

Task: 1.3  
Version: 1.3  
Date: 1999-04-12



## A Small OpenMath Type System

James Davenport

Bath

**1.3.2c (Public)**



## **Abstract**

This paper is based on various discussion with the OpenMath Consortium, and notably recently with the NAG team. All errors are mine. Revised in the light of discussion at Oxford 4.2.1999.



## 1 Why a “Small Type System”?

Nothing in this document should be inferred as meaning that a Small Type System is “better” than ECC theoretically: indeed clearly it is not. One should understand “Small” in the sense of “light-weight”. There are two main uses of a potential Small Type System for OpenMath signatures.

- Tools which read “new” CDs automatically, e.g. a search engine which suddenly encountered as a symbol from a CD that it did not recognise. In the context of a “Small” type system, there is little more that can be offered to such systems than arity checking.
- Human beings reading the whole of a CD’s fields, to determine how they ought to implement the CD, either as OpenMath-reading software, or, more interestingly, as OpenMath-writing software. These humans need to interpret the symbols in a way that (short of ECC or re-building an Axiom-like category system) cannot be totally formalised. However they “ought to be” imbued with that nebulous quality of “common sense”.

To meet these goals, we define a system that encodes arity (and a little more) in a totally formal way, but leaves clues in names etc. that should help the human reader.

## 2 Principles

1. A signature is an OpenMath object, and should be encoded as such. Alas, I do not guarantee that I have got the encodings right, but this is the principle of the document.
2. Every OMS must appear in a CD. For the sake of specificity, I have chosen a separate CD, `sts`, for those symbols specific to the Small Type System. This could clearly be changed.
3. It is a truism of type theory that a constant can be replaced by a nullary function. Early precursors of Axiom did this, but this had to be changed, since the pragmatics of, say, `zero` (thought of as `zero()`) and `random()` are very different. Even equality cannot be the same. Hence we distinguish between the two. On the same lines, forcing all functions to have one argument by currying is tempting, but a pragmatic disaster. This is not to say that a more adventurous type system could not do it, but this defeats both goals in the introduction at a “Small” level.
4. Clearly  $n$ -ary functions must be allowed. However, there are two types of these: those that simply take  $n$ -arguments, and those that are essentially binary associative functions normally written  $n$ -ary. As examples of these

$$\text{list}(1,\text{list}(2,3)) \neq \text{list}(1,2,3)$$

even though `list` is an  $n$ -ary function, but

$$\text{plus}(1, \text{plus}(2, 3)) = \text{plus}(1, 2, 3)$$

since `plus` is also associative. It would be convenient (especially for search engines) to distinguish the two. Further justification for this view is given in the Appendix.

5. Some names (to be decided, and probably evolving with CD's) should refer to specific OpenMath known types, and some cannot be prescribed. For example, it may<sup>1</sup> be convenient to know that the second argument of `elt` (element of a list) must be a positive integer, and hence we can say this by writing

```
<OMS name="PositiveInteger" cd="sts"/>
```

whereas in other cases we may wish to convey to the human reader that the types have some generic property that the Small Type System does not formally encode, e.g. a `gcd` operation might wish to name its arguments and results without a formal definition, and therefore use

```
<OMV name="GCDDomain"/>
```

In particular, we use

```
<OMS name="Object" cd="sts"/>
```

to denote any OpenMath object (as far as the Small Type System is concerned).

6. Function arguments (or results) are first-class objects. This is certainly true of OpenMath, so should be true of its type system.
7. Parameterised types are probably too complex for the Small Type System. Use ECC instead!

### 3 Encoding

This attempts to write a quasi-BNF grammar for the OpenMath encoding of a signature `SIG`. I use `OMS!` as shorthand for an Openmath symbol encoding, (possibly attributed) and similarly `OMV!`. Quotation marks ‘ ’ denote literal OpenMath, except that I quote one possible order for `name=` and `cd=`, but intend both. Similarly, I do not go into the blank space and comment rules.

1. A rule to represent principles 5 and 6.

```
OMSV = OMS! | OMV! | APPL
```

---

<sup>1</sup>Or it may not, if we are writing about list algebras. This distinction is actually hard.

2. Rules to represent principle 4.

```
NARYS = '<OMS name="nary" cd="sts"/>' |
        '<OMS name="nassoc" cd="sts"/>'
```

```
NARY = '<OMA>'
        NARYS
        OMSV
        '</OMA>'
OMSVN = OMSV | NARY
```

3. Rule for application<sup>2</sup>.

```
APPL = '<OMA>'
        '<OMS name="mapsto" cd="sts"/>'
        OMSVN*
        OMSV
        '</OMA>'
```

The last OMSV is to be thought of as the target, the rest as source domains. There also needs to be a side-rule, which I cannot write in BNF, that at most one of the OMSVN\* can be an NARY, the rest must be OMSV.

4. Finally,

```
SIG = OMS! | OMV! | APPL
```

(in fact OMSV would do, but this confuses the distinction between constants (the first two) and functions).

## 4 Examples

Here I give, as best I can, some examples of this encoding.

```
zero <OMV name="AbelianMonoid">
random <OMA>
    <OMS name="mapsto" cd="sts"/>
    <OMS name="Object" cd="sts"/>
</OMA>
```

---

<sup>2</sup>As usual in OpenMath, this is abstract application, and implies no computation.

```

minus <OMA>
  <OMS name="mapsto" cd="sts"/>
  <OMV name="AbelianGroup"/>
  <OMV name="AbelianGroup"/>
  <OMV name="AbelianGroup"/>
</OMA>

```

There is an important point of interpretation here. OMVs represent the same object in a local context<sup>3</sup>, so this means that **minus** takes two objects from the *same* AbelianGroup, and returns an element of the *same* AbelianGroup.

```

list <OMA>
  <OMS name="mapsto" cd="sts"/>
  <OMA>
    <OMS name="nary" cd="sts"/>
    <OMS name="Object" cd="sts"/>
  </OMA>
  <OMS name="Object" cd="sts"/>
</OMA>

```

This allows for heterogenous lists, but I doubt that the Small Type System is up to enforcing anything else.

```

plus <OMA>
  <OMS name="mapsto" cd="sts"/>
  <OMA>
    <OMS name="nassoc" cd="sts"/>
    <OMV name="AbelianSemiGroup"/>
  </OMA>
  <OMV name="AbelianSemiGroup"/>
</OMA>

```

```

determinant <OMA>
  <OMS name="mapsto" cd="sts"/>
  <OMV name="SquareMatrix"/>
  <OMV name="CommutativeRing"/>
</OMA>

```

Note that principle 7 implies that we do not attempt to parameterise the **SquareMatrix**, so that the human reader has to infer that we mean one over the following **CommutativeRing**. However, we can convey hints by writing **SquareMatrix** rather than **Matrix**, and not just writing **Ring**.

---

<sup>3</sup>Current meaning an OMOBJ, but I am not pre-empting any change from the DOM work here.

## 5 The Treatment of OMBIND and OMATTRIB

We now have to integrate OMBIND into this construct. The clue is provided by the fact that a signature with `mapsto` means, precisely, that that symbol can appear as the head of a OMA. By analogy, we need a symbol, say `binder`, to say that a symbol can appear as the head of a OMBIND.

In addition to being used as the head of OMA or OMBIND, some symbols in a CD may be intended to be used as a ‘key symbol’ in the symbol–value pairs of an OpenMath Attribution. Such a symbol is specified using the sts symbol `attribution`.

We therefore augment the grammar given above by the following two clauses:

```

BIND = ‘<OMA>’
      ‘<OMS name="binder" cd="sts"/>’
      OMV!
      OMSV
      ‘</OMA>’

ATTRIB = ‘<OMA>’
        ‘<OMS name="attribution" cd="sts"/>’
        ‘</OMA>’

```

and change the definition of SIG to be

```
SIG = OMS! | OMV! | APPL | BIND | ATTRIB
```

(equivalent to OMSV | BIND | ATTRIB).

## A Justification for nassoc

It would be possible not to describe `nassoc` functions differently from `nary` ones. This would be possible if OpenMath were not a semantic system. In particular, the Formal Mathematical Properties (FMPs) are hard to write. The commutativity of a binary addition can be addressed, as in `arith.ocd`, as follows.

```

<FMP>
<OMOBJ>
  <OMBIND>
    <OMS cd="quant" name="forall"/>
    <OMBVAR>
      <OMV name="a"/>
      <OMV name="b"/>

```

```

</OMBVAR>
<OMA>
  <OMS cd="relation" name="eq"/>
  <OMA>
    <OMS cd="arith" name="plus"/>
    <OMV name="a"/>
    <OMV name="b"/>
  </OMA>
  <OMA>
    <OMS cd="arith" name="plus"/>
    <OMV name="b"/>
    <OMV name="a"/>
  </OMA>
</OMA>
</OMBIND>
</OMOBJ>
</FMP>

```

However, this does not even state that a ternary plus could be commutative in the first two arguments, and certainly does not state that, say,  $a + b + c = a + c + b$ .

A similar problem is posed by associativity: one can write a property equivalent to  $a + (b + c) = (a + b) + c$ , but again this does not cover  $n$ -ary functions. It would be possible to cope with ternary usages, e.g. by writing an FMP equivalent to  $(a + b) + c = a + b + c = a + (b + c)$ , but even this would not cope with quaternary usages, as in  $a + b + c + d$ .

There is a fundamental dichotomy between allowing  $n$ -ary operations for notational and practical convenience, on the one hand, and a formal semantics system, which generally only allows operations of fixed arity. Treating such operations as “special” is the easiest way of solving this.