Set Types
Handout C

6th June 2007

Sets of things are unordered collections of things, where each thing may occur at most once. While the usual definition of sets on mathematics is somewhat different– more extensive and more constructive– the above, simplified idea is what made its way into several programming languages.

To understand the use of sets assume, for example, that you want to model file access attributes. 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 is to construct a set type restricted to a subset of an enumeration type with the enumerated values `read`, `write`, `execute`. For each value v of this type, we can 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 seems like little more than a minor notational convenience (and even that 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 can easily ensure e.g. that a file is not writable and not executable, by simply performing set intersection with {`read`}; implementing the same with the aforementioned record requires two statements and would likely be less efficient.

Below are some examples of applying this intersection:

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

0.1 Language support for set types

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

(a) Built-in primitives like `ARRAY` etc.

(b) Supported through functions in the standard library

(c) Not supported or only supported through external libraries

1 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
```

Here, `ARRAY T OF BITS 1 FOR BOOLEAN` implies a "packed array", wherein each boolean variable in the array is assigned precisely one bit. The advantage of such a packed representation is that it requires very little memory: An `ARRAY [0..1023] OF BITS 1 FOR BOOLEAN` takes up a mere 128 bytes.

1.1 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, ..., 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.
```

1.2 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 IN (x + y) if and only
if (e IN x) OR (e 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 IN (x - y) if and
only if (e IN x) AND NOT (e 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 IN (x * y) if and
only if (e IN x) AND (e 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 IN (x / y) if
and only if (e IN x) # (e 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.

Thanks to the "packed array" representation of sets, the above operations can be implemented very efficiently, operating over several set elements at the same time.

2 Set types in other languages

2.1 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.

2.2 SetL

The programming language *SetL* is unique in that it was defined solely in terms of sets, albeit with a more powerful notion of sets that is closer to the usual mathematical definition.

2.3 Other languages

Many modern languages provide sets as part of their standard library. For example, Java provides java.util.Set as a general interface to set implementations, with java.util.HashSet as one possible implementation of sets. Python, since version 2.3, provides a sets module, with similar operations. Haskell provides sets by means of special operations over lists, which can otherwise be used in a manner similar to SetL.