

CSCI 3155: Set types

October 1, 2003

1 Set types

Sets are an important mathematical concept; many mathematicians even consider it to be the most important concept in mathematics nowadays. Even outside of this field, sets are important tools in modelling certain properties. Assume, for example, that you want to model file access attributes for the file's owner. One approach would be to construct a record

```
TYPE Perm = RECORD
    read, write, execute : BOOLEAN;
END;
```

for this. Each element of `Perm` could now set `read`, `write` and `execute` permissions separately.

An alternative, which to many people would seem more intuitive, would be to construct a set type restricted to a subset of `{read, write, execute}`. For each value v of this type, we would be able to separately determine whether the (enumeration) elements `read`, `write` or `execute` are “contained” in v , i.e. whether `read` $\in v$ (analogously for `write` and `execute`).

At first, this would seem like little more than a minor notational convenience (and even this would be arguable). However, values of set types generally allow us to express (at least) the set operations for union and intersection, \cup and \cap :

$$\begin{aligned}\{A, B\} \cup \{B, C\} &= \{A, B, C\} \\ \{A, B\} \cap \{B, C\} &= \{B\}\end{aligned}$$

Using these, we could easily ensure e.g. that a file is not writable and not readable, by simply performing set intersection with `{execute}`; emulating this with the aforementioned record would require two statements and have a good chance of being less efficient (see Question #2 on the next worksheet), except for particularly well-optimising compilers.

The computation would look as follows:

$$\begin{aligned}\{\text{read, write, execute}\} \cap \{\text{execute}\} &= \{\text{execute}\} \\ \{\text{read, execute}\} \cap \{\text{execute}\} &= \{\text{execute}\} \\ \{\text{read, write}\} \cap \{\text{execute}\} &= \{\}\end{aligned}$$

1.1 Language support for set types

There are different ways in which sets can be implemented in programming languages:

1. Built-in primitives like `ARRAY` etc.
2. Supported through functions in the standard library
3. Not supported or only supported through external libraries

2 Set types in Modula-3

Modula-3 is one of the languages that treat set types as being special built-in types; more precisely, it treats `SET OF` as a built-in type constructor. The following passage is taken verbatim from the Modula-3 language definition:

2.2.6 Sets

A set is a collection of values taken from some ordinal type. A set type declaration has the form:

```
TYPE T = SET OF Base
```

where `Base` is an ordinal type. The values of `T` are all sets whose elements have type `Base`. For example, a variable whose type is `SET OF [0..1]` can assume the following values:

```
{ }      {0}      {1}      {0,1}
```

Implementations are expected to use the same representation for a `SET OF T` as for an `ARRAY T OF BITS 1 FOR BOOLEAN`. Hence, programmers should expect `SET OF [0..1023]` to be practical, but not `SET OF INTEGER`.

2.1 Constructing sets

Modula-3 allows literal set values to be expressed in a program; this is detailed below.

2.6.8 Set, array, and record constructors

A set constructor has the form:

$S\{e_1, \dots, e_n\}$

where S is a set type and the e 's are expressions or ranges of the form $lo..hi$. The constructor denotes a value of type S containing the listed values and the values in the listed ranges. The e 's, lo 's, and hi 's must be assignable to the element type of S .

2.2 Operations on sets

A number of operations on sets are pre-defined in Modula-3; an abbreviated account is given below.

2.6.11 Relations [excerpts]

`infix IN (e: Ordinal; s: Set): BOOLEAN`

Returns TRUE if e is an element of the set s . It is a static error if the type of e is not assignable to the element type of s . If the value of e is not a member of the element type, no error occurs, but `IN` returns FALSE

Note that `infix` here means that the operator being defined there can be used as an infix operator; i.e. "`1 IN S`" for a set S is a valid expression in Modula-3.

2.6.10 Arithmetic operations [excerpts]

`infix + (x,y: Set) : Set`

On sets, `+` denotes set union. That is, $e \text{ IN } (x + y)$ if and only if $(e \text{ IN } x) \text{ OR } (e \text{ IN } y)$. The types of x and y must be the same, and the result is the same type as both.

`infix - (x,y: Set) : Set`

On sets, `-` denotes set difference. That is, $e \text{ IN } (x - y)$ if and only if $(e \text{ IN } x) \text{ AND NOT } (e \text{ IN } y)$. The types of x and y must be the same, and the result is the same type as both.

`infix * (x,y: Set) : Set`

On sets, `*` denotes intersection. That is, $e \text{ IN } (x * y)$ if and

only if $(e \text{ IN } x) \text{ AND } (e \text{ IN } y)$. The types of x and y must be the same, and the result is the same type as both.

`infix / (x,y: Set) : Set`

On sets, `/` denotes symmetric difference. That is, $e \text{ IN } (x / y)$ if and only if $(e \text{ IN } x) \# (e \text{ IN } y)$. The types of x and y must be the same, and the result is the same type as both.

For this, note that the `#` infix operator in Modula-3 expresses inequality, i.e. $a \# b$ if and only if $a = b$ is NOT true.

3 Set types in other languages

3.1 Pascal

Pascal provides set types similarly to Modula-3, but restricts them to operate over enumeration types, i.e. it does not allow them over subranges.

3.2 C++

C++ has no built-in support for sets in the language (although certain low-level hacks— see Question #3— are often used to emulate them where they would be needed). The Standard Template Library provides a templated class `std::bitset`, which corresponds to set types over a subrange starting at 0.

3.3 Java

The closest thing to set types in Java is the class `java.util.HashSet` in the Java standard library. This class behaves very differently from the sets we have seen, however: Only objects can be added to it (i.e., no integers or floats), and there is no concrete set a hashset has to be a subset of (unless you count the “set of all objects”).

The practical effect of this is that it is not possible to invert such a set, and it is rarely clear what precisely one should expect from a given `HashSet`.

Java also implements `java.util.BitSet`, very similar to the `std::bitset` of C++, except that it can dynamically resize.

3.4 Haskell

Haskell’s built-in set type works very similarly to Java’s `HashSet`, except that it concretely specifies the type of all elements in the set. Haskell does not have subranges (except for four built-in types corresponding to integers of different byte-size representations), but it does allow for enumeration types, which can be used here. Haskell further specifies some operations on sets which other languages don’t have; all of these are due to its functional heritage.