Primitive Data Types

By Arthur H. J. Sale*

This tutorial paper is addressed primarily to practising programmers, and only secondarily to language designers, compiler writers and machine designers. It has two purposes:

* To bring some of the concepts of primitive (indivisible) data-types to the attention of the industry-at-large showing how our understanding of the fundamental processes and objects of programming is gradually improving, with some insights of my own, and
* To emphasize that programmers ought to think in terms of well-structured concepts, and that when and if one of the less attractive and archaic computing languages has to be used, then it is better to translate thoughts into the concrete forms required leaving plenty of traces of the original intentions. In other words, do not write *in* a language; think and then code *into* it.

Keywords and Phrases: data-types, typing, programming methodology
CR Categories: 4.20, 4.9, 5.24

"The goal of science is order. Science is constructed from facts just as a house is constructed from stones, but an accumulation of facts is no more a science than a pile of stones is a house."

Henri Poincare

1. THE CONCEPT OF DATA-TYPE

The concept of *type* is familiar to mathematicians, programmers and the man-in-the-street, albeit in a somewhat fuzzy way. A type is a concept which defines a collection of objects (possibly infinite in number) that exhibits a set of important common properties, and has a set of standard manipulations defined upon it.

Simple examples from everyday life are:

* tree: properties of size, leaves, trunk, etc; defined manipulations of growth, planting, cutting down, furnishing shade, etc.
* chair: object for placing on horizontal surface and interposing between surface and human posterior; defined manipulations of sitting, rising.

These are rather complex sets of objects (complicated types) and would require much work to fully represent in a computer. A mathematician's ideas are somewhat simpler:

* function: there are a lot of functions possible, defined as a mapping from some parameter values to a target value.
* set: primary manipulations are insertion and removal from a set and testing for presence.
* number: of some kind, with defined arithmetic operations of the usual sort.

Some of these are still rather complex, but we can begin to see some familiar things, as for example when a mathematician writes:

Let f be a real function of two real variables, or
let i be an integer in the range 1 to 100.

In these two examples things are happening: (1) the particular data-type is defined (as for example the set of integer numbers in the range 1 to 100) as a particular case of some more general set, and (2) a name is associated with the type (in the above case i is a variable name which can have the desired values).

All programmers are familiar with some examples of data-types, for example in the FORTRAN declarations:

```
INTEGER I, J, COUNT
REAL X, Y, DELTAX, MAXMUM
LOGICAL ERRFLG
```

which declare the listed names to be of one of the standard types INTEGER, REAL or LOGICAL of FORTRAN. Such standard types are best thought of as *pre-defined types* of a language: types whose utility is so great, and whose forms are so standardized that the particular language provides them as a regular feature. In FORTRAN the predefined types are of course INTEGER (integral numeric values), REAL (approximations to arbitrary numeric values), DOUBLE PRECISION (as for REAL but more precise), COMPLEX (approximations to mathematician's "complex" values), and LOGICAL (TRUE or FALSE). There are only these types and the programmer can have no others. (New ANS FORTRAN will add a character type.)

Nevertheless, programmers need to be able to define objects which are different from this restricted set; some simpler, some more complex. FORTRAN for example does permit some more complex objects: remember the array or vector:

```
INTEGER TABLE (132), TEMP (10)
```

and indeed the array is one of the simpler and more common data-objects that can be defined by a programmer.

It is useful to make a distinction between objects which are only altered in toto (for example assigning a new value to an integer destroys all of the old value) and those which are modified in part (as in changing one element of an array). I shall in future call objects of the first kind *data-types* of the language, problem, machine or whatever, and objects of the second kind *data-structures* of the language, problem, machine, etc. It must be pointed out that this naming convention is not universal, though it accords well with most usage. Further, the boundary between the two things may change between languages: for example arrays in FORTRAN are clearly data-structures, whereas in some other languages they partake of the nature of full data-types (for example *APL*).

Concentrating for the moment on data-types, we can assume that they have the following properties:

* a data-type determines the class of values that may be assumed by a variable or expression,
* the type of variable or expression is determined by the programmer, and should be determinable by a translator, without any explicit knowledge of the sequence of values the variable or expression takes on in execution,
* associated with a type is a set of *operators* (which expect operands of fixed types and deliver results of

* *Department of Information Science, University of Tasmania.* *Manuscript received 3rd November 1976, revised version 24th March 1977*

fixed type),
* the properties of the type and its associated operators are determined by a set of *axioms* (read *rules* if you prefer), and
* type information is used in programming to (1) prevent meaningless constructions or allow of their detection, (2) delay or ignore implementation problems of representation, and (3) eventually to determine appropriate implementation on a computer.

In looking at some simple data-types we shall therefore be concerned to determine the class of values associated with the type, how they are manipulated (the defined operations), how they may be implemented, and what happens when the boundary assumptions of the axioms are violated. We start with *unordered types* and *enumerated types*; the very simplest of the primitive data types.

2. NOTATION

In what follows, I shall adopt a flexible notation to describe declarations of type and programming examples, rather than sticking to stultifying restrictions of FORTRAN or other languages. This corresponds to no actual programming language (though it could easily be implemented). PASCAL is the nearest equivalent. Examples in this language are printed in lower case; to simplify your understanding the reserved words (those with fixed meanings) are in boldface to distinguish them from programmer-chosen names.

You will notice later that declarations are turned end-for-end as compared with FORTRAN and Algol (though similarly to COBOL or PL/I). There is no hidden significance to this except that it reads a lot more logically, especially with the more complex declarations we shall look at.

I shall need the occasional example in a widely available computing language (FORTRAN, PL/I, PASCAL) and such examples may be easily spotted; they are in upper-case. I shall not however restrict myself to FORTRAN's ridiculous six-character names, so as not to compromise readability.

3. UNORDERED DATA-TYPES

For our first foray into data-types let us look at sets of values which have no inherent ordering relation between them (we may force one to exist, but that is another story). A very simple example is afforded by the FORTRAN type LOGICAL: a logical expression can take on one of the two values **false** or **true**, and these alone. No order can be discerned between **false** and **true**; one is not 'bigger' than the other, and thus we have here a good example of a *two-valued unordered data-type*. No doubt with this clue you can think of many other types of this kind. To continue our discussion, I introduce a notation to describe such types which looks as follows:

declare type logical **as unordered** (false, true);

or to take another example:

declare type sex **as unordered** (male, female).

Note that fixed-meaning words are in boldface to distinguish them from chosen names only for your convenience in making sense of the statement.

But unordered types with more values can also exist; any familiarity with tax forms will enable you to think of:

declare type maritalstatus **as unordered** (single, married, divorced, separated, widowed);

or from other fields;

declare type primarycolor **as unordered** (red, blue, yellow);

declare type gender **as unordered** (masculine, feminine, neuter).

In fact, if you come to think of it, unordered types are probably one of the most common things encountered in filling in forms! Notice how above we have given a name to the type (to take an example: *gender*) and that it will be conceptually possible to be able to declare variables of that type, perhaps by writing something like:

declare variables nounform, verbform **of** gender

Note too that such variables may only take on the values of *masculine, feminine* or *neuter* (whatever such values may be); not numbers, not anything else, just *masculine, feminine* or *neuter* . . .

So much for the associated values; what about the associated operations?

Well, the most obvious operations are those of *assignment* or *replacement*: to be able to transfer a value into a variable. The sort of effect we want can be described something like this:

declare type colour **as unordered** (red, green, yellow, blue);
declare variables light, illumination **of** colour;
. . . .
light:=green;
illumination:=light;
. . . .

In the above we assigned a (constant) value to the variable *light* and later we transferred the value held by it into the variable *illumination*. No errors are possible in these operations provided the value to be assigned is of the correct type (here *colour*).

The next operation that may be useful is the test for equality of two values. We might want to be able to write things like:

if (light=red) **then** watchfordrunks;
if (light **notequalto** illumination) **then** sunriseprocedure;

Again, quite straightforward. Extend the idea somewhat to permit testing of values in a selection:

```
with light select from
     case red:      executeastop;
     case yellow:   slowdown;
     case green:    keeponthroughintersection;
     else:          exerciseextremecaution
```

And the remaining useful thing we can do with unordered types is to sequence through all possible type values (of course in any arbitrary order). A possible use would be:

```
for i:=all colour do
     spotlight_i:=off;
```

presuming that we have a set of four spotlights: $spotlight_{red}$, $spotlight_{green}$, etc. each of some type (on, off).

You will notice that in principle all the above manipulations can be checked by the programmer at program writing time, and by the compiler at translation time (if a suitable compiler exists) to check perfectly that nothing could go wrong with unordered data-type computations. Since all we can do with unordered data-types is to generate correct values and shuffle them around, no type-handling errors are possible! Delightful! In practice though, very few languages are that good, and the programmer has to do more work than he ought by working out some nitty-gritty details, and opening the door wide to slips and mistakes . . .

If we now look at how we might store information of this type in a computer, the first thing to remember is that we have to store the information in a pattern of bits, and further, most computers give us cells (words) of fixed numbers of bits to work with. Only the first constraint is really important here for if each computer word has n bits and there are m values in an unordered type, then provided

$2^n \geqslant m$

then we can choose any distinct word-patterns to represent the values of our unordered type. To take an example, the type *logical* exists in many languages and machines, and requires representation of two values: *true* and *false*. I know of at least one instance in each of the following cases where the language designers chose the implemented patterns of *true* and *false* to be:

true	false
0000. .00	1000. .00
1111. .11	0000. .00
0000. .01	0000. .00

The choice is obviously arbitrary in the absence of further information. In actuality, the above cases were influenced by minor and peculiar properties of the particular computer's instruction manipulations. It is interesting however to impose an extra constraint to see its consequences: how can we represent these values using the least number of bits?

For m values we are going to need $j = \lceil \log_2 m \rceil$ bits, so that $2^{j-1} < m \leqslant 2^j$. (Please read the notation $\lceil x \rceil$ as 'the ceiling of x' and understand it to mean the integer at least equal to or greater than x.) This guarantees that we have enough variations available to represent all values, and yet we cannot dispense with one more bit, as we would not have enough. If $m = 2^j$ then we must simply assign the values to all the patterns, if $m < 2^j$ then we have some spares and it is appropriate (though not essential) to choose our represented values in some logically compact way. To take a concrete example, the type *colour* we worked with above has $m = 4$, so that $j = 2$ and an appropriate value representation would be:

red = 00
green = 01
yellow = 10
blue = 11

while for type *gender* $m = 3$, $j = 2$, and the following is possible:

masculine = 00
feminine = 01
neuter = 10
11 is not used.

4. LOGICAL TYPE

Although in the previous section I introduced unordered data types through type logical as it exists in FORTRAN or Algol, this is a slightly misleading example. Logical values do have all the properties of unordered types, but they also have a few more: the logical operations **and**, **or** and **not** (and possibly more) are pre-defined for logical expressions.

This brings up an important point: no classification I or anyone else can device can possibly cover all representable objects or values. All we can do is to point to the important classes of objects, and decide which properties of the things we want to represent should be embedded in a type. Type logical is of course very important: it warrants the special treatment it gets.

To re-iterate the points made before might bring the discussion into focus: logical type is an unordered data-type of two values; it is predefined and has operators predefined especially for it; and the constants of the type are denoted by the character sequences "true" and "false".

5. ENUMERATIVE DATA-TYPES

The next simplest data-type is the *enumerative* type: again a finite set of values (usually small), but here it is appropriate to recognise a *linear ordering* in the set of values. In a modification of the notation I used earlier, I shall omit the word **unordered** in favour of **ordered** and let the ordering relation be specified by the order in which the values appear enumerated in the list. In other words, if we read the sign '<' as meaning 'comes before' then in the type declarations:

declare type floorlevel **as ordered** (basement, ground, mezzanine, top);

then we have the relations:

basement < ground < mezzanine < top.

Other examples are:

declare type cardrank **as ordered** (two, three, four, five, six, seven, eight, nine, ten, Jack, Queen, King, Ace);

declare type romannumeral **as ordered** (I, V, X, L, C, D, M)

The manipulations of enumerative types are the same as for unordered types with two additions made possible by the existence of a defined order between the values: testing the ordering relation, and moving up and down the order. The first is easily shown by the following program fragment:

declare variable actualcard **of** cardrank;
. . . .
if (actualcard < Jack) **then** . . . .;

while the second required us to define a manipulating instruction or two. Suppose that we have two such manipulations possible.

succ(x)
pred(x)

which give a value which is the *successor* or *predecessor* respectively of the value of x in the type sequence to which x belongs. You can see the use of this in:

if (actualcard **notequal** Ace) **then** winningcard:=succ (actualcard);

This example also points out a potential disaster: what is succ (Ace)? or pred (two)? and what will happen if the unwary programmer sets up a statement that causes either of these to be taken?

Of course the only appropriate answer is that such operations are meaningless, and programs that attempt to perform them are incorrect. Since however the occurrence of such a disaster cannot always be detected by inspection of the source text, then some error-handling must occur during execution-time: while the program is running. For example the piece of program that specifies how succ(x) is to be found must have incorporated in it a test for x being at the limit, and cause termination of the program or some precautionary message if it is. Alas, for few languages make any such provision, and if like FORTRAN or PL/I they do not even recognise such types all responsibility is thrown back on the programmer (who is generally quite unreliable in this regard).

The conventional way to implement enumerative data-types is also the most compact. The listed values are mapped onto the integers 0, 1, 2, . . . or in actuality onto

the bit-patterns which correspond to these in the computer. In this way the ordering property of integers is used to preserve the order of the list. Of course, once again, at least $\lceil \log_2 m \rceil$ bits are necessary to represent a set of m values. Example:

two	0000	ten	1000
three	0001	Jack	1001
four	0010	Queen	1010
five	0011	King	1011
six	0100	Ace	1100
seven	0101		
eight	0110		
nine	0111		

You may also be tempted to assign instead the values 2, 3, 4, . . . to *two, three, four*, etc. (it may seem more 'natural' to you in this case), but note that the value of *Ace* must still be greater than any other (and should not be 1). Or to take another case the type *romannumeral* could be represented by 0 to 6, or by the integers 1, 5, 10, 50, 100, 500 and 1000. The choice is arbitrary in the absence of further constraints provided that the ordering is preserved.

Again, the common languages have not yet appropriate mechanisms to handle such types, and we must resort to some work to implement them.

6. CIRCULARLY ORDERED DATA-TYPES

In a common variation on the theme of ordering, a *circularly ordered data-type* can be encountered. Superficially this resembles a simple (linearly) ordered data type discussed in the previous section, but in addition the ordering relation is also defined between the first and last elements of the list. A good example is afforded by the days of the week:

declare type weekday **as circularly ordered** (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday);

which defines the ordering relations:

Monday < Tuesday < Wednesday < Thursday < Friday < Saturday < Sunday < Monday . . . . . .

You can see from this description that if we define the succ(x) and pred(x) operators on this type, then

succ(Sunday) is Monday, and
pred(Monday) is Sunday,

so no type errors are possible! On the other hand it is no longer possible to ask for conditions which test the ordering relation between members of the set which are not neighbours: it is a meaningless question to ask if Tuesday comes before Sunday . . .

Some confusion exists between linear and circular orderings in the literature at present, and the difference is not often made. However the implications of the difference in the *succ* and *pred* operators is important as will transpire later.

Implementation of circularly ordered types is usually the same as for linearly ordered types: the integers 0 to (m−1) are used, and either an explicit test in the *succ* and *pred* operator coding or a computation modulo m is used to keep the values in that range.

7. PRAGMATICS

If you have a compiler for PASCAL available to you, much of what I have said so far is available to you, albeit in a form which is not quite as regular or comprehensive as I have been discussing.

If not, then you are in the common situation of most people and the entire responsibility of typing rests with you, the programmer, and not with the compiler or the system. In these circumstances, it is best to *think* in terms of the data-types I have been talking about, but to (i) modify your coding habits to reflect your better awareness of what you do, and (ii) leave clear commentary trails to explain how the language influences what you have to code. In most cases the small finite types are best mapped onto an integer type, though logical can be used if there are only two values in the type.

To illustrate what can be done, the following examples show

* a PASCAL program fragment (Fig. 1),
* how the macro-facility (DEFINE) in Burroughs B6700 Algol can be used to construct partial types (Fig. 2) and
* a possible way to construct and document the ideas of type FORTRAN and PL/I (Fig. 3).

```
PROGRAM STAGELIGHTING;
TYPE
     COLOUR = (RED, GREEN, BLUE, WHITE);
     SWITCH = (ON, OFF);
VAR
     WHICHSPOT   :  INTEGER;
     SPOTSWITCH  :  ARRAY [1..10] OF SWITCH;
     FLOODSWITCH:   SWITCH;
     FLOODCOLOUR:   COLOUR;
BEGIN
     FLOODCOLOUR:=RED;
     FLOODSWITCH:=ON;
     FOR WHICHSPOT:=1 TO 10 DO BEGIN
        SPOTSWITCH[WHICHSPOT]:=OFF;
     END;
     ....
     ....
END.
```

Figure 1: An incomplete PASCAL program.

8. CHARACTERS

One particular example of a simple data-type is very important; important enough to warrant special examination. This example is that of *characters* such as a,b, ... A,B, ... 6,7, ... etc. Since most communication with computers at the present state of the art is by the written word (worse: the typed word) characters loom large in this communication process. Historically, character sets were relatively restricted things: one set of the alphabetic characters, the decimal digits and a handful of others. That makes 26+10+11 (say) therefore requiring $\lceil \log_2 47 \rceil = 6$ bits per character, and indeed most early computers had six-bit characters. The situation persists, and these are still to be seen, together with printers, etc. with only one case of alphabetic characters (usually upper-case). Truly programmers must be the most proficient people in the world in reading block capitals! The trend nowadays is slowly towards bigger and more useful character sets and seven- and eight-bit codes are becoming commonplace, especially for data communication traffic.

```
%-------------------------------------
% DEFINE (IN MACROS) A TYPE -COLOUR-
%-------------------------------------
DEFINE
   COLOUR      = INTEGER#,

   RED         = Ø#,
   GREEN       = 1#,
   YELLOW      = 2#,
   BLUE        = 3#;

%-------------------------------------
% VARIABLES OF TYPE COLOUR
%-------------------------------------
COLOUR LIGHT,ILLUMINATION;

LIGHT:=RED;
```

Figure 2: Macro-definitions in Burroughs B6700 Algol.

room for variation. (Is $9 < A$ or $Z < 0$? Does A to Z form a compact group?) and several perplexing problems (Is $a < A$ or vice versa? We'd like them to be the same for alphabetic ordering reasons ... Where does the space fit in the sequence?). Suffice it to say that none of these are problems that you will have to worry about much. The character set used internally in a particular computer (or in its communications with other machines) is usually standardized and not under your control. The effort of changing it into your own variety is seldom worth the effort.

Character sets may therefore be regarded as a predefined ordered data-type, something like:

declare type character **as ordered** (nul, . . ., . . .,
space, exclamation, doublequote, . . . , . . . , . . . ,
zero, one, two, . . . , . . . ,
A, B, C, D, . . . , . . . , a, b, c, d, . . . , . . . ,
. . . , erase);

9. THE ALMOST-INFINITE TYPES

We have progressed from looking at collections of values which are defined by a programmer listing the values,

```
C-------------------------------------------------
C  FOLLOWING VARIABLES ARE OF A TYPE -ROMANNUMERAL-
C-------------------------------------------------

      INTEGER ROMANCHAR,NEXTROMANCHAR

C-------------------------------------------------
C  THESE ARE THE CONSTANTS OF TYPE -ROMANNUMERAL-
C-------------------------------------------------

      INTEGER I,V,X,L,C,D,M
      DATA I,V,X,L,C,D,M /1,5,1Ø,5Ø,1ØØ,5ØØ,1ØØØ/
```

```
/*  PL/I EXAMPLE */
/*  CONSTANTS OF A TYPE -FLOORLEVEL-  */
DECLARE BASEMENT   BINARY FIXED INITIAL Ø;
        GROUND     BINARY FIXED INITIAL 1;
        MEZZANINE  BINARY FIXED INITIAL 2;
        FIRST      BINARY FIXED INITIAL 3;
/*  VARIABLES OF TYPE -FLOORLEVEL-  */
DECLARE DESIREDFLOOR,LIFTPLACE,ENTRYPORT BINARY FIXED;
```

Figure 3: FORTRAN and PL/I substitutes for type declarations.

What kind of data-type is a character set? To start with we know that most people associate an ordering relation with the alphabet from A to Z, and there is a strong ordering of the digits from 0 to 9 (not 1234567890!). Consequently there is strong pressure to regard a character set as an ordered type to preserve these relations: I know of no character set of computer-age vintage that does not preserve them, at least as used internally in the computer. Having said this however we have still left considerable

through some pre-defined types to type **character**. All these have the characteristic that the possible values have a very small finite cardinality: from 2 for type **logical** to 256 for an EBCDIC **character** set. The next step is to move on to those common types which are *conceptually infinite* in values, though in practice a computer can only represent a finite subset of the values. I am referring, of course, to the varieties of number.

Some of the properties of numbers are held in common,

and can be discussed here. First, and perhaps most obvious, is that the arithmetic operators **plus, minus, multiply** and **divide** are defined for all numbers that are in common use. These operators produce a result from two values which is of the same type as the values. We also want all the axioms of number manipulations to hold for all operations on computer numbers, and unless there is a common violation of these rules, I shall not comment further on them. A second property is that of ordering: the values of the type are regarded as strictly ordered, and it is meaningful to ask for any ordering relation between two values to be tested. Consequently any of the things that are sensible with enumeratively ordered types are sensible with numbers. The only possible exception comes with the number-types which are conceptually infinitely subdividable, such as type **real**.

10. THE NATURAL NUMBERS AND INTEGERS

It is an amazing fact that almost no high-level languages have seen fit to distinguish between the two number types of denumerably infinite values: *natural numbers* (or unsigned integers) and *integers*. Machine designers have long distinguished, and way back in the mists of antiquity (before FORTRAN) computers had special facilities for dealing with words as unsigned numbers, as against those with signs. Indeed, FORTRAN itself bears many hallmarks to show that its INTEGER type was constrained by the facilities on the IBM704 for handling natural numbers. To take a more modern example; the rash of two's complement 16-bit mini-computers is a direct result of not wishing to have to provide separate instructions for natural and signed numbers. The PDP-11 range concedes the point with a separate set of branch tests . . .

Why the fuss? Well, in actuality most of the applications we casually use integers for are counting operations, or matching operations. The natural numbers:

0, 1, 2, 3, 4, 5, ...

are directly intended for counting and matching in one-for-one correspondences. This is of course why machine addresses are so often mapped onto natural numbers. In such applications a negative value has no sense, and in line with our maxim of *knowing* what we are doing and making it explicit, we ought to recognize that we deal with natural numbers at least as often as (if not more often than) with genuine signed integers. And if so, a good computing language ought to permit you to make the fact explicit. Examples that occur to me from recent experience are variables that select a bit in a computer word, or a disk sector in a code-file, or that determine the maximum number of applications of an interative algorithm, or that rank a record in an ordered collection.

About signed integers there is little that need be said; most programmers are thoroughly familiar with the concept and the behaviour of the type. The signed integers are infinitely extended (in concept anyway) either side of zero:

. . . . . ,–5,–4,–3,–2,–1,0,+1,+2,+3,+4,+5, ...

and the common representations usually give a range which is approximately equal either side of zero.

The standard operations on natural numbers and integers are all the ordering tests ($= \neq < \leqslant > \geqslant$), plus (+), minus (–), multiply (x), and the division operators. Division poses a bit of a quandary, because it is undefined for division by zero (inevitably a run-time error), and because division of a natural number by another, or an integer by another, produces two results instead of the usual one result: a *quotient* and a *remainder*. This faces designers of high-level languages with a dilemma: whether to stick with the usual infix operator notation of conventional mathematics, or to develop a special notation for division. Almost without exception, language designers opt for the first alternative and provide two division operators, one of which gives the quotient result, and the other the remainder result. For example, here are the solutions of two language designers:

B6700 ALGOL	ANSI FORTRAN
Q:=DD DIV DR:	Q = DD / DR
R:=DD MOD DR:	R = MOD(DD,DR)

We are not yet out of the division wood yet, for the question of what is the remainder when one of the operands is negative finds different machine designers taking different views. Altogether an unappetizing mess, and this even in one of the commonest types!

Given all the above operators, and from what we know about representations, it is reasonable to assume that if a programmer writes code involving integer or natural numbers, then the representations and arithmetic are *exactly* analogous to the conceptual properties of the types provided that division by zero does not occur, and provided that no result is computed that is unrepresentable. If either of these actions occur, I at least expect that the programmer be notified of the fact (and possibly the computation terminated as it has become meaningless). Alas, foolish me! There are any number of exceedingly mysterious errors that compilers and machines will allow to happen without warning or notice. While this attitude might have been allowable when hardware was very expensive, it is almost unforgivable in the present situation. The fact that integer overflow is seldom signalled to the operating system (and thence the programmer) is partly a fault of machine designers who have not resolved the conflict between natural numbers and integers, and partly the fault of programmers who have for long taken advantage of minute knowledge of arithmetic to write esoteric assembly code routines. Despite this, I must maintain that a good representation of natural numbers or integers should obey the following properties, as well as the rules of correct arithmetic:

* An attempt to divide by zero should raise an error event, and be notified to the programmer. This is usually true.
* A result which is out of the representable range, whether for integers or natural numbers (as appropriate) should raise an error event and be notified to the programmer. This almost never happens.
* If the representation has a –0 as well as a +0, then from the high-level language programmer's view the two ought to be indistinguishable. One's complement machines are the biggest offenders.
* The results of a division should satisfy the following axioms:

  sign(remainder) = sign(dividend)

  dividend = (quotient x divisor) + remainder

  $0 \leqslant |\text{remainder}| < |\text{divisor}|$

11. THE REAL NUMBERS AND BEYOND

The organization of real numbers is probably of great interest only to a subset of programmers; those dealing with scientific or engineering problems. And yet the issues are of

interest to all, for in representing real numbers our computers are attempting to represent values which are not even denumerably infinite (as are integers) but are more infinite still! Consequently the representation of real numbers must not only be restricted in range like the integers, but also an infinite number of slightly different mathematically real numbers has to be represented by a single computer REAL (or floating-point) number. As a consequence a number of mathematical laws which hold for numbers cease to be true in computer arithmetic; for example

$$((a + b) + c) \quad \text{and} \quad (a + (b + c))$$

are **not** equivalent in computer REAL arithmetic: the two results may be different (perhaps only in some low bits, but different nonetheless).

I cannot afford the time to go into the details of what constitutes a good real number representation apart from saying that any unrepresentable result should raise an event and be notified to the high-level programmer, whether it be overflow or underflow. For various reasons which I do not agree with, some systems signal overflow but not underflow ... If anyone wants to read further on the desirable properties of REAL arithmetic and representation see Knuth (1969) and the excellent critique of the CDC6600 by Wirth (1972). I hasten to add that there are an awful lot of computer architectures quite as bad as that of the CDC6600; I hope machine designers have started to listen.

The last few sections have skipped a very large number of issues, such as why we have floating-point number representation of real numbers at all; why we don't have a representation for the rational numbers (fractions), and so on. Let me finish off by pointing out that in FORTRAN at least we have two further pre-defined types to go: DOUBLE PRECISION and COMPLEX. DOUBLE PRECISION is of course simply a more precise kind of REAL; it usually includes the REAL values as a subset. COMPLEX is a type which holds a pair of REAL values, called (confusingly) the real and imaginary parts of the complex number. The ad-hoc nature of FORTRAN prevents us seeing this in a consistent framework where we might have complex-integers as well as complex-reals and complex-doubles, and only the one variety crops up.

12. SUBSETS AND SUBRANGES

If you have been attentively following the discussion of numbers discussed you will have realized that there often exists a subset relation between different types of numbers: all numbers of the one kind are included in the second kind. This hierarchy for the conceptual number systems is shown in Fig. 4. This leads to the knotty problems of coercions, and of whether an integer ought to be permitted to appear inside a real expression (for example). The present view is that automatic *coercions*, or in other words *default type-changes*, ought to be permitted only if the value to be coerced is of a subset type of the type it is being coerced into. Programmers, please note. If a language you use has automatic coercions in the reverse direction (for example in FORTRAN from REAL to INTEGER across an assignment), it is well-structured programming practice to always make the type transfer explicit, so as to at least document that you were aware of the restriction on values. For example

```
WHOLEDOLLARS=IFIX(HOURLYRATE*HOURS)
```

The situation is bedevilled by yet more quirks of our computers; for example on the Burroughs B6700 and some other computers the conceptual subset relations shown in the figure are true: all natural numbers are exactly representable in the integer range, which in turn are exactly representable in the real range. However in the IBM370 range, to take one example, neither of these subset relations is strictly true. In its 32-bit words there are some natural numbers which cannot be represented as signed integers, and some integers which cannot be exactly represented as real values.

The question of subsetting however brings up another important concept: that of a *subrange*. A subrange is considered by some (for example in the language PASCAL) as a new type with some, but not all, properties of its parent rangetype. It consists of all representable values between two specified limits, and the concept is applicable to all linearly ordered types:

```
declare subrange types
    bargainfloors as subrange of floorlevel (basement to
    mezzanine);
    age as subrange of integer (0 to 150);
    digitchars as subrange of char ("0" to "9");
    normalizevalue as subrange of real (−1.0 to +1.0);
```

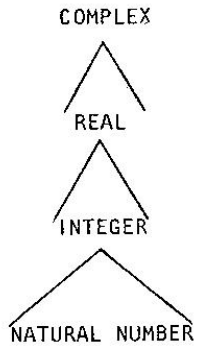

Figure 4: The subset relations of common numbers.

Having a language which permits this sort of declaration allows the programmer to notify and therefore document his intention of restricting the range of values to be taken by a variable to a smaller set of values of the type. The advantages of this facility are threefold, and I shall discuss them in order of importance:

Firstly, such a compiler may be able to economize on storage for the variables concerned if it can determine the size of the subrange. For example the *age* subrange defined above will fit into a single 8-bit byte of a PDP-11 or an IBM 370, instead of a larger integer word. This is a simple space gain and must be matched against the complexity of packing and unpacking the variable.

Secondly, the compiler may generate checking code to determine at critical points whether the values of the subrange variables are in fact within the declared range, possibly under the control of a compiler option. Or it may be able to find some usages at compile-time that are incompatible with the subrange declaration, as in

```
declare variable ageofsamuel of age;
....
ageofsamuel:=−1; % conceived but not yet born
....
```

While the above is a plausible usage, it is not in accord with the declaration, and one or the other needs to be altered. You will realize that compilers that can aid in this task may make a powerful contribution towards debugging, especially if there is a hardware assist for bounds checking at run-time.

And thirdly, a subrange of some types can make them more finite, and therefore usable for different purposes. As we saw earlier, all enumerated types had a rather small number of members (usually less than 256 for example) and consequently constructing jump tables or using them as array indices posed no problems. However integers are almost-infinite, and fitted very awkwardly into such contexts. Taking a subrange of integer will therefore bring the almost-infinite set of values down to a manageable and finite set, as for example in the definition of the subrange *age*. This is not true of real numbers, for which any subrange should conceptually still have an infinite set of values. This conflict of purpose shows up fairly clearly in the language PASCAL, which attempted to introduce subranges in the manner outlined here with partial success.

There is the same message here for programmers using more prosaic languages: again, think in terms of well-structured data-types and in subranges, and document your thoughts as far as possible in standard coding and in commentary. In addition, if you have a macro-facility, consider using it to incorporate subrange-checking code into your program at critical points of assignment. A macro-facility is an extremely powerful tool for extending a language, though not as good as a proper compiler for the extended language. Here for example is a way of carrying out a check using the Burroughs Algol DEFINE facility:

```
DEFINE CHECKAGE(VAL)=
    CHECKINTEGERBOUNDS(VAL, 0, 150)
    % A CHECK PROCEDURE
    % ABOVE REPLACED BY (VAL) WHEN DEBUG
    % COMPLETE
    #;

....
AGEOFSAMUEL:=CHECKAGE(AGEOFSAMUEL + 1);
% BIRTHDAY
....
```

13. SELF-ASSESSMENT EXERCISES

Once upon a time, I would have called these *problems*, but now the trendy phrase is self-assessment quiz. Joking apart, it is quite possible to read a paper, nod your head wisely, and absorb none of it. To help you retain some of the ideas I have tried to put into the paper, and to give you the opportunity to see if you have really understood what was being said, here are a few problems to tackle. They are all fairly easy, and mostly permit different answers. Consequently sample solutions are not provided; if their absence worries you I suggest you discuss them with a colleague. The problems are coded according to the section of the paper they refer to.

3.1 Find a form of some kind, and document how many responses on it are unordered data-types. Write some definition for the types of these responses.

3.2 Two definitions of type *sex* follow. Should a high-level language programmer who can define such types be able to detect any differences between the two definitions or not?

(i) **declare type** sex **as unordered** (male,female);

(ii) **declare type** sex **as unordered** (female,male);

5.1 If you were to set up a data-type to hold values which are the chesspiece names: (pawn, knight, bishop, rook, queen, king), would you choose an unordered data-type or an ordered data-type? Why?

5.2 A tax computation program uses different methods according as to whether a taxpayer's assessable income is less than $5000, from $5001 to $6000, from $6001 to $8000, from $8001 to $12000, from $12001 to $20000, and more than $20000. The programmer wishes to create an ordered type which he can use in defining values of a variable *incomerange* in his program. (1) Write a definition of the type, and (2) create some appropriate declarations and comments in case he writes the program in a language of your choice (FORTRAN, PL/1, COBOL, etc.).

6.1 Think of another circularly ordered type and write a definition for it.

8.1 For EBCDIC and ASCII character codes (and that of your computer, if it is different) determine

(1) the relative ordering of a-z, A-Z, and 0-9 within each code, and

(2) where the character *space* appears in each order. What effect does *space* have upon alphabetic ordering in

(a) a telephone directory, and (b) a dictionary?

8.2 An early code devised for teleprinters (and still used in such systems) uses a five-bit code. Since $2^5 = 32$, this is not enough to even represent the alphabet and the digits! So, a stratagem is employed: two of the codewords are reserved to have special meanings. One when transmitted informs the receiver to interpret all following characters as being from one set of printable and other characters, and can only be reversed by sending the other one which forces a different interpretation on the other codewords. You may be familiar with this concept in the shiftkey of a typewriter. This allows a total of $(32 - 2) \times 2 = 60$ transmittable meanings. What disadvantages can be forecast for this apparently good scheme for getting a quart out of a pint pot?

10.1 If you know the machine-instruction set of your computer, ask yourself if there are instructions which treat machine-objects as natural numbers instead of signed integers. What are they? Are the natural number values wholly included in the integer values?

10.2 Take the last program you wrote and see how many usages of integers were in reality counting operations using natural numbers.

10.3 Does your computer or compiler do anything about integer overflow? If not, why not? Have you complained (you are the consumer)?

12.1 Take the last program you wrote and see how many usages of numbers could have been thought of as subrange objects, or enumerative data-types.

14. ACKNOWLEDGEMENT

The preceding tutorial paper is heavily based on the work of C.A.R. Hoare on data-types, and covers some of the initial parts of the section by him in the first reference. This work also covers structuring techniques for creating new types (discriminated unions, powersets, etc.) which are not discussed in this paper.

The programming language PASCAL embodies a number of these ideas, though it does not go as far as I suggest in

the paper. Being designed with better concepts of data-typing than older languages, it is clearly superior in conceptualizing type, and is well worth study by interested readers regardless of its other features. The best-known implementation is PASCAL-6000 for CDC 6000-series and Cyber-range computers.

Unfortunately, most programmers at present working in the industry are constrained by other forces to use less sympathetic languages. This tutorial is an appeal to these programmers to modify their thought-habits (at least) so as to embody better-structured data-typing into these more restrictive forms. Even in PASCAL, programmers should not write *in* a language; rather think and then code *into* it.

15. SUGGESTIONS FOR FURTHER READING

DAHL, O.-J., DIJKSTRA, E.W. & HOARE, C.A.R. (1972): "Structured programming", *Academic Press*. (Especially Hoare's section on data structuring.)

HOARE, C.A.R. (1969): "An axiomatic basis for computer programming", *Comm. ACM 12*, pp. 576-581.

JENSEN, K. & WIRTH, N. (1974): "PASCAL User Manual and Report", *Springer-Verlag*, Lecture Notes in Computer Science 18. (Describes PASCAL and the defining report.)

KNUTH, D.E. (1969): "The Art of Computer Programming, Volume 2: Seminumerical Algorithms", *Addison-Wesley*, pp. 161-248.

WIRTH, N. (1976): "Algorithms + Data Structures = Programs", *Prentice-Hall*. (Takes up where this article leaves off in discussing more complex structures from Wirth's viewpoint.)

WIRTH, N. (1972): "On 'PASCAL'. Code Generation, and the CDC6000 Computer", *Stanford University Report STAN-CS-72-257.*