RM 23/5/84

A simplified syntax for instances and module definitions, with a proposal for treating sharing specifications

(for discussion at ML meeting, 6-8 June)

1. Motivation

I believe that clarity, and some generalisation, can be got by two means:

(1) We take as basis a simple syntax for the class inst of instance† expressions, and then take

module mod_id (...) {: sig} = inst

as the syntax for modules. Dave's modules correspond to the particular case where the right hand side above takes the form "*inst* ... *end*", but the new form allows right-hand sides which are arbitrary instance expressions, involving perhaps other module identifiers. (See Section 3, concerning Dave's limitation).

(2) In this simple syntax, we demand that every module takes exactly one instance parameter. This not only yields a rather uncluttered basic syntax, but also clarifies the rôle of *sharing* specifications in modules — in fact, it reduces them completely to sharing specifications occurring in signatures only. A simple derived form then allows modules with n instance parameters for any $n \geq 0$.

Two remarks: First, I very much prefer pedagogically the idea of starting with instances — or better, structures! — as the basic concept, and regarding *module definitions* as structure schemata. One would very naturally teach it in this way. Second, the essence of my proposal for sharing specifications does not *demand* that the basic syntax of module definitions should contain exactly one instance parameter; the derived form could be taken as basic, but the essential idea is to include *sharing* specifications in the formal parameter part.

† I would much prefer "*structure*" to "*instance*", and therefore "*struct*" in place of "*inst*", but I want to preserve this note using Dave's notation wherever possible

2. Instance expressions

First, before defining the class inst, we suppose that declarations are extended by the new form

```
dec  ::=  instance  ib
ib   ::=  inst-id {: sig} = inst
          ib1 and ... and ibn
```

This is just as Dave had it (except that I have added {: sig}); but one generalisation will occur because this general form will be allowed inside instance expressions and hence inside module bodies. I can see no compelling reason to restrict such occurrences to

```
mod_inst_dec  ::=  instance  inst_id = inst_path
```

as in Dave's proposal (but see Section 3 below for possible reason).

The class dec will also contain new forms like "open inst_id+", but let's ignore this for the present.

The basic syntax for instance expressions will now be

```
inst  ::=  inst dec end
           inst_path
           mod_id (inst)
```

Note that dec can be empty (we shall need the empty instance), and that we have given mod_id exactly one instance parameter. Note also that we have simplified the first alternative from Dave's

```
inst  mod_spec* dec  end
```

because (a) sharing specifications will no longer be needed in this form, and (b) the other kind of mod_spec, mod_inst_dec, has been subsumed by dec.

3. Modules

The new form for modules is

$$\boxed{\text{mod} ::= \underline{\text{module}}\ \text{mod\_id}\ (\text{inst\_id} : \text{sig})\ \{: \text{sig}\} = \text{inst}}$$

As noted in §1 above, the body of a module is now an arbitrary instance expression, in which any free occurrences of inst_id are bound by the module definition. I have resisted introducing syntax for anonymous modules, such as "$\underline{\text{mod}}$ (inst_id : sig) inst", since it would raise too many questions if we were to propose any use of such module expressions occurring without being applied to, an explicit instance argument (as for example "$\underline{\text{fun}}$ (vs) e" can occur without an explicit value argument).

The restriction (in this basic form) to a single instance parameter is related to sharing specifications, to which we now turn. After that, we give derived forms allowing many (or no) parameters.

Remark Dave appears to have restricted modules so that a module will contain no global references to other modules or to instances. Thus, all that is needed to compile one of his modules is the signatures to which it refers. I see that this will have some pragmatic advantage, but I also think it has caused him to adopt some rather piecemeal restrictions.

I would favour adopting syntax which allows such global references (as this proposal does); then in a first implementation we can simply impose the restriction "no global identifiers in modules except signature identifiers", if there is strong enough reason to do so.

4. Sharing specifications

Consider Dave's GEOMETRY example (his page 11):

```
module GeometryMod (R : RECT, C : CIRCLE) : GEOMETRY
    instance R = R
    instance C = C
    sharing R.P = C.P
    ....
end
```

Note first that the sharing specification is independent of whether or not R and C are to be inherited by the module (as they are here, via the two instance declarations). One can easily imagine a case in which sharing of R.P and C.P is to be specified even though R and C are merely used locally in GeometryMod.

I argue that such sharing specifications within modules will only be required to specify sharing among the components of the formal parameter instances, and that as such the sharing specifications properly belong in the formal parameter part of the module definition. In fact, in the basic syntax which I am proposing, the above module definition would become:

```
module GeometryMod (RC : {sig instance R : RECT and C : CIRCLE
                              sharing R.P = C.P end}) : GEOMETRY
= inst
      instance R = RC.R and C = RC.C
      .....
  end
```

but a derived form (given later) will bring it back closer to Dave's form. The essence of the basic idea is that, by inventing a signature for a single formal parameter instance, we gain access to the use of sharing specifications within signatures.

5. Derived forms for multiple instance parameters

The main point of our derived form for defining modules with many instance parameters is that sharing specs (exactly in Dave's syntax) are allowed in the formal parameter part. For the GEOMETRY example we would have

```
module GeometryMod (R:RECT, C:CIRCLE sharing R.P=C.P) : GEOMETRY
  = inst  instance R = R  and  C = C
              . . .
      end
```

and uses of the module — defined by this derived form — will be of the form

GeometryMod (Rect, Circle)

or GeometryMod (RectMod (Point), CircleMod (Point))

exactly as Dave would allow.

In general then, we propose the derived definition form (for n ≥ 0):

```
module M (I1 : S1, ..., In : Sn {share_spec}) : Sig = inst
        ⟼
   module M (I : sig instance I1:S1 and ... and In:Sn {share_spec} end) : sig
                                   = inst {I.I1/I1, ..., I.In/In}
```

where I is a fresh inst_id, and the simultaneous substitution {...} needs a little care to take account of clashes with new uses of I1, .., In in inst. Correspondingly, all uses of this module M (applied to n instance expressions) are regarded as derived forms, as follows:

```
M (inst1, ..., instn) ⟼ M (inst instance I1 = inst1 and ... and In = instn end)
```