A[j] \Rightarrow i=j).$$

There is no reason why you can't quote the integerset inside the invariant as far as I know."

Coding the Representation

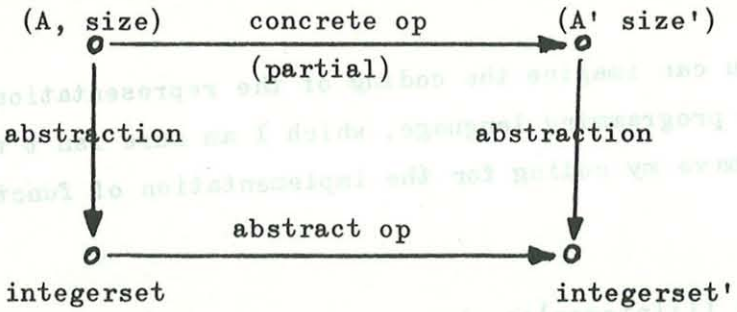
Now we have taken practically all the major decisions that need to be taken. At this stage we have made our informal ideas about the reprentation precise. In the implementation of the representation we need to write pieces of code operating on the concrete variables 'A' and 'size', which will have the desired effect on the value of the abstract integerset variable. So corresponding to the operations 'is a member of', 'insert', 'delete' and 'initialise', we will need to specify code to carry out these

operations. In fact we must write code to carry out on the concrete variables 'A' and 'size' the "same" operation that the abstract program carries out on an abstract integerset variable. In other words to code a concrete operation such that it maintains a form of commutivity with the abstract function:

abstraction (concrete op (A, size)) $\sqsubseteq$ abstract op (abstraction (A, size))

Since concrete operations are usually not quite as powerful as abstract operations, for such reasons as arithmetic overflow, the best that we can achieve is a weakened form of equality, $\sqsubseteq$. If both sides of $\sqsubseteq$ are defined they will indeed be equal, but in certain cirmcumstances the concrete operation on the left may not be defined, whereas the abstract operation is.

This commutivity can be shown in a diagram as in Figure 2, where dashed names refer to the values of the items after the operation 'op' has been carried out.



where 'op' stands for 'insert(i)', 'remove(i)' etc.

Figure 2

Dr. Milner "It seems to me that one would wish to specify abstracting those elements of the abstract domain on which the thing 'works', and then say that given this condition the equivalence is strict."

Professor Hoare "That is perfectly true. The question of how often your representation or program terminates is a very vital question which can hardly be ignored at any stage of the design. However, if you will be satisfied for the moment with partial correctness, we can postpone that problem until we find a case where the program doesn't work when we want it to."

Dr. Milner "It could be that the way one deals with it later is that one somehow proves that the program always terminates, but this separates the proof of termination from the thing it arises from which is the fact that the representation is not fully defined where it should be. It might be nicer if the fully defined-ness of the representation were made explicit at that point, rather than later."
Professor Hoare "I tend to say, without very much evidence to support it, that it does pay to separate the two aspects of the proof. There is a meaning of 'formal correctness' which says that at least if the program terminates correctly it has given the right result.
Dr. P. Lauer "Can I give you some evidence against that? If you prove the correctness of division by subtraction, as Floyd does, where you are using a loop for which you don't have to presuppose that the divisor is greater than zero, because now it terminates unsuccessfully. So the reduction of the correctness of that algorithm to a question of termination is only possible if you have a program with the possibility of indefinitely looping. If you haven't one with that possibility there is a problem."

I am sure you can imagine the coding of the representation written in your favourite programming language, which I am sure isn't the same as mine. Here I have my coding for the implementation of function 'has':

```
function has (i:integer):boolean;
assert has ≡ i ∈ integerset;
begin has:= false;
      for j:1..size while ¬ has do
          if A[j]=i then has:= true

end;
```

It is important when writing this function to annotate it, to say what it really means. What it means is that its parameter is a member of the integerset. Now if you replace the 'integerset' by its definition

(the abstraction function given earlier) you will get a very clear statement of what must be true at the end of its loop, namely

has ≡ ∃j:1..size, A[j]=i .

I think most of you could correctly form a loop to do this. I suspect that even an automatic programming system (and you can't get much lower than that) might be able to construct this piece of program.

However, 'insert' is a different matter and requires real ingenuity! We wish to state that insertion of i is contained in the operation of assigning to the integerset the value obtained by taking its union with the unitset containing i. Again, the code is not very difficult for a human being to construct:

```
procedure insert (i:integer);
assert ⊑ integerset:∪ {i};
if ¬ has (i) then
if size = C then goto overflow
else begin size:=size+1; A[size]:=i end;
```

If i is not in the set, we must be careful of Dr. Milner's problem. If size=C we must be careful to fail to terminate, since we cannot complete the operation due to overflow. Failure to terminate is a very ugly phenomenon and I have used a very ugly programming language construct to deal with it. Professor Dijkstra may substitute a decent abortion clause!

I have also got to program 'remove' which I haven't done at the moment because that requires some real ingenuity, which incidentally in the published version of this talk I didn't exercise. It suggests to me that questions of efficiency are equally important as questions of termination. These are more closely related to each other, I think, than questions of formal or conditional correctness are. I would rather deal with efficiency and termination together, rather than treating termination together with formal correctness; but I have no very good reason for this.

Proof Method

How do we prove the correctness of the representation? We have got to prove that the effect of executing a bit of concrete code is the same as a certain abstract assignment. The effect of 'insert (i)' has got to be the same as the assignment:

$$\text{integerset} := \text{integerset} \cup \{i\}.$$

We have got to prove an equivalence, or at least a containment (if you are willing to be satisfied with a containment) between the concrete body of the procedure 'insert' and the corresponding operation in the abstract program. Here we come up against two problems which baffled me for many years. Firstly, how can I use the assertional method, which associates assertions with certain points in the program, in order to prove an equivalence, or rather a containment? Secondly, 'integerset', which is the target of the assignment in the abstract program, isn't a variable at all, it is an expression; and we all know, from our knowledge even of Algol, that you can't assign a new value to an expression. Well, I can't explain why, but these problems do seem to go away.

First some notation; since I am using { } for sets, I have used thicker brackets as follows:

$$P\{\!|Q|\!\}R$$

means - if P is true before execution of Q then R will be true when Q terminates; if Q does not terminate, this is vacuously valid.

In order to prove the equivalence, introduce a fresh variable i_0 to stand for the value of 'integerset' before the operation. I shall prove that

$$\text{invariant} \;\&\; \text{integerset} = i_0 \;\{\!|\text{body of insert}|\!\}\; \text{invariant} \;\&\; \text{integerset} = i_0 \cup \{i\}$$

Now this is a way of translating an equivalence (or rather a containment relationship) into the corresponding assertional problem. The equivalence problem is to prove that

body $\subseteq$ integerset: $U\{i\}$.

I have translated the problem in equivalence theory into the corresponding problem of assertional theory, neglecting the fact that the integerset is an expression and not a variable. Now just replace the invariant and integerset by the definitions that we gave to them when we designed this representation. Integerset and the invariant will now both be expressions. Then we prove the result using the same methods as have been used for proving algorithms, which I won't go into now. The problem of assigning to something which is an expression just seems to go away.

The validity of the translation of the proof problem in equivalence theory to the proof problem in assertional theory depends on certain restrictions:

(1) No variable other than 'A' and 'size' may be accessed by a procedure of the representation.
(2) No program except these procedures may access 'A' or 'size'.

If we really wish to modularise our programs and separate questions of representation from questions of algorithm, these seem to me to be very reasonable restrictions. It is important in such cases that we should not have to go through an elaborate proof procedure to show that the restrictions have been observed. Instead we should be able to rely on syntactic definitions, in other words compile-time checks of the scope of these variables, in order to enforce these restrictions.

The Simula Class

The implications of the above for language design are clear. What we need in order to be able to declare and program our own representations of new data types or classes is something very

similar to the Simula Class concept. This is a piece of program introduced by the **word** class, which otherwise has exactly the same form as an Algol procedure declaration. It starts with begin and finishes with end and inside it may contain declarations of data and procedures. It also contains code which is executed when the class is invoked, in other words when a new variable of this class is declared, in order to initialise its value. For classes it is extremely important that you should give the variables a good initial value which satisfies the invariant.

In Figure 3 I have written the representation of integerset as a Simula class, but have put in the various assertions which I feel should be a standard practice in the use of classes which declare new data types. I have put the assertions, the mapping and the invariant, and then in each procedure body is the assertion which describes its intended effect, followed by dots to show where the implementer will actually have to write some code.

```
class   integerset;
begin   A: array [1..C] of integer;
        size: integer;
assert  integerset = {i| ∃ j:1..size, A[j] = i}
        & size = #integerset & 0 ≤size≤C;
procedure  has (i:integer):boolean;
assert  ≡ i ∈ integerset;
        . . .
procedure  insert (i:integer) ; assert ⊑ integerset: ∪{i};
        . . .
procedure  remove (i:integer); assert ⊑ integerset:-{i};
        . . .
    size:= 0; assert  integerset = {};
end integerset;
```

Figure 3

Implementation

The implementation of Simula class is likely to use closed subroutines, but I would like to explain it by a process of textual substitution, which is the same method used in Algol 60 to explain procedure calls. I hope this is a good way of convincing you that it means something, and that what it means is not too complicated, and is not going to require any run-time administration apart from that with which you are already familiar, the run-time stack.

When this class is used, as I have explained, it allows the declaration of variables of the new type 'integerset'. Figure 4 is an example of a block whose body is Q (which may contain other declarations of variables of type 'integerset'). In order to implement the declaration of s we make a new copy of this block, with the declaration replaced by a declaration of variables 'A' and 'size' which are going to be used to represent the particular instance 's' of an integerset. In order to make sure they get unique names, I will call these 's.A' and 's.size' using the dot notation to construct new identifiers which are guaranteed to be fresh. So corresponding to each variable of type 'integerset', in the real executed program we set up a different array 'A' and a different variable 'size'. Then we also declare different instances of the functions 'has', 'insert' and 'remove' and call them 's.has', 's.insert' and 's.remove'. Inside the body of 's.has' etc. we will refer to 's.A' and 's.size' rather than 'A' and 'size'. Then we have the initialisation and finally execute the body of the block as before. The body Q will contain calls of 's.has' etc., this time with the dot actually in place, so that Q may actually operate on 's' by means of these procedures which are now local to this instance of 's'.

```
    begin  s:integerset; Q end
```

is implemented as

```
    begin s.A: array [1..C] of integer;
          s. size: integer;
          function s.has (i:integer):boolean;
                    . . .
                    . . .
          s.size:=0;
          begin Q end
end;
```

Figure 4

Lecture 2 Monitors

In this lecture I will show how one can implement the representation of an abstract variable in a manner which makes it possible to share that variable between two or more parallel processes. I will show how the additional problems that arise in parallel processing can be tackled in a systematic and structured way by the class mechanism.

My example is concerned with adding and subtracting items to and from a sequence of integers, and includes the abstract operations:

(i) s:integersequence;

(ii) s:= < >;

(iii) s:= s^ <x>;

(iv) {y:= first (s) ; s:= rest (s) };

where statement (i) declares a single variable, s, to be of type integersequence, and (ii) initialises s to be the empty sequence, denoted by <>. Statement (iii) indicates output to the end of the sequence, which I equate with concatenating the existing sequence to the sequence of one item, <x>, and assigning the result back to s. Using the abbreviation introduced in the previous lecture, I could write s:^<x>. Statement (iv) is the operation of inputting an element from the sequence to the variable y and removing this element from the sequence. It is this pair of assignments, taking place as a single unit of action, that we will model in our concrete representation.

A practical example of this algorithm in a non-parallel processing environment might be of a problem which we solve by division into sub-problems. Whenever we reach a sub-problem which we cannot solve immediately, we store an integer representation of it in the sequence. Then each time we finish all the processing necessary, we pick a sub-problem off the front of the sequence and

```
class    integersequence;
begin    buffer : array [0.. N-1] of integer;
         firstpointer, count : integer ;
  assert          0 ≤ firstpointer ≤ N-1
      and         0 ≤ count ≤ N
      and         integersequence =
        if count = 0 then empty
                     else  <buffer [firstpointer],
                            buffer [firstpointer ⊕ 1],...
                            buffer [firstpointer ⊕ (count-1) ]>;
      note ⊕ means addition modulo N;

procedure  output (x : integer);
      assert ⊑ sequence := sequence^<x>;
      begin if count ≥ N then goto error;
            buffer [firstpointer ⊕ count] :=x;
            count := count + 1
end output;
procedure input (result x : integer);
      assert ≡ x:= first (sequence); sequence := rest (sequence);
      begin if count = 0 then goto error;
                 :
end input;

count := 0; firstpointer :=0;    assert ≡ sequence := <>
end integersequence;
```

Figure 5

start to process it. Further sub-problems are appended to the end of the sequence. In that way we proceed sequentially to the solution of the full problem, processing sub-problems in the order in which they occur. In our example, we assume that we know that the number of items in the sequence will never be greater than N, and so a cyclic buffer is an appropriate representation for the sequence.

Now we will program this problem using the lessons learnt in the previous lecture (Figure 5). The class is called integersequence. It consists of an N-element array of integers called 'buffer' which represents the sequence; an integer, firstpointer, to point to the array element which is the next to be input from the buffer; an integer, count, indicating the number of items in the array and thus the length of the abstract sequence; some assertions on these variables; the procedures input and output; and some initialisation code.

Let us look at the assertions on the variables since these will define their purpose (and we start with the invariants this time). I make it a practice to write out as many facts about the variables I can think of. Clearly, firstpointer must remain within the range of the buffer (0..N-1), there can never be less than zero or greater than N items in the buffer, but there are no constraints upon the values of the integers which may be stored in the buffer. Turning to the abstraction function integersequence, we will define it in terms of count. When count is zero the sequence must be empty, but when count is not zero the sequence consists of the displayed sequence of items, the first item is always pointed to by firstpointer. The next item is obtained by adding one to firstpointer and looking at that position in the buffer, provided the arithmetic is always done modulo N. This is the essence of the cyclic nature of the buffer.

The design of this representation has already involved a number of important decisions. For instance, the "administrative" variables are chosen as firstpointer and count, whereas I might have chosen firstpointer and lastpointer. However, using firstpointer and last-pointer, it is impossible to tell whether a sequence is empty or full.

Because I started with the invariants and abstraction function, I was able to see and avoid this kind of problem before actually writing any code at all. Thus I believe that formulation of assertions gives a preliminarily test of the adequacy and desirability of a chosen representation. But to be quite honest, the real reason why I took the right decisions was from bitter experience of writing this program many times before.

In the previous lecture I made some unkind remarks about pictures. Those remarks notwithstanding, I hope Figure 6 will confirm what you have understood from the assertions.

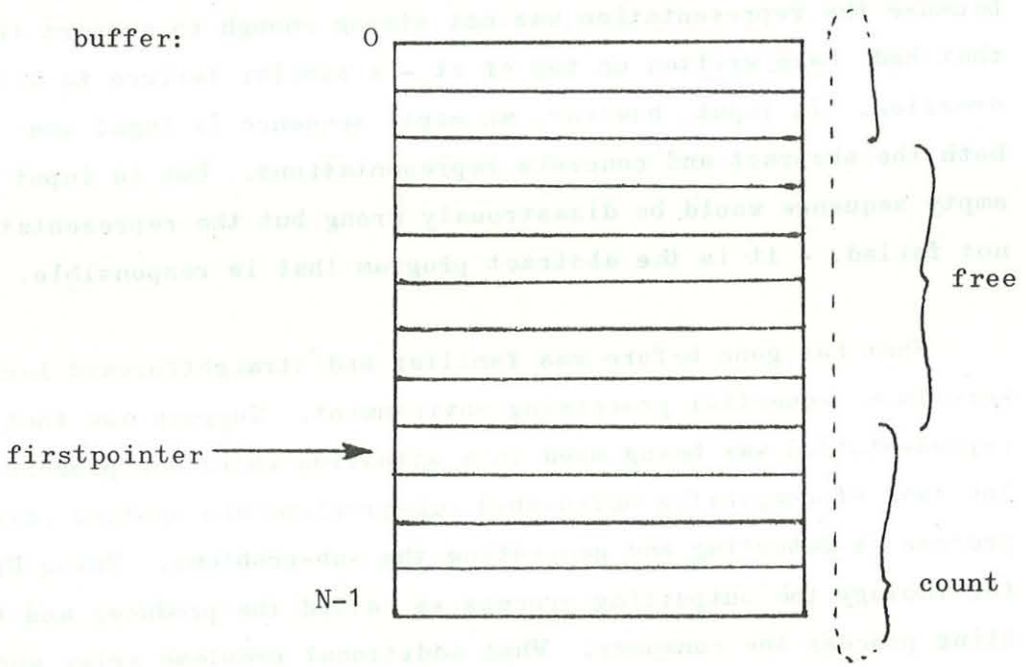


Figure 6

Referring again to Figure 5, in the procedures output and input I have stated precisely what each is to do by their equivalence to statements in the abstract algorithm. Also, I am very careful to check that I am able to perform an operation before I actually perform it. The goto is intended to make it clear that if I cannot perform an operation, then control will never return to the point following the jump to the label "error". A jump is intended to be suicidal. We have put this

test on the length of sequences in here because the concrete representation I have chosen is only partial: it will only deal with a subset of possible sequences. We could have removed this test and made it a pre-condition of a user of the class that he does not generate sequences which are too long. And, indeed, by including this test, if the user over-extends the sequence, his program will die suddenly in an unexpected place. How repsonsible the user should be for his own actions, is one of the decisions that the designer of the representation must make. Similarly, for input there is a termination clause if the sequence is empty. It is interesting to compare the respective tests. In output, failure is caused by a breakdown in the representation because the representation was not strong enough to support the program that had been written on top of it - a similar failure to arithmetic overflow. In input, however, an empty sequence is legal and valid in both the abstract and concrete representations. But to input from an empty sequence would be disastrously wrong but the representation has not failed. - it is the abstract program that is responsible.

What has gone before was familiar and straightforward because we were in a sequential processing environment. Suppose now that this representation was being used in a situation where one process has the task of completing unfinished sub-problems and another parallel process is detecting and generating the sub-problems. Using Dijkstra's terminology the outputting process is called the producer and the inputting process the consumer. What additional problems arise when we have to represent a variable which is to be shared in this environment? We are fortunate because the proof of validity of the chosen representation is independant of the order the procedures are invoked, either in a sequential or a parallel environment. What cannot be tolerated is that the bodies of the procedures be invoked in parallel or in an interleaved fashion. This would cause chaotic results and proofs about the program would not apply.

The programmer of the producer and consumer procedures cannot be expected to solve this exclusion problem, and so we must assume that there are facilities in our language, and in an underlying run-time mechanism, which prevent it ever happening. That is, once one procedure operating on a given variable is in execution there is no way any

other may begin execution until the first has finished. The bodies of such procedures are known as critical regions. Meanwhile, there is a second problem. How do we prevent input from an empty buffer or output to a full buffer? This I call the synchronisation problem. The boundaries between this problem and the exclusion problem are not absolute but within any given data representation we are able to distinguish between them. It is no use for the producer to check to see if the variable count is equal to N before actually entering the output procedure, because some other producer may get in between the test and the entry and make the variable equal to N; in which case the entry is invalid. So the synchronisation aspect which prevents this happening has got to be programmed as part of the data representation itself, and these tests on the full or empty states of the buffer have got to be part of the indivisiable actions which the program is trying to invoke. When we attempt to invoke an action which is undefined we must delay the attempt until some other process has brought the data into a condition in which the action is defined. Thus we introduce the idea of a process waiting in situations where in the sequential environment it would just fail. The important thing is to be able to specify the conditions on which a process must wait. Conditions in programming languages are specified by Boolean expressions and here we use the notation

when B do Q;

which means intuitively, wait until B is true; if it is true already then proceed and execute Q; if B is not true then wait until it becomes true. In a parallel program it makes sense to wait for something because there should always be some other process operating which can proceed and may eventually "awaken" the waiting process. If this is not the case, and moreover, every process goes into a wait state then deadlock has occured. Deadlock is a special case of non-termination, and again I shall ignore it.

Figure 7 shows the output procedure. Its meaning and action are exactly the same as before but I have replaced the error jump, which

causes what I like to call a detectable non-termination of the procedure, by a command which causes the process to go to sleep until some other process acts upon the representation in such a way that count does go below N; and then the waiting process will automatically resume.

```
procedure output (x : integer);
assert ≡ sequence := sequence⌢<x>;
when count < N do
        begin buffer [firstpointer ⊕ count] := x;
              count := count + 1
        end;
```

Figure 7

The proof rules for this synchronisation condition are very simple: we take the proof rules we used for a sequential program and add another clause to the pre-condition which states that if you pass the do symbol you may be certain the condition after the when is true, e.g. that count is less than N. No more effort is required. Two important restrictions mentioned in the first lecture apply here. First, the procedures of a representation may not access or change any variable other than a variable of the representation. Second, that the only way of changing the value of a representation variable is by executing one of the procedures of the representation. While a process waits on a when, the exclusion on the procedure bodies must be released so that other processes can invoke these procedures to change the values of variables. Only by this means can the wait condition of a sleeping process ever become true. However, when the condition does become true, the successful test must be part of the same critical region as the action which follows. Thus at most one process (at a time) will be resumed.

The mutual exclusion can be implemented by associating with each represented variable an extra component variable of type semaphore. This extra variable is called mutex, for mutual exclusion, and is initialised to one:

```
mutex: semaphore; mutex:=1;
```

We protect critical regions by surrounding each call on a procedure by the pair of operations:

```
P(mutex)
   :
   .
V(mutex)
```

which prevents more than one process executing the procedure at a time.

Synchronisation requires a more elaborate operation. For this we introduce a second semaphore with each variable of the class. This semaphore I shall call sync. It is initialised to zero and, indeed, will remain zero throughout its lifetime, since the only reason it exists is for waiting and the only time we can be sure we are going to wait on a semaphore is when its value is zero.

```
sync: semaphore; sync:=0;
```

Thus sync is a semaphore which "holds" a queue of processes waiting for their conditions to become true. Now at the end of each operation on a representation variable it is necessary to test whether any of the conditions of the waiting processes have become true. It is only at the end of such operations that the values of these variables will have been changed. We replace the simple V(mutex) at the end of each operation by

```
while anyone-is-waiting-on-sync do V(sync);
      V(mutex);
```

The while-loop releases all the processes waiting on sync. When we execute the V(mutex) operation just one of these processes, all of

which have P(mutex) as their first operation, is able to proceed. The process which proceeds must first re-evaluate its wait-condition, B. If it finds it true then it continues, and all the other processes remain waiting on the mutex semaphore. If it finds it false then it will hang up on the synchronisation semaphore again, having released mutex to allow one of the other processes to test its wait-condition. This continues until a process is able to execute its desired operation. In this way, all waiting processes are able to retest their wait-conditions whenever another process leaves a critical region.

We will implement when B do ... by

```
while ¬B do begin
                V(mutex);
                P(sync);
                P(mutex)
                end;
. . .
```

That is, if B is true we wish to just proceed. But if B is false then we release the exclusion to allow the other processes to enter (which may make our condition true), then we delay ourselves on the synchronisation semaphore. When we are released from sync by the action of another process we immediately try and sieze the exclusion again. When we eventually succeed, we come back and retest the wait-condition. If the condition is now true we proceed with our operation without releasing the exclusion again, but if it is still false we must repeat the process.

The construction I have described is called a critical conditional region. The use of these conditions to guard regions of code has been elegantly generalised by Dijkstra, and applied to the solution of problems in sequential programs. This is yet another example where discoveries necessary to organise parallel programs have been found useful when applied to sequential programs.

There are, however, some serious disadvantages with this implementation. If waiting is a relatively rare event then this does not matter very much. But if waiting is fairly common, and the queue of waiting processes becomes long, then there is a great deal of retesting to do. Another difficulty, particularly for operating systems, is that we have no control over the order in which the waiting processes are resumed when more than one has a condition satisfied. Further, some processes may wait for ever even though their condition is satisfied, because every time it is satisfied the condition of some other process is also satisfied, and that one always gets control first. Ways around this drawback can sometimes be programmed explicitly.

There are two approaches to solving these problems. One can look for a more subtle implementation. For example, if one associates different synchronisation semaphores with different conditions (and the condition does not depend upon parameters of the procedure) then it is possible first to retest the condition only once on behalf of the process at the head of the queue. If it is false, then everybody else will be true. When the condition depends upon parameters transmitted by the process into the representation then there is less alternative to retesting. It may be rather inefficient, but it is a general method and the code is extremely short. An alternative approach is to introduce a more primitive synchronisation tool which allows you to program the waiting loop yourself. This may give you the ability to program more efficient solutions when you have to.

Lecture 3 Parallel Programming : An Exploration

I want to take you briefly through a chain of reasoning by which one might develop a theory and approach to the problems of parallel programming; or perhaps better, an approach away from the problems, since I always prefer to avoid problems rather than solve them. My objective is to discover cases in which parallel programs are no more difficult to design correctly than sequential ones, and I will take as a thesis that difficulty is correlated with the difficulty of proving the correctness of the program.

Notation:

Q1 // Q2 Q1 and Q2 are to be executed in parallel, terminating only when both Q1 and Q2 terminate. If either fails to terminate, then the whole construction fails to terminate.

$Q1 \sqsubseteq Q2$ (procedural containment) Whenever Q1 terminates, it has exactly the same value and exactly the same effect on all variables as Q2.

Professor Dijkstra "Does this only apply to deterministic programs?"
Professor Hoare "Yes, I will only apply it to deterministic programs."
Dr. P.E. Lauer "This containment does not take account of a program becoming undefined because of an error rather than an infinite loop."
Professor Hoare "I like to treat all errors in the same way, as equivalent to coming up against a time imposed by the implementation. So I'll regard division by zero as equivalent to non-termination."

$Q1 \equiv Q2$ (procedural equivalence) Mutual containment, that is, both $Q1 \sqsubseteq Q2$ and $Q2 \sqsubseteq Q1$

P {Q} R (conditional correctness) If Q terminates, and P was true before it started, then R will be true on termination of Q. If Q never terminates, then this is vacuously true.

Informal conventions:

"process" a part of a program which proceeds in parallel with other parts of the program.

"parallel program" a program which contains, at some stage or another in its execution, two or more processes.

I can't and won't define what I mean by a process.

The following theorem is a direct consequence of the definitons:

$$\frac{Q1 \sqsubseteq Q2 \ , \ P \{Q2\} R}{P \{Q1\} R}$$

Disjoint Programs

I wish to pose the question: Under what conditons can we be sure that a parallel program has an identical effect to a sequential one, that is, that

$$(Q1 \ // \ Q2) \equiv (Q1 \ ; \ Q2)?$$

There are many conditions under which this will be true, but we would like to have conditions which are very easy to establish. We would like to have a very simple condition which can be checked by syntactic criteria, and not by having to consider the semantics of the program, or the behaviour of the program when it is executing (because reasoning about that is difficult). We would like to have a program to check whether the construction of any program satisfies the rules which make the above equivalence trivial. We would be able to guarantee equivalence, not by a proof but by compile time checks.

My answer is contained in the title of this section - when Q1 and Q2 are disjoint (in the sense that no variable which is possibly updated by one of them is mentioned in the other) the equivalence will certainly hold. In a carefully designed language it is possible to tell which variables can be assigned to, and in such a language it is possible to check whether two processes are disjoint.

This may seem to be evading the problem, but I suspect that quite a lot of the practical uses to which programmers want to put a parallel programming facility can be constructed to conform to this restriction. Here is a simple program which overlaps input/output operations.

```
begin  lastone, thisone, nextone : item;
input (lastone);
{ process(lastone) // input(thisone) } ;
    while  notempty(input) do
    begin { input(nextone) // process(thisone)  // output(lastone) } ;
          lastone := thisone ; thisone :=nextone
    end;
{ process(thisone) // output(lastone) } ;
output(thisone)
end
```

The main part of the loop overlaps input of the next item to be processed, processing of this item, and output of the last item. These three things can be done in parallel because we allocate three disjoint locations of store on which the three processes are to operate. When all three have finished, this item and the next item are copied across to set up conditions for the loop to proceed in parallel again. The outside of the loop deals with situations where the buffers only contain one or two items and the amount of parallelism is consequently limited.

In implementation, the parallel split in the innermost loop could be an expensive operation, so that this program is not a very good one unless the overlapped operations take rather a long time.

We proceed with our investigation by asking why the condition of disjointness might be found to be too restrictive in the construction of parallel programs. It may be possible to relax the restriction a little, and so increase the class of parallel programs which can be written and proven without any more difficulty than sequential programs.

Competing Programs

One reason why two parallel processes might have to interact is that, for example, both processes wish to use a single line-printer (or some other resource which is in short supply). Each of them, fortunately, wishes to use the line-printer only for relatively short periods during its execution, and is quite prepared to relinquish the line-printer after each period of use. We could define a resource as any part of a

computer system which can, and must, be allocated to only one process at a time. Examples are a line printer, a computer operator or even a single word of main store.

We know that an Algol-like language allocates and de-allocates main storage in a dynamic fashion, and the implementation ensures that no individual word of main store is allocated to two purposes simultaneously. This gives a clue as to how we might extend the range of parallel processing possibilities, while retaining the essential feature of a compile time check, together with a proof method which is no more difficult than for the sequential case. Algol-60 insists that before you use a word of main store, it must be declared and given a name; and ensures automatically that after you finish using the main storage it must be returned for reallocation. These restrictions are inherent in the lexicographic structure of the language itself and are not enforced by any run time checking.

The question arises : Under what conditions can we be sure that the claiming, use, and release of a resource by a process has an identical effect to the declaration and use of a variable in a block?

The answer is inherent in the above discussion. Design a language to use the same notation for claiming and locally using a resource as for all other declarations (which also claim a resource - main store). Here is an example.

```
begin   listing : lineprinter ; s : integersequence ;
          .
          .
          .
listing.output(line);
s.output(x)
end
```

Within this block a parallel process wishes to use the line-printer, so it declares a variable of type 'lineprinter', which is perhaps a built-in type of an implementation. Inside the block it can operate on that line-printer by calling the built-in facilities of the line-printer class, as provided by the implementation, in exactly the same way as operations on variables of other classes like

'integersequence'.

The normal scope rules of Algol-60 ensures that no one can use a resource without claiming it; it is logically impossible to send a line to the line-printer except within the scope in which the line-printer has been declared. The declaration is intimately connected with the acquisition of the resource, and exit from that scope is intimately connected with the release of the resource. At least for terminating programs, the rules also ensure that no one forgets to release a resource. Furthermore, wherever and whenever multiple use of the same resource is available (for example two line-printers) an implementation may satisfy several claims simultaneously. The same parallel program will run with a shorter elapsed time on such a machine.

Processes which have to interact because they require use of the same resources, I call competing processes. The cases of disjoint processes and competing processes include, I believe, fairly large classes of useful programs.

Commutivity

Now I would like to return to disjoint processes and ask a rather important question : Why do we believe that the effect of executing disjoint processes in parallel is the same as executing them sequentially? I don't know whether it is possible to give a completely rigourous proof of this, since any proof must depend on some definition of atomic units of action evoked by the execution of a program (a line which has been explored by Dr. Bekic).

Here is a very informal proof. Let U(Q) be the set of 'atomic' units of action evoked by the execution of Q (for any definition of 'atomic'). U(Q1) is said to commute with U(Q2) if (q1 ; q2) ≡ (q2 ; q1) for any q1 ∈ U(Q1) and any q2 ∈ U(Q2). Actions from disjoint programs obviously commute, as shown by the example:

x := y+z; w := v+z ≡ w := v+z; x := y+z

Under this condition, an arbitrary interleaving of units of action from Q1 and Q2 has the same effect as any other, and in particular the interleaving with all units from Q1 proceeding all those from Q2 - which is of course the sequential case.

Thus, $Q1 \;//\; Q2 \equiv Q1 \;;\; Q2$

and indeed $Q1 \;//\; Q2 \equiv Q2 \;;\; Q1$

Equivalent programs obviously can be proved by the same proof rule.

Co-operating Processes

Can we extend the principle of commutivity and apply it to co-operating **processes** which work on some common task to the solution of which they both contribute? Consider an example. Suppose we are given a set variable s, and an operation :

```
s.remove(n)
```

which has the effect of removing the integer n from the set s (i.e. $s := s - \{n\}$). I am not going to describe how the set s is represented, and will provide only an abstract program. Let us also suppose that the remove operation is atomic. If two parallel processes attempt to remove something at the same time the effect is the same as if one had been completed before the other begins. The remove operation commutes, having the rather nice property that for all m and n it doesn't matter in which order they are removed from the set.

$s := s-\{n\};\ s := s-\{m\} \equiv s := s-\{m\};\ s := s-\{n\}$ for all m,n.

Removal is of course defined so that if the integer is not a member of the set, then nothing happens.

Provided we can be sure that whatever interleaving occurs, the effect is the same as the sequential case, then we also know that the actual order of the operations is immaterial. Below is a simple example using this knowledge in a program.

```
begin  sieve : set of integers;
     procedure  remove multiples of (n : integer);
     begin  i : integer;
     for  i :=n² step n until N do sieve.remove(i)
     end;
sieve := {i| 2 ≤ i ≤ N } ;
p1:= 2 ;  p2:= 3;
while p1² ≤ N do
    begin  {remove multiples of(p1) // remove multiples of(p2) };
    p1:= min {i | i > p2 & i ε sieve  };
    if p1² <N then p2 := min {i | i > p1 & i ε sieve }
    end
end
```

The basic principle of the sieve of Eratosthenes is that within the innermost loop all multiples of a given prime are removed from the sieve. This particular program is organised so that multiples of the two primes, p1 and p2, are removed in parallel. No difficulty arises because the body of the procedure 'remove multiples of(n)' operates on the sieve only by a commutative operation, 'remove'.

Communicating Processes

Provided co-operating processes co-operate via commutative operations, we can treat them without very much more difficulty than we treat sequential processes. But there is a serious, and absolutely vital, restriction on the forms this co-operation can take, because the rule of commutivity obviously prohibits communication between two processes. It is impossible to communicate any information between processes which operate on shared variables only by commutative operators. The simplest proof of this is that with commutative operators, Q1//Q2 is equivalent to both Q1;Q2 and Q2;Q1. Both of these are valid implemenations of the // operator. It is impossible to communicate from Q1 to Q2, since if it were possible to communicate, it would be impossible to execute them in order Q2;Q1. Similarly for the impossibility of communication from Q2 to Q1.

Professor Michaelson "That doesn't sound correct. Both Q1 and Q2 could, before terminating, check whether their task was completed, and if not, perform the remainder of the task. They would commute externally, but would perform different actions depending on the order in which they were executed."

Professor Hoare "This is the case where Q1 and Q2 co-operate by means of memo functions. Q1 calls a computation which stores its result, and then Q2 performs an operation which if the result has already been stored just uses it, but otherwise first computes and stores the result. However, the progress of Q2 cannot depend on anything Q1 might or might not have done; Q2 cannot rely on Q1 having achieved a half-way objective".

Professor Michaelson "The completion of Q1 may not depend on what Q2 has done, but the actual things Q1 does may so depend. Are you defining 'communication' in a way that would only permit one process to complete if the other has set a signal - a rather restrictive form of definition?"

Professor Hoare "No. I think that it is impossible for Q1 to call a function which delivers a result that is dependent on the progress of Q2, and therefore the only technique for passing results from Q2 to Q1 is the memo function. But perhaps some further thought is required. My argument is, of course, entirely informal, and is based on the view that results cannot be communicated from something that happens later to something that happens earlier."

I introduce a new definition. U(Q1) is said to semicommute with U(Q2) if $(q2;q1) \sqsubseteq (q1;q2)$ for any $q1 \in U(Q1)$ and any $q2 \in U(Q2)$. When this is the case, we can sort all the interleaved actions of Q1 and Q2 in such a way that all the actions of Q1 move to the left, and all those of Q2 move to the right. Everytime we make an interchange, moving an action of Q1 to the left of an action of Q2, we make the program more defined. Eventually, all the actions of Q1 will be to the left of those of Q2. The program is then sequential, and is at its most defined. Thus,

$$Q1//Q2 \sqsubseteq Q1;Q2 .$$

So in this situation we can prove the correctness of the sequential program Q1;Q2, and be confident that the same proof applies to the program Q1//Q2, which may terminate less often.

How can we be sure that this program terminates at all? Well, we can't be sure, because the weak conditional correctness methods allow an implementation to reject the computation of any program whatsoever, at any stage. The engineering quality of an implementation can be judged by the frequency with which it rejects programs, and implementations can be compared on this basis, but in the last resort, the implementation which rejects all programs is perfectly valid and just a bit worse than all other implementations.

So, we make it the responsibility of an implementation to ensure that if Q2 attempts an operation q2 at a time when this is undefined, the operation will be merely <u>delayed</u> until (hopefully) Q1 performs such operations as will make q2 defined. The delaying of operations was considered earlier under the term 'synchronisation', where I provided a method for thinking about synchronisation problems, and a notation for specifying how to determine whether an operation need be delayed. Thus using classes and conditional critical regions, we can ourselves program a representation of the variable shared between parallel processes. Obviously, the example I'm working towards is the classical case of producers and consumers.

Given that a language or programmer has implemented the concept of a sequence, and provides as atomic operations:

```
s.output(x)        ( ≡ s := s ^ <x                    )
s.input(y)         ( ≡ y := first(s); s := rest(s))
```

then these two operations semicommute:

$$s.input(y)\ ;\ s.output(x) \sqsubseteq s.output(x)\ ;\ s.input(y).$$

The inequality is strict, since in the case that s is empty, input from s is not defined whereas output to s is defined. A process which only outputs to s is known as a 'producer on s', and a process which only inputs from s is known as a 'consumer of s'. Notice that the rule of semicommutivity only permits forward communication, and in this

particular case, only one producer and one consumer, because with two producers the output operations do not commute. To allow multiple producers and consumers the shared abstract object can be a multiset or a bag - on which both the input and the output operations do commute.

Very often the ability to define successful communication and co-operation depends on the ability to abstract the operations needed. The parallelism I have offered seems entirely deterministic since it always gives the same results as a deterministic sequential program. However, it may well be the case that the concrete values (of the representations) of the shared variables do depend on the sequence in which the operations have been invoked. At the level of the implementation on a computer, the process may be highly non-deterministic. Only after applying the abstraction function do the operations commute. This suggests that abstraction is an absolutely necessary key to the organisation of parallel processing.

Take the example of sharing a line-printer - why do we believe that a line-printer can be shared between two processes (provided that use of the line-printer is an atomic action, to prevent arbitrary interleaving of lines)? This is allowable when we do not mind in which order the files of output appear. That is, in just the case when the listings are burst before we receive them, and therefore can't tell what the order was on output. So here is another example of a very important abstraction function - the ruler used by the operator to separate sheets of line-printer paper. Without this abstraction the system would not have the freedom to change the order in which it executes jobs.

I would like to admit that there is a serious, perhaps unsolvable, weakness of this technique, which effectively prevents it from being used for two-way communication. I discussed placing an obligation, or at least an objective, upon an implementation, to terminate a program whenever this is possible. Certainly, any implementation

must be prohibited from terminating (successfully) a program which contains an infinite loop or an undefined operation. But otherwise, if the implementation can find a way of terminating a program successfully, it should do so. Now suppose we have a situation in which some interleavings of the actions on shared variables lead to deadlock, and some do not. Then, I think that all non-deadlocked computations will produce the same result, even though such computations may be in the minority. Apparently, our implementation is obliged, by means of horrendous backtracking, to find a terminating interleaving whenever one exists. Terrible! Of course an implementation which does not backtrack is permissible. The problem is to decide which implementation is preferable, and why. My only solution is to ask the programmer how much he is prepared to pay, in which case he will prefer the non-backtracking implementation. But fortunately, with one-way communication, deadlock is impossible, and the problem does not arise.

Professor Scott "Does Schwarz's language SETL have anything to do with your use of the class concept?"

Professor Hoare "I've often asked myself that question. As far as I can make out, there are two fundamental flaws in that language. One is that it does not bind variables and expressions to a given type, which I regard as an essential prop to my weak intellect."

Professor Scott "Is it not your implementation which is weak?"

Professor Hoare "Well, both I think. In this respect the computer needs the prop as much as I do. I don't think of typing as a sordid matter which the computer should not need to impose on a human being, but rather as an aid to constructive thinking, as do you, and Dr. Bekic. He introduces types, not because of any execution requirement, but because he needs them to check the sensibleness of what he is writing down. The second flaw (in the SETL project) is that too much emphasis is placed on skilfull optimisation. I don't belive in that level of

optimisation. Obviously I agree that you should describe what your program is trying to do before you try to do it. The idea that the representation is a lower level of decision which should be kept separate is a good one, but to delegate it to the computer leads to difficulty. The best that I can do is to give the programmer a way of thinking about his designs and how to translate them into an implementation. After that I wouldn't want to deprive him of the fun of actually doing it."

Dr. Burstall "Is it the case that if $Q1 \equiv Q1'$ then for all $Q2$, $Q1//Q2 \equiv Q1'//Q2$? Is it true when parallelism is restricted to semicommutative operations?"

Professor Hoare "I can't, unfortunately, give compile time checks for semicommutativity. If you're willing to bear with me on that difficulty, I think that my form of parallelism is best treated by relating it directly to the equivalent sequential program. In the case of semicommutativity this is fine; only the second process can wait for the first one. With communication in both directions then (a) the proof is very complex and (b) the problems of deadlock are even worse. For semicommutative operations, the parallel operator can be thought of as permission to the implementation to execute things in a different order if it so chooses. However, to perform Q1 followed by Q2 would still be valid. So, in answer to your question, if the answer is yes for sequential programs then it is yes for my restricted form of parallel programs also."

Professor Dijkstra "Well, to my feeling, the restriction to one-way communication is as serious as to ask someone to walk with one leg. For instance, if you wish to exploit the possibility of tremendous parallelism in the super fast Fourier transform, then you just cannot write sequential programs. The interleaving of the separate activities is an essential feature. Each of the processes is synchronised and goes through a cycle in which it receives two results and produces

two results."

Professor Hoare "I make no claim to have exhausted the possibilities of parallelism. Almost like programming, however you try to pin it down, someone will always think of a beautiful way of using parallelism which you haven't covered."

Professor Dijkstra "With one leg you can't walk, so you can't stumble either?"

Professor Hoare "Yes. It's a good rule for those who would otherwise have two left feet."

Dr. Milner "Can we never find a definition of parallelism sufficiently general that people don't find new ways of using parallelism? I'm rather depressed to hear you give up like that."

Professor Hoare "I've not given up. I'm going to see what I can do about Professor Dijkstra's algorithm."

Dr. Milner "This is my criticism. Until we look at more difficult examples of parallelism, the problem will remain unsolved. The simpler examples should come out as a nice special case of the general results."

Professor Hoare "Obviously we're all hoping to meet in the middle. You're starting from one end and I'm starting from the other. You can readily point out that I'm not yet half way. I do think the assertional method I use has a fundamental weakness (which is also its strength - it is in many ways the reason why it is so simple) and that is that it throws away every aspect connected with the passage of time. You cannot even say that a computation is finite, much less that it is bounded by $2^{2^{2^n}}$. Such problems cannot even be formulated in a pure assertional method."

Some Remarks on Teaching

I've listened with fascination to the talks of the other speakers. They have filled me with a desire (not likely to be fulfilled in the immediate future) to teach their subjects in my department. I don't think that my own lectures fill me with the same desire at all. What I have had to say is more in the nature of on-going research than a completed discipline suitable for teaching. If you find any of these ideas attractive, then I would suggest you incorporate them in some existing course. There are three fairly strong reasons against a separate course on this material.

1. No textbook.
2. Many doubts and open questions remain.
3. No facilities for practical work (perhaps least important).

I enjoy theoretical programming but some students don't.

I will describe briefly how this material fits in with courses at Belfast. Abstraction and representation of data is very much emphasised in a 2nd year advanced programming course. Since we do not have a language which implements the class concept, the treatment is fairly theoretical. Concepts of exclusion and synchronisation by means of monitors, together with synchronisation and operating system algorithms form a large part of the 3rd year operating systems course. This way of looking at synchronisation and scheduling problems is a great simplification for the students. They can solve problems which a few years ago would have been research articles (if semaphores had to be used). There is also a 3rd year course of 30 lectures on the Theory of Programming, of which the last three present - not very successfully - the material I have presented here.