

# Chapter 3: Sort Hierarchies and Membership Axioms

In the previous chapter, we saw the the line

```
subsort Bit < Bits .
```

which allowed us to treat ``objects" of sort `Bit` as a subsort of `Bits`, allowing us to use operations defined over `Bits` on objects of sort `Bit`. In this case, we essentially only get syntactic convenience: it simplifies and shortens our equational definitions of the behaviour of our operators. Since Maude allows operator names to be overloaded, we could have done without the subsort operator. However, Maude also provides *membership axioms* - expressions stating what it means for an object to be of a particular sort.

## 3.1. Example

Here is an example from the Maude manual:

```
fmod PATH is

  protecting MACHINE-INT .

  sorts Edge Path Path? Node .

  subsorts Edge < Path < Path? .

  ops n1 n2 n3 n4 n5 : -> Node .
  ops a b c d e f : -> Edge .

  op _;_ : Path? Path? -> Path? [assoc] .

  ops source target : Path -> Node .

  op length : Path -> MachineInt .

  var E : Edge .
  var P : Path .

  cmb (E ; P) : Path if target(E) == source(P) .

  ceq source(E ; P) = source(E) if E ; P : Path .
  ceq target(P ; E) = target(E) if P ; E : Path .

  eq length(E) = 1 .
  ceq length(E ; P) = 1 + length(P)
    if E ; P : Path .

  eq source(a) = n1 .
  eq target(a) = n2 .
  eq source(b) = n1 .
  eq target(b) = n3 .
  eq source(c) = n3 .
  eq target(c) = n4 .
  eq source(d) = n4 .
  eq target(d) = n2 .
  eq source(e) = n2 .
  eq target(e) = n5 .
  eq source(f) = n2 .
  eq target(f) = n1 .

endfm
```

The module *PATH* describes a graph, with nodes `n1` to `n5` and edges `a` to `f`. We are interested in *paths* through the graphs - sequences of edges in which the tail of one edge and the head of the next are the same node. To do this, we introduce three sorts:

- Edge which are paths of length 1;

- Path which are "proper" paths; and
- Path? which "might be" paths - or which might be disjoint sequences of edges.

The concatenation operator `_;` joins sequences of sort `Path?` - but notice the line

```
cmb(E ; P) : Path if target(E) == source(P) .
```

This is a *conditional membership axiom* which "narrows" `(E ; P)` from `Path?` to `Path` provided the end of `E` is the same node as the start of `P`. There is a corresponding non-conditional axiom `mb`.

This is a very convenient mechanism for turning what could be complex and clumsy code into something simple, elegant and readable. Formally, we are no longer dealing with many-sorted algebra, but with *order sorted* algebra, which has a number of mathematical implications. However, these will not affect us. In fact, we are only going to make relatively small use of sub sorting.

### 3.2. Membership Axioms in Microprocessor Models

We can use membership axioms to simplify our microprocessor representations, and make them easier to read. Most of you will recall that in specifying microprocessors in System Specification (when using Maude), we needed to use an "unsatisfying" technique to control word lengths. We used the `BITS` module (an extension of that shown in the *previous chapter*) to deal with operations on bit strings: the problem with this is that it deals with bit strings of arbitrary length, and we commonly wish, with microprocessors, to deal with *fixed* lengths. This leads to unsatisfactory code, like the following memory module.

```
fmod MEM is

  protecting BITS .

  sorts Mem .

  *** Memory Read
  op _[_] : Mem Bits -> Bits .

  *** Memory Write
  op _[_/_] : Mem Bits Bits -> Mem .

  var M : Mem .

  vars A B : Bits .
  var W : Bits .

  eq M[W / A][A] = bits(W,7,0) .
  ceq M[W / A][B] = bits(M[bits(B,31,0)],7,0)
    if %(A) /= %(B) .

endfm
```

Recall that the `_[_]` operator represents memory read, and `_[_/_]` represents memory write. We wish to represent a memory that stores 8-bit words (bytes), and which has 32-bit addresses. However, to enforce this, we must use the `bits` operator, which extracts substrings, to truncate words read to 8 bits (`bits(W,7,0)`) and addresses to 32 bits (`bits(A,31,0)`). In addition, because we are dealing with potentially arbitrary length strings, and because Maude will not regard, say, `00110` and `110` as equal (they are different *strings*, even though they represent the same *number*), we must use the `%` operator to "normalise" by deleting leading zeros. (You may observe that there is still a potential problem, with two's complement numbers, because of leading ones.)

#### 3.2.1. A Better Solution with Subsorts

Here is a much neater alternative, using subsorts:

```
fmod MACHINE-WORD is

  protecting BITS .

  sorts Byte Word .

  subsort Byte < Bits .
  subsort Word < Bits .
```

```

vars A0 A1 A2 A3 A4 A5 A6 A7 : Bit .
vars A8 A9 A10 A11 A12 A13 A14 A15 : Bit .
vars A16 A17 A18 A19 A20 A21 A22 A23 : Bit .
vars A24 A25 A26 A27 A28 A29 A30 A31 : Bit .

*** Membership axiom for bytes
mb (A0 A1 A2 A3 A4 A5 A6 A7) : Byte .

*** Membership axiom for words
mb (A0 A1 A2 A3 A4 A5 A6 A7
    A8 A9 A10 A11 A12 A13 A14 A15
    A16 A17 A18 A19 A20 A21 A22 A23
    A24 A25 A26 A27 A28 A29 A30 A31) Word .

*** Other stuff (probably) deleted
endfm

fmod MEM is

protecting BITS .

sorts Mem .

op _[_] : Mem Word -> Byte .
op _[_/_] : Mem Byte Word -> Mem .

var M : Mem .

vars A B : Word .
var W : Byte .

eq M[W / A][A] = W .
ceq M[W / A][B] = M[B] if A /= B .

endfm

```

We have introduced two new sorts `Byte` and `Word`, which are subsorts of `Bits`. A bitstring is a `Byte` if its length is 8 bits, and a `Word` if its length is 32 bits. This representation is much neater - it allows us to ditch the `%` operator, and the clumsy use of `bits`: it also allows us to precisely specify the sorts that the memory read and write operators use - rather than just `Bits`.

### 3.2.2. A Little Experiment...

You may well think that the membership axioms above are clumsy - especially for words. A fairly obvious alternative is

```

var Y : Bits .

cmb Y : Byte if | Y | == 8 .
cmb Y : Word if | Y | == 32 .

```

where `|_|` : `Bits` -> `MachineInt` is the length operator. The trouble with this is the way Maude deals with [conditional] membership axioms. In the *next chapter* is an example, which uses the first form of non-conditional axiom, but which contains the conditional form commented out. Try using the conditional form first. While you are waiting for it to finish, read the paragraph below (Hint: I'd type `ctrl-C` now: I tried this as an experiment on a 667MHz PIII with 1GByte of memory and gave up after 10 minutes CPU time.)

Maude will continually try to apply any membership axioms to narrow a sort to one of its subsorts: in this case, `Bits` to one of `Byte` or `Word`. It does in the usual way: by pattern matching. In the non-conditional case, only 8-bit and 32-bit words match, and all others get left alone. In the conditional case, *every* occurrence of `Bits` is a potential candidate, which needs to have its length checked with `|_|` (which is recursively defined, and takes more rewrites for longer bit strings). This is an additional overhead, made worse because there are likely to be many more occurrences of `Bits` than you may first think. Consider the string `0 0 0 0 0 0 0 0` representing a `Byte` constant zero. This is not one occurrence, but seven because the string is built by successive concatenation of bits using the `__` operator (see *the basic Maude chapter*). So first we construct `0 0`, which is checked against all the membership axioms, and then `0 0 0` and so on. If you really want to do some comparisons of the number of rewrites, I suggest you modify the example in the *next chapter* to reduce

something simple like  $pc(Mp, Md, Pc, Rg)$  (i.e. just extract the program counter value from the state). This takes 2 rewrites with the non-conditional axioms, and 720 000 (!) with the conditional ones.