# Experiences with the ML Module System

or

# Why I Hate ML

Perdita Stevens

ML Club 5/2/98

# YMMV

My experience with ML is from three years of maintaining the

Edinburgh Concurrency Workbench.

Neither it nor I may be representative:

- I was prepared to be converted and have not been: but perhaps I'm incorrigibly corrupt;
- the CWB is not generally considered (even by all its writers) an example of good ML style.

Today I speak as a software engineer, not language designer, compiler writer, theoretician, clever programmer, or even teacher of programming languages.

**The opinions to come are not necessarily yours.**

## Focus of this talk

When people try to sell you ML, they say (not necessarily in this order):

1. It's functional, and of course you want that don't you;
2. It's type safe, so your programs won't have bugs;
3. It has a formal semantics, so ... err ... it's good.

Arguing over 1 is probably fruitless: I'm object oriented and don't find functional programming natural, but that's not what I want to talk about today. I claim that 3 is irrelevant, and that

**ML's drawbacks are more than sufficient to demolish any benefits that might accrue from 2.**

# Encapsulation

Structure of the program is widely visible because structure of the namespace is the same as structure of the program.

E.g.

```
sig
  structure A : AS
  val foo : A.t -> unit
end
```

Things (especially “remote” things) that use this signature probably shouldn’t be concerned about its internal structure: but they must be.

## Changes to the structure propagate through a system

## Alternatives?

Expose types, not structures? `include AS` instead, then `open A` in the functor body? In either case you need multiple sharing constraints if the component is in fact used in several places.

Despite my efforts at clearing up, the CWB is full of code like

```
functor TestOps ( structure Eq   : EQUIV;
                  structure TG   : TGRAPH;
                  structure GenP : GENPRE;
                  structure PG : POLYGRAPH;
                    sharing PG = Eq.PG = TG.PG = GenP.DivC.Elem.PG
                        and type GenP.DivC.Elem.accset = TG.AS.accset
                )
```

**We need some kind of package system, to control the namespace at the right level.**

## Sharing types

is a total pain, mostly for philosophical reasons (it's objects that are the atoms of thought, not types or functions).

More concretely, it has bizarre features e.g. illegality of

```
sharing type t = int list
```

There are hackish fixes, but that's not good enough.

Effect is to force me into putting more weight onto the module structure, which can't stand it.

**Of trivial irritations is hatred born.**

## What to functorise on?

Functorising is the only standard way to get encapsulation.

CWB used to try to parameterise on any structure that was mentioned: even things that were essentially library data structures, like SL. But we build structures straight after loading their source – so if you missed out a parameter from a later functor you probably never knew it...

Total maintenance pain: must alter both functor body and its calls any time you introduce a new data structure.

Now I only parameterise on a thing if I might conceivably want to instantiate it 2 ways (so Agent depends on Act, but not on SL).

But ML97 will stop me doing this??

**There's no Right Thing, because functorisation is not a good primary structuring mechanism.**

# Non-integrated module system...

...which leads to effects which range from the catastrophic to the risible.

Catastrophic: illegality of

```
structure Act = if sccs then SCCSAct(Part) else CCSAct()
```

Risible: illegality of

```
structure _ = A()
```

**A language and its encapsulation mechanisms should be designed together.**

# Usable interfaces

Modules often need separate protected and public interfaces. E.g. the writer of a structure implementing auxiliary commands like `diverges` needs more generous access to AGENT features than the outside world does.

e.g. maybe AGENT should expose:

```
structure Act : ACT
type agent
```

whereas AGENT-PROTECTED may expose:

```
structure Act : ACT-PROTECTED
datatype agent = ...
```

Don't want duplication between AGENT and AGENT-PROTECTED. (It's bad enough that a datatype defn has to be duplicated in sig and struct!)

Eventually, with help from local experts I found out how to do this. After a fashion.

**But it's basic, and should have been easy.**

## Exceptions

Are a jolly good idea, and widely used by ML programmers – but insufficiently uncontrolled. Cf Java, which took them on in tamed form.

A CWB component writer can write code with an internal exception that is not in the interface, and in one case is not handled. It will rise to the top level. All I can do is catch it and print an error “This is a bug”. Big help.

**If your program can die because of an unhandled exception, who cares that it can’t die because of a run-time type error?**

# OK for type theorists...

... but what is a mortal supposed to do with the error message

```
Error: nongeneric weak type variable
  e : '0Y E.env ref
```

ML's decisions about references, eqtypes etc. can be seen as pragmatic iff you understand the type theory underlying them (or trust someone who does).

Otherwise they're just irritating – e.g. when a change to the implementation of a data structure forces changes to its interface (replacing `'a` by `'_a` throughout).

**By all means let sophisticated type theory provide correctness guarantees, but keep it away from programmers.**

# Complete type inference considered harmful

Claim: humans have to understand types to understand code.

```
| os n = SL.bigmerge obs_le true
       (map (fn (seq,P) =>
             flatten(map (fn (a,P) =>
                  map (fn Q => (seq@[a],Q))
                  (tauclosure P))
                     (SL.remove fst A.le (A.tau,transitions P))))
        (os (n-1)))
```

So why isn't this information in the program? Because someone thought it was clever to leave it out – and when ML polymorphism is all you have, you can't put in enough information to be useful without also making the code too inflexible.

**Lack of type information $\Rightarrow$ unreadable code $\Rightarrow$ bugs.**

## Safety in numbers 1: tools

Basic facilities are absent: clean, efficient, powerful collections, for example. (Even basic array functions!) Debuggers. Development environments. (Free ones!)

So programmers use lowlevel data structures like lists. Especially when efficiency is paramount: previous CWB maintainers have avoided putting in encapsulation because “it’s too slow”.

The CWB had many bugs of the form, `f` assumes that `g`’s output is sorted and it isn’t, because `g` called `h` called `k` which did something to mess up the sorting. Hard to test for, hard to fix.

The error above was introduced early in 1992.

**This kind of error would not have occurred if the CWB had been written in Smalltalk.**

## Safety in numbers 2: community

Why is it easy to write powerful programs in Perl, despite that language's many failings?

Because if you want a sensible component, it exists, is free, and you can find it. You can also find someone who knows how to solve your problem.

The object-oriented community has a growing understanding of design patterns – well-understood good solutions to common problems in context. The context includes being in an OO system.

Doubtless if ML had a similar community, there would be patterns that assumed you were working in the ML module system.

**But it hasn't.**

# And another thing

Even the core language isn't perfect! Complaints selected to fit on slide:

- Can't have recursive type and datatype: e.g. must write

  ```
  datatype privateAgent = ....
  and agent = DUMMY of (privateAgent * int) ref
  ```

  (And I can't avoid exposing the hack in the signature, either.)

- Syntax for giving type information is clumsy

- More exception weirdness:

  ```
  fun g _ =
    let exception Foo
        val _ = raise Foo
    in 1
    end handle Foo => 2
  ```

# In conclusion

Only with great reluctance would I decide to develop another large, long-lived system, even a new CWB, in ML.

Some of the problems can be fixed by tinkering with the language. Others, I believe, result from the fundamental problem of designing the language in two layers.

My guess is that nothing recognisable as ML will ever be a serious, popular, industrial programming language.

**We should be concentrating on isolating and understanding the good things about it and getting them into new languages.**