Acta Informatica 1, 271—281 (1972)
© by Springer-Verlag 1972

Proof of Correctness of Data Representations

C. A. R. Hoare

Received February 16, 1972

Summary. A powerful method of simplifying the proofs of program correctness is suggested; and some new light is shed on the problem of functions with side-effects.

1. Introduction

In the development of programs by stepwise refinement [1–4], the programmer is encouraged to postpone the decision on the representation of his data until after he has designed his algorithm, and has expressed it as an "abstract" program operating on "abstract" data. He then chooses for the abstract data some convenient and efficient concrete representation in the store of a computer; and finally programs the primitive operations required by his abstract program in terms of this concrete representation. This paper suggests an automatic method of accomplishing the transition between an abstract and a concrete program, and also a method of proving its correctness; that is, of proving that the concrete representation exhibits all the properties expected of it by the "abstract" program. A similar suggestion was made more formally in algebraic terms in [5], which gives a general definition of simulation. However, a more restricted definition may prove to be more useful in practical program proofs.

If the data representation is proved correct, the correctness of the final concrete program depends only on the correctness of the original abstract program. Since abstract programs are usually very much shorter and easier to prove correct, the total task of proof has been considerably lightened by factorising it in this way. Furthermore, the two parts of the proof correspond to the successive stages in program development, thereby contributing to a constructive approach to the correctness of programs [6]. Finally, it must be recalled that in the case of larger and more complex programs the description given above in terms of two stages readily generalises to multiple stages.

2. Concepts and Notations

Suppose in an abstract program there is some abstract variable t which is regarded as being of type T (say a small set of integers). A concrete representation of t will usually consist of several variables $c_1, c_2, \ldots, c_n$ whose types are directly (or more directly) represented in the computer store. The primitive operations on the variable t are represented by procedures $p_1, p_2, \ldots, p_m$, whose bodies carry out on the variables $c_1, c_2, \ldots, c_n$ a series of operations directly (or more directly) performed by computer hardware, and which correspond to meaningful operations on the abstract variable t. The entire concrete representation of the type T can

be expressed by declarations of these variables and procedures. For this we adopt the notation of the SIMULA 67 [7] class declaration, which specifies the association between an abstract type T and its concrete representation:

```
class T;
   begin ... declarations of c1, c2, ..., cn ...;
         procedure p1 ⟨formal parameter part⟩; Q1;
         procedure p2 ⟨formal parameter part⟩; Q2;
                      .......                                  (1)
         procedure pm ⟨formal parameter part⟩; Qm;
         Q
   end;
```

where Q is a piece of program which assigns initial values (if desired) to the variables $c_1, c_2, \ldots, c_n$. As in ALGOL 60, any of the p's may be functions; this is signified by preceding the procedure declaration by the type of the procedure.

Having declared a representation for a type T, it will be required to use this in the abstract program to declare all variables which are to be represented in that way. For this purpose we use the notation:

var(T) t;

or for multiple declarations:

var(T) $t_1, t_2, \ldots$;

The same notation may be used for specifying the types of arrays, functions, and parameters. Within the block in which these declarations are made, it will be required to operate upon the variables $t, t_1, \ldots$, in the manner defined by the bodies of the procedures $p_1, p_2, \ldots, p_m$. This is accomplished by introducing a compound notation for a procedure call:

$$t_i \cdot p_j \langle\text{actual parameter part}\rangle;$$

where t_i names the variable to be operated upon and p_j names the operation to be performed.

If p_j is a function, the notation displayed above is a function designator; otherwise it is a procedure statement. The form $t_i \cdot p_j$ is known as a *compound identifier*.

These concepts and notations have been closely modelled on those of SIMULA 67. The only difference is the use of **var**(T) instead of **ref**(T). This reflects the fact that in the current treatment, objects of declared classes are not expected to be addressed by reference; usually they will occupy storage space contiguously in the local workspace of the block in which they are declared, and will be addressed by offset in the same way as normal integer and real variables of the block.

3. Example

As an example of the use of these concepts, consider an abstract program which operates on several small sets of integers. It is known that none of these sets ever has more than a hundred members. Furthermore, the only operations

actually used in the abstract program are the initial clearing of the set, and the insertion and removal of individual members of the set. These are denoted by procedure statements

$$s \cdot \text{insert}(i)$$

and

$$s \cdot \text{remove}(i).$$

There is also a function "$s \cdot \text{has}(i)$", which tests whether i is a member of s.

It is decided to represent each set as an array A of 100 integer elements, together with a pointer m to the last member of the set; m is zero when the set is empty. This representation can be declared:

```
class smallintset;
begin integer m; integer array A [1:100];

        procedure insert(i); integer i;
          begin integer j;
            for j:=1 step 1 until m do
              if A[j]=i then go to end insert;
            m:=m+1;
            A[m]:=i;

  end insert: end insert;

        procedure remove(i); integer i;
          begin integer j, k;
            for j:=1 step 1 until m do
              if A[j]=i then
                begin for k:=j+1 step 1 until m do A[k-1]:=A[k];
                  comment close the gap over the removed member;
                  m:=m-1;
                  go to end remove
                end;

end remove: end remove;

        Boolean procedure has (i); integer i;
          begin integer j;
              has:=false;
              for j:=1 step 1 until m do
                if A[j]=i then
                  begin has:=true; go to end contains end;

end contains: end contains;

    m:=0; comment initialise set to empty;

end smallintset;
```

Note: as in SIMULA 67, simple variable parameters are presumed to be called by value.

4. Semantics and Implementation

The meaning of class declarations and calls on their constituent procedures may be readily explained by textual substitution; this also gives a useful clue to a practical and efficient method of implementation. A declaration:

$$\textbf{var}(T)t;$$

is regarded as equivalent to the unbracketed body of the class declaration with **begin** ... **end** brackets removed, after every occurrence of an identifier c_i or p_i declared in it has been prefixed by "$t\cdot$". If there are any initialising statements in the class declaration these are removed and inserted just in front of the compound tail of the block in which the declaration is made. Thus if T has the form displayed in (1), **var**$(T)t$ is equivalent to:

... declarations for $t\cdot c_1, t\cdot c_2, \ldots, t\cdot c_n \ldots$;
procedure $t\cdot p_1(\ldots); Q_1'$;
procedure $t\cdot p_2(\ldots); Q_2'$;
.....
procedure $t\cdot p_m(\ldots); Q_m'$;

where $Q_1', Q_2', \ldots, Q_m', Q'$ are obtained from $Q_1, Q_2, \ldots, Q_m, Q$ by prefixing every occurrence of $c_1, c_2, \ldots, c_n, p_1, p_2, \ldots, p_m$ by "$t\cdot$". Furthermore, the initialising statement Q' will have been inserted just ahead of the statements of the block body.

If there are several variables of class T declared in the same block, the method described above can be applied to each of them. But in a practical implementation, only one copy of the procedure bodies will be translated. This would contain as an extra parameter an address to the block of $c_1, c_2, \ldots, c_n$ on which a particular call is to operate.

5. Criterion of Correctness

In an abstract program, an operation of the form

$$t_i\cdot p_j(a_1, a_2, \ldots, a_{n_j}) \tag{2}$$

will be expected to carry out some transformation on the variable t_i, in such a way that its resulting value is $f_j(t_i, a_1, a_2, \ldots, a_{n_j})$, where f_j is some primitive operation required by the abstract program. In other words the procedure statement is expected to be equivalent to the assignment

$$t_i := f_j(t_i, a_1, a_2, \ldots, a_{n_j});$$

When this equivalence holds, we say that p_j *models* f_j. A similar concept of modelling applies to functions. It is desired that the proof of the abstract program may be based on the equivalence, using the rule of assignment [8], so that for any propositional formula S, the abstract programmer may assume:

$$S^{t_i}_{f_j(t_i, a_1, a_2, \ldots, a_{n_j})}\{t_i\cdot p_j(a_1, a_2, \ldots, a_{n_j})\}S.^1$$

1 S^x_y stands for the result of replacing all free occurrences of x in S by y: if any free variables of y would become bound in S by this substitution, this is avoided by preliminary systematic alteration of bound variables in S.

In addition, the abstract programmer will wish to assume that all declared variables are initialised to some designated value d_0 of the abstract space.

The criterion of correctness of a data representation is that every p_j models the intended f_j and that the initialisation statement "models" the desired initial value; and consequently, a program operating on abstract variables may validly be replaced by one carrying out equivalent operations on the concrete representation.

Thus in the case of smallintset, we require to prove that:

$$\begin{aligned}
&\textbf{var}(i)t \text{ initialises } t \text{ to } \{\ \} \text{ (the empty set)}\\
&t \cdot \text{insert } (i) \equiv t := t \cup \{i\}\\
&t \cdot \text{remove}(i) \equiv t := t \cap \neg\{i\}\\
&t \cdot \text{has}(i) \equiv i \in t.
\end{aligned} \tag{3}$$

6. Proof Method

The first requirement for the proof is to define the relationship between the abstract space in which the abstract program is written, and the space of the concrete representation. This can be accomplished by giving a function $\mathscr{A}(c_1, c_2, \ldots, c_n)$ which maps the concrete variables into the abstract object which they represent. For example, in the case of smallintset, the representation function can be defined as

$$\mathscr{A}(m, A) = \{i: \text{integer} \mid \exists k (1 \leqq k \leqq m \,\&\, A[k] = i)\} \tag{4}$$

or in words, "(m, A) represents the set of values of the first m elements of A". Note that in this and in many other cases $\mathscr{A}$ will be a many-one function. Thus there is no unique concrete value representing any abstract one.

Let t stand for the value of $\mathscr{A}(c_1, c_2, \ldots, c_m)$ before execution of the body Q_j of procedure p_j. Then what we must prove is that after execution of Q_j the following relation holds:

$$\mathscr{A}(c_1, c_2, \ldots, c_n) = f_j(t, v_1, v_2, \ldots, v_{n_j})$$

where $v_1, v_2, \ldots, v_{n_j}$ are the formal parameters of p_j.

Using the notations of [8], the requirement for proof may be expressed:

$$t = \mathscr{A}(c_1, c_2, \ldots, c_n)\ \{Q_j\}\ \mathscr{A}(c_1, c_2, \ldots, c_n) = f_j(t, v_1, v_2, \ldots, v_{n_j})$$

where t is a variable which does not occur in Q_j. On the basis of this we may say: $t \cdot p_j(a_1, a_2, \ldots, a_n) \equiv t := f_j(t, a_1, a_2, \ldots, a_n)$ with respect to $\mathscr{A}$. This deduction depends on the fact that no Q_j alters or accesses any variables other than c_1, $c_2, \ldots, c_n$; we shall in future assume that this constraint has been observed.

In fact for practical proofs we need a slightly stronger rule, which enables the programmer to give an invariant condition $I(c_1, c_2, \ldots, c_n)$, defining some relationship between the constituent concrete variables, and thus placing a constraint on the possible combinations of values which they may take. Each operation (except initialisation) may assume that I is true when it is first entered; and each operation must in return ensure that it is true on completion.

In the case of smallintset, the correctness of all operations depends on the fact that m remains within the bounds of A, and the correctness of the remove operation is dependent on the fact that the values of $A[1], A[2], \ldots, A[m]$ are all different; a simple expression of this invariant is:

$$\text{size}(\mathscr{A}(m, A)) = m \leqq 100. \qquad (I)$$

One additional complexity will often be required; in general, a procedure body is not prepared to accept arbitrary combinations of values for its parameters, and its correctness therefore depends on satisfaction of some precondition $P(t, a_1, a_2, \ldots, a_n)$ before the procedure is entered. For example, the correctness of the insert procedure depends on the fact that the size of the resulting set is not greater than 100, that is

$$\text{size}(t \cup \{i\}) \leqq 100$$

This precondition (with t replaced by $\mathscr{A}$) may be assumed in the proof of the body of the procedure; but it must accordingly be proved to hold before every call of the procedure.

It is interesting to note that any of the p's that are functions may be permitted to change the values of the c's, on condition that it preserves the truth of the invariant, and also that it preserves unchanged the value of the abstract object $\mathscr{A}$. For example, the function "has" could reorder the elements of A; this might be an advantage if it is expected that membership of some of the members of the set will be tested much more frequently than others. The existence of such a concrete side-effect is wholly invisible to the abstract program. This seems to be a convincing explanation of the phenomenon of "benevolent side-effects", whose existence I was not prepared to admit in [8].

7. Proof of Smallintset

The proof may be split into four parts, corresponding to the four parts of the class declaration:

7.1. Initialisation

What we must prove is that after initialisation the abstract set is empty and that the invariant I is true:

$$\textbf{true}\ \{m := 0\}\ \{i \mid \exists k (1 \leqq k \leqq m \,\&\, A[k] = i)\} = \{\}$$
$$\&\ \text{size}(\mathscr{A}(m, a)) = m \leqq 100$$

Using the rule of assignment, this depends on the obvious truth of the lemma

$$\{i \mid \exists k (1 \leqq k \leqq 0 \,\&\, A[k] = i\} = \{\} \,\&\, \text{size}(\{\}) = 0 \leqq 100$$

7.2. Has

What we must prove is

$$\mathscr{A}(m, A) = k \,\&\, I\,\{Q_{\text{has}}\}\,\mathscr{A}(m, A) = k \,\&\, I \,\&\, \text{has} = i \in \mathscr{A}(m, A)$$

where Q_{has} is the body of has. Since Q_{has} does not change the value of m or A, the truth of the first two assertions on the right hand side follows directly from their truth beforehand. The invariant of the loop inside Q_{has} is:

$$j \leqq m \,\&\, \text{has} = i \in \mathscr{A}(j, A)$$

as may be verified by a proof of the lemma:

$$\begin{aligned} & j < m \,\&\, j \leqq m \,\&\, \text{has} = i \in \mathscr{A}(j; A) \\ \supset\ & \textbf{if } A[j+1] = i \textbf{ then } (\textbf{true} = i \in \mathscr{A}(m, A)) \\ & \qquad \textbf{else } \text{has} = i \in \mathscr{A}(j+1, A). \end{aligned}$$

Since the final value of j is m, the truth of the desired result follows directly from the invariant; and since the "initial" value of j is zero, we only need the obvious lemma

$$\textbf{false} = i \in \mathscr{A}(0, A)$$

7.3. Insert

What we must prove is:

$$P \,\&\, \mathscr{A}(m, A) = k \,\&\, I\{Q_{\text{insert}}\}\mathscr{A}(m, A) = (k \cup \{i\}) \,\&\, I,$$

where $P \equiv_{\text{df}} \text{size}\left(\mathscr{A}(m, A) \cup \{i\}\right) \leqq 100$.

The invariant of the loop is:

$$P \,\&\, \mathscr{A}(m, A) = k \,\&\, I \,\&\, i \tilde{\in} \mathscr{A}(j, A) \,\&\, 0 \leqq j \leqq m \tag{6}$$

as may be verified by the proof of the lemma

$$\begin{aligned} & \mathscr{A}(m, A) = k \,\&\, i \tilde{\in} \mathscr{A}(j, A) \,\&\, 0 \leqq j \leqq m \,\&\, j < m \supset \\ & \qquad \textbf{if } A[j+1] = i \textbf{ then } \mathscr{A}(m, A) = (k \cup \{i\}) \\ & \qquad\qquad \textbf{else } 0 \leqq j+1 \leqq m \,\&\, i \tilde{\in} \mathscr{A}(j+1, A) \end{aligned}$$

(The invariance of $P \,\&\, \mathscr{A}(m, A) = k \,\&\, I$ follows from the fact that the loop does not change the values of m or A). That (6) is true before the loop follows from $i \tilde{\in} \mathscr{A}(0, A)$.

We must now prove that the truth of (6), together with $j = m$ at the end of the loop, is adequate to ensure the required final condition. This depends on proof of the lemma

$$j = m \,\&\, (6) \cup \mathscr{A}(m+1, A') = (k \cup \{i\}) \,\&\, \text{size}\left(\mathscr{A}(m+1, A')\right) = m+1 \leqq 100$$

where $A' = (A, m+1 : i)$ is the new value of A after assignment of i to $A[m+1]$.

7.4. Remove

What we must prove is

$$\mathscr{A}(m, A) = k \,\&\, I\{Q_{\text{remove}}\}\mathscr{A}(m, A) = (k \cap \neg\{i\}) \,\&\, I.$$

The details of the proof are complex. Since they add nothing more to the purpose of this paper, they will be omitted.

8. Formalities

Let T be a class declared as shown in Section 2, and let $\mathscr{A}$, I, P_j, f_j be formulae as explained in Section 6 (free variable lists are omitted where convenient). Suppose also that the following $m+1$ theorems have been proved:

$$\textbf{true}\ \{Q\}I\,\&\,\mathscr{A}=d_0 \tag{7}$$

$$\mathscr{A}=t\,\&\,I\,\&\,P_j(t)\,\{Q_j\}I\,\&\,\mathscr{A}=f_j(t) \quad \text{for procedure bodies } Q_j \tag{8}$$

$$\mathscr{A}=t\,\&\,I\,\&\,P_j(t)\,\{Q_j\}I\,\&\,\mathscr{A}=t\,\&\,p_j=f_j(t) \quad \text{for function bodies } Q_j. \tag{9}$$

In this section we show that the proof of these theorems is a sufficient condition for the correctness of the data representation, in the sense explained in Section 5.

Let X be a program beginning with a declaration of a variable t of an abstract type, and initialising it to d_0. The subsequent operations on this variable are of the form

(1) $t:=f_j(t, a_1, a_2, \ldots, a_{n_j})$ if Q_j is a procedure

(2) $f_j(t, a_1, a_2, \ldots, a_{n_j})$ if Q_j is a function.

Suppose also that $P_j(t, a_1, a_2, \ldots, a_{n_j})$ has been proved **true** before each such operation.

Let X' be a program formed from X by replacements described in Section 4, as well as the following (see Section 5):

(1) initialisation $t:=d_0$ replaced by Q'

(2) $t:=f_j(t, a_1, a_2, \ldots, a_{n_j})$ replaced by $t\cdot p_j\,(a_1, a_2, \ldots, a_{n_j})$

(4) $f_j(t, a_1, a_2, \ldots, a_{n_j})$ by $t\cdot p_j(a_1, a_2, \ldots, a_{n_j})$.

Theorem. Under conditions described above, if X and X' both terminate, the value of t on termination of X will be $\mathscr{A}(c_1, c_2, \ldots, c_n)$, where $c_1, c_2, \ldots, c_n$ are the values of these variables on termination of X'.

Corollary. If $R(t)$ has been proved true on termination of X, $R(\mathscr{A})$ will be true on termination of X'.

Proof. Consider the sequence S of operations on t executed during the computation of X, and let S' be the sequence of subcomputations of X' arising from execution of the procedure calls which have replaced the corresponding operations on t in X. We will prove that there is a close elementwise correspondence between the two sequences, and that

(a) each item of S' is the very procedure statement which replaced the corresponding operation in S.

(b) the values of all variables (and hence also the actual parameters) which are common to both "programs" are the same after each operation.

(c) the invariant I is true between successive items of S'.

(d) if the operations are function calls, their results in both sequences are the same.

(e) and if they are procedure calls (or the initialisation) the value of t immediately after the operation in S is given by $\mathscr{A}$, as applied to the values of c_1, $c_2, \ldots, c_n$ after the corresponding operation in S'.

It is this last fact, applied to the last item of the two sequences, that establishes the truth of the theorem.

The proof is by induction on the position of an item in S.

(1) *Basis.* Consider its first item of S, $t:=d_0$. Since X and X' are identical up to this point, the first item of S' must be the subcomputation of the procedure Q which replaced it, proving (a). By (7), I is true after Q in S', and also $\mathscr{A}=d_0$, proving (c) and (e). (d) is not relevant. Q is not allowed to change any non-local variable, proving (b).

(2) *Induction step.* We may assume that conditions (a) to (e) hold immediately after the $(n-1)$-th item of S and S', and we establish that they are true after the n-th. Since the value of all other variables (and the result, if a function) were the same after the previous operation in both sequences, the subsequent course of the computation must also be the same until the very next point at which X' differs from X. This establishes (a) and (b). Since the only permitted changes to the values of $t \cdot c_1, t \cdot c_2, \ldots, t \cdot c_n$ occur in the subcomputations of S', and I contains no other variables, the truth of I after the previous subcomputation proves that it is true before the next. Since S contains *all* operations on t, the value of t is the same before the n-th as it was after the $(n-1)$-th operation, and it is still equal to $\mathscr{A}$. It is given as proved that the appropriate $P_j(t)$ is true before each call of f_j in S. Thus we have established that $\mathscr{A}=t\&I\&P_j(t)$ is true before the operation in S'. From (8) or (9) the truth of (c), (d), (e) follows immediately. (b) follows from the fact that the assignment in S changes the value of no other variable besides t; and similarly, Q_j is not permitted to change the value of any variable other than $t \cdot c_1, t \cdot c_2, \ldots, t \cdot c_n$.

This proof has been an informal demonstration of a fairly obvious theorem. Its main interest has been to show the necessity for certain restrictive conditions placed on class declarations. Fortunately these restrictions are formulated as scope rules, which can be rigorously checked at compile time.

9. Extensions

The exposition of the previous sections deals only with the simplest cases of the Simula 67 class concept; nevertheless, it would seem adequate to cover a wide range of practical data representations. In this section we consider the possibility of further extensions, roughly in order of sophistication.

9.1. Class Parameters

It is often useful to permit a class to have formal parameters which can be replaced by different actual parameters whenever the class is used in a declaration. These parameters may influence the method of representation, or the identity

of the initial value, or both. In the case of smallintset, the usefulness of the definition could be enhanced if the maximum size of the set is a parameter, rather than being fixed at 100.

9.2. Dynamic Object Generation

In Simula 67, the value of a variable c of class C may be reinitialised by an assignment:

$$c := \textbf{new}\ C\ \langle\text{actual parameter part}\rangle;$$

This presents no extra difficulty for proofs.

9.3. Remote Identification

In many cases, a local concrete variable of a class has a meaningful interpretation in the abstract space. For example, the variable m of smallintset always stands for the size of the set. If the main program needs to test the size of the set, it would be possible to make this accessible by writing a function

integer procedure size; size:=m;

But it would be simpler and more convenient to make the variable more directly accessible by a compound identifier, perhaps by declaring it

public integer m;

The proof technique would specify that

$$m = \text{size}\ (\mathscr{A}(m, A))$$

is part of the invariant of the class.

9.4. Class Concatenation

The basic mechanism for representing sets by arrays can be applied to sets with members of type or class other than just integers. It would therefore be useful to have a method of defining a class "smallset", which can then be used to construct other classes such as "smallrealset" or "smallcarset", where "car" is another class. In SIMULA 67, this effect can be achieved by the class/subclass and virtual mechanisms.

9.5. Recursive Class Declaration

In Simula 67, the parameters of a class, or of a local procedure of the class, and even the local variables of a class, may be declared as belonging to that very same class. This permits the construction of lists and trees, and their processing by recursive procedure activation. In proving the correctness of such a class, it will be necessary to assume the correctness of all "recursive" operations in the proofs of the bodies of the procedures. In the implementation of recursive classes, it will be necessary to represent variables by a null pointer (**none**) or by the *address* of their value, rather than by direct inclusion of space for their

values in block workspace of the block to which they are local. The reason for this is that the amount of space occupied by a value of recursively defined type cannot be determined at compile time.

It is worthy of note that the proof-technique recommended above is valid only if the data structure is "well-grounded" in the sense that it is a pure tree, without cycles and without convergence of branches. The restrictions suggested in this paper make it impossible for local variables of a class to be updated except by the body of a procedure local to that very same activation of the class; and I believe that this will effectively prevent the construction of structures which are not well-grounded, provided that assignment is implemented by copying the complete value, not just the address.

I am deeply indebted to Doug Ross and to all authors of referenced works. Indeed, the material of this paper represents little more than my belated understanding and formalisation of their original work.

References

1. Wirth, N.: The development of programs by stepwise refinement Comm. ACM. **14**, 221–227 (1971).
2. Dijkstra, E. W.: Notes on structured programming. In Structured Programming. Academic Press (1972).
3. Hoare, C. A. R.: Notes on data structuring. *Ibid.*
4. Dahl, O.-J.: Hierachical program structures. *Ibid.*
5. Milner, R.: An algebraic definition of simulation between programs. CS 205 Stanford University, February 1971.
6. Dijkstra, E. W.: A constructive approach to the problem of program correctness. BIT. **8**, 174–186 (1968).
7. Dahl, O.-J., Myhrhaug, B., Nygaard, K.: The SIMULA 67 common base language. Norwegian Computing Center, Oslo, Publication No. S-22, 1970.
8. Hoare, C. A. R.: An axiomatic approach to computer programming. Comm. ACM. **12**, 576–580, 583 (1969).

Prof. C. A. R. Hoare
Computer Science
The Queen's University of Belfast
Belfast BT 71 NN
Northern Ireland