SET: A neglected data type, and its compilation for the ARM

Niklaus Wirth

8. 8. 2007

Although the data type SET had been introduced in 1970 in Pascal and retained in Modula and Oberon, it seems to remain little appreciated. The idea stemmed originally form C.A.R. Hoare out of an attempt to raise types like Bits and Bitset onto a higher level of mathematically appealing abstraction. The key idea was to regard the bits of a computer word as a set of integers, namely the numbers of the one bits. In Pascal, the declaration of a set type explicitly specified the type of the set's elements. It is typically an enumeration or a subrange type. For Oberon, the set type was chosen to be not generic, but that the elements were fixed to small integers ranging from 0 to *n*, the wordsize. Examples with *n* = 8 are the following:

0	00000000	{}
1	00000001	{0}
5	00000101	{0, 2}
55H	01010101	{0, 2, 4, 6}

The disregard of most programmers for this type may be due to the restriction of elements to small integers. But more so, I suspect, because programmers fear a heavy mechanism to lie behind the set concept, possibly involving dynamic, linked data structures, resulting in a slow implementation. One must be aware that SET means "Set of small integers".

But the set concept does occur, even if not expressed in the proposed notation. A well-known example are status and command registers of device interfaces, where individual bits have their specific, independent meanings. Here, a more natural formulation would be highly desirable, rather than an "encoding" as an integer. Another example is a sequence of Boolean terms, for example

```
(n = 2) OR (n = 3) OR (n = 5) OR (n = 7) OR (n = 11) OR (n = 13) OR (n = 17)
```

which is much more simply expressed as

```
n IN {2, 3, 5, 7, 11, 13, 17}
```

This note is intended to encourage (Oberon) programmers to make use of this data type, and to show that it is not only convenient but also highly efficient.

In the language Oberon, a value of type SET is represented as a word of 32 bits. The operators are set union (+), intersection (*), difference (-), and symmetric difference (/). There is set complementation (unary minus), and the relation of equality and set inclusion. A simple example will reveal that a very efficient implementation is quite obvious. Let s0 = {0, 1} and s1 = (0, 2}. Then

s0 + s1 = {0, 1, 2}	s0 – s1 = {1}
s0 * s1 = {0}	s0 / s1 = {1, 2}

Evidently, the set operations are to be represented by simple and fast logical operations available in every computer, namely

union	or	0011 or 0101 = 0111
intersection	and	0011 and 0101 = 0001
difference	and not (bic)	0011 and 1010 = 0010
sym. difference	xor	0011 xor 0101 = 0110

Set complementation is implemented by a *not* instruction, and inclusion follows from

$$s0 \subseteq s1 \equiv s0 - s1 = \{\}$$

Accordingly, the compiled code for the ARM processor and the following module consists of only a few instruction. Note that here we make use of the facility to have variables allocated in registers (here s0 in R0, s1 in R1, and s2 in R2).

```
MODULE M;
   PROCEDURE* P;
      VAR s0, s1, s2: SET;
   BEGIN s0 := s1 + s2; s0 := s1 * s2; s0 := s1 - s2; s0 := s1 / s2;
      s0 := -s1; s0 := s0*s1 + s0/s1;
   END P;
END M.
```

```
 3   E1810002        OR      R0 R1 R2            s0 := s1 + s2
 4   E0010002        AND     R0 R1 R2            s0 := s1 * s2
 5   E1C10002        BIC     R0 R1 R2            s0 := s1 - s2
 6   E0210002        XOR     R0 R1 R2            s0 := s1 / s2
 7   E1E00001        MVN     R0 R0 R1            s0 := -s1
 8   E0009001        AND     R9 R0 R1
 9   E0208001        XOR     R8 R0 R1
10   E1890008        OR      R0 R9 R8
```

The generation of set values is handled at compile-time whenever possible, i.e. when the elements are constants.

```
MODULE M;
   PROCEDURE* P;
      VAR s0: SET;
   BEGIN s0 := {}; s0 := {0}; s0 := {8, 10 .. 12, 15};
   END P;
END M.
```

```
11   E3A00000        MOV     R0 R0 0             s0 := {}
12   E3A00001        MOV     R0 R0 1             s0 := {0}
13   E3A09C9D        MOV     R9 R0 40192         s0 := {8, 10 .. 12, 15}
14   E1A00009        MOV     R0 R0 R9
```

The cases where the set elements are general expressions are more intricate. In fact they pose a genuine challenge to the compiler designer. The ARM processor fortunately features an attractive, regular instruction set. Its outstanding property is that within one instruction and one cycle both an arithmetic or logical operation, and a shift of the second operand may occur. This leads to the following surprisingly short code for the statements shown below. (Note that LSL means “logical shift left”. *m* is in register R11, *n* in R10).

```
MODULE M;  (*sets*)
   PROCEDURE* P(m, n: INTEGER);
      VAR s0, s1, s2: SET;
   BEGIN s0 := {m}; s0 := {0 .. n}; s0 := {m .. n}; s0 := {m+n .. m-n};
      IF n IN s0 THEN s0 := {0} END ;
      IF s1 <= s2 THEN s0 := {4} END
   END P;
END M.
```

```
 3   E3A08001        MOV     R8 R0 1
 4   E1B00B18        MOV     R0 R0 R8 LSL R11        s0 := {m}

 5   E3E09001        MVN     R9 R0 1
 6   E1E00A19        MVN     R0 R0 R9 LSL R10        s0 := {0 .. n}

 7   E3E09001        MVN     R9 R0 1
 8   E1E09A19        MVN     R9 R0 R9 LSL R10
 9   E3E08000        MVN     R8 R0 0
10   E0090B18        AND     R0 R9 R8 LSL R11        s0 := {m .. n}

11   E09B900A        ADD     R9 R11 R10              R9 := m+n
12   E05B800A        SUB     R8 R11 R10              R8 := m-n
13   E3E07001        MVN     R7 R0 1
14   E1E07817        MVN     R7 R0 R7 LSL R8
15   E3E06000        MVN     R6 R0 0
16   E0070916        AND     R0 R7 R6 LSL R9         s0 := {m+n .. m-n}
```

```
17  E3A09001    MOV   R9 R0 1
18  E1100A19    TST   R0 R0 R9 LSL R10      IF n IN s0 THEN
19  0A000000    BEQ      0
20  E3A00001    MOV   R0 R0 1

21  E1D1E002    BIC   LNK R1 R2             IF s1 <= s2 THEN
22  1A000000    BNE      0
23  E3A00E01    MOV   R0 R0 16
```