# Close Encounters of the Higher Kind

## Emulating Constructor Classes in Standard ML

Yutaka Nagashima
Data61, CSIRO / NICTA
yutaka.nagashima@nicta.com.au

Liam O'Connor
UNSW Australia and Data61, CSIRO / NICTA
liamoc@cse.unsw.edu.au

*We implement a library for encoding* constructor classes *in Standard ML, including elaboration from minimal definitions, and automatic instantiation of superclasses.*

## 1. Introduction

In our recent work [5] on automating Isabelle proofs, we discovered that several proof search problems can be elegantly expressed as a monadic program. Unfortunately, Standard ML does not natively support the kinds of polymorphism required to easily express a `Monad` abstraction, nor similar abstractions such as `Applicative` and `Functor`[1]. In this paper, we present a technique for encoding constructor classes such as `Monad`, which relies only on the Standard ML module system.

Several others have attempted to enable constructor classes in Standard ML by changing the language. While it is tempting to customise the language by adding new features, new features tend to cause duplication [2] and inconsistency [3]. Furthermore, avoiding language extensions makes our approach transferable to all other ML dialects with a module system.

Our contributions are twofold: we develop a usable library for monads, monad transformers, applicatives, and more in Standard ML, and demonstrate an elegant technique using ML functors to elaborate minimal definitions of each abstraction to avoid code duplication. For example, given a minimal definition of the `list` monad, *e.g.* `return` and `bind`, our library derives other basic functions, such as `>=>`, `join`, and `liftM` automatically. Moreover, using the hierarchical relationship among constructor classes, our library automatically instantiates `list` as a member of the parent classes, *e.g.* `applicative` and `functor`. Thus, for each `monad`, users can derive more than twenty functions from two manually written functions, *i.e.* `return` and `bind`.

## 2. Constructor Classes in Standard ML

Figure 1 shows the structure of the class hierarchy as it is implemented in our library. Each node represents a ML `signature`. Straight arrows stand for subtyping relations, whereas dashed arrows with labels stand for ML `functor`s and their names. The ML `functor`s expressed as vertical dashed arrows, *e.g.* `mk_Monad`, produce full definitions of constructor classes from the corresponding minimal definitions; those expressed as horizontal dashed arrows, *e.g.* `Mona_Min_To_App_Min`, generalise minimal definitions for a class to its superclass.

For example, the following code snippets show the specifications of `MONAD_MIN` and `MONAD`.

```
signature MONAD_MIN =
sig
```

[1] Not to be confused with an ML `functor`

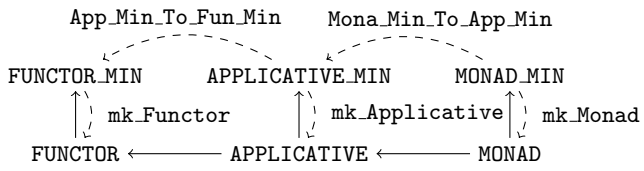


**Figure 1.** Automatic instantiation and function derivation.

```
 type 'a monad;
 val return:'a -> 'a monad;
 val bind:'a monad -> ('a -> 'b monad) -> 'b monad;
end;
signature MONAD =
sig
 include APPLICATIVE MONAD_MIN;
 sharing type monad = applicative;
 val liftM : ('a -> 'b) -> ('a monad -> 'b monad);
 val join: ...; val forever: ...; val ...
end;
```

Since every `monad` is `applicative`, we express this subtyping relation using the ML keyword `include`. In Haskell, this relation is expressed as `class Applicative m => Monad m`.

In order to create a concrete instance of a constructor class, the user merely supplies its minimal definition. For example, one can instantiate the type constructor `list` as a member of `MONAD` by defining the following module.

```
structure ListMonadMin:MONAD_MIN =
struct
 type 'a monad = 'a list;
 fun return x = [x];
 fun bind seq func = List.concat (map func seq);
end;
```

Then, passing `ListMonadMin` to the ML `functor` `mk_Monad` produces a full-fledged instance of `MONAD`:

```
structure ListMonad:MONAD = mk_Monad(ListMonadMin)
```

A minimal instance of `APPLICATIVE` can also be produced from our structure by using the appropriate functors:

```
structure ListAppMin:APPLICATIVE_MIN
   = Mona_Min_To_App_Min(ListMonadMin)
```

Note that these ML `functor`s go in the same direction as the subtyping relation, unlike the elaboration functors such as `mk_Monad`. The following shows the definition of the `Mona_Min_To_App_Min`

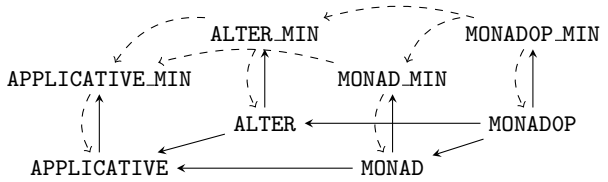


**Figure 2.** Diamond case.

functor.

```
functor Mona_Min_To_App_Min (Min:MONAD_MIN) =
struct
 open Min;
 type 'a applicative = 'a monad;
 val pure = return;
 fun <*> (fs, xs) = bind fs (fn fs' =>
                    bind xs (fn xs' =>
                    return (fs' xs')));
end : APPLICATIVE_MIN ;
```

The ML `functor Mona_Min_To_App_Min` produces instances of `APPLICATIVE_MIN` in terms of `MONAD_MIN` functions. This is in contrast with the constructor classes in Haskell where `return` is defined as `pure`. It is this inversion that enables our library to derive superclass instances for a given type constructor.

The elaboration functor `mk_Monad` is defined as follows.

```
functor mk_Monad (Min : MONAD_MIN): MONAD =
struct
 type 'a monad = 'a Min.monad;
 structure App_Min = Mona_Min_to_App_Min (Min);
 structure App = mk_Applicative (App_Min);
 open App Min;
 fun liftM f m = bind m (fn m' => return (f m'));
 fun join n = ...; fun forever a = ...; fun ...
end;
```

Apart from producing the various `MONAD` functions, `mk_Monad` instantiates `list` as a member of `APPLICATIVE` by elaborating the result of the functor `Mona_Min_To_App_Min` with `mk_Applicative`, which in turn instantiates the `FUNCTOR` class similarly.

We formalise monad transformers as ML `functor`s, too. For instance, the state monad transformer is a `functor` that takes two modules, the minimal definition of the base monad and a module containing just the type of the state, and produces a minimal definition of the transformed monad.

## 3. Corner Cases

Some functions in Haskell involve multiple classes, such as `foldM`:

```
foldM :: (Foldable t, Monad m)
      => (b -> a -> m b) -> b -> t a -> m b
```

We formalise these as ML `functor`s that take multiple modules conforming to the appropriate signatures and return a module containing the function.

We can easily extend our approach to other constructor classes, even if they involve multiple inheritance. Figure 2 shows an example of such a case. Since our library is based on statically known mathematical properties, we avoid so-called *diamond problem*s. For instance, given a type constructor of `MONADOP` in Figure 2, it does not matter semantically from which of `ALTER` and `MONAD` this type constructor inherits the methods of `APPLICATIVE`, as both of them have the same properties.

## 4. Comparison and Related Work

Our approach offers some benefits over traditional Haskell type classes. In particular, the ML module system allows more flexibility, as more than one instance can be provided for a given type. This flexibility is appreciated in constructor classes, too — for example, there are two perfectly valid `Applicative` instances for lists, one with a cartesian and one with a pairwise product operation. In Haskell, this necessitates the use of the `newtype` feature for one of the instances. In ML, both instances are equally natural.

Wehr *et al.* [1] first introduced an approach to translate Haskell type classes in ML modules. They discussed that their scheme is not able to handle constructor classes, nor translate either recursive class constraints or default definitions into ML modules, while we addressed all of these. One example of a recursive class constraint would be:

```
instance (Monad f, Monad g) => Monad (f :*: g)
```

We express these using ML `functor`s: in this case, we define a `functor mk_ConsProd`, which takes two modules of `MONAD_MIN` and returns a module of `MONAD_MIN`. Even though we can define `mk_ConsProd` parametrically, two concrete type constructors `f` and `g` must be supplied in order to instantiate `MONAD_MIN` for `f :*: g`.

Our approach is similar to the library code in Dreyer *et al.* [2]; however, we additionally support constructor classes, instance elaboration, and automatic instantiation of superclasses. We did not, however, extend the language as they did, as we did not wish to deviate from Standard ML, although we foresee no fundamental problems incorporating their implicit typing scheme into our library. Furthermore, we chose to express class hierarchies with flat module structures, while they did so hierarchically. Our choice allows users to avoid nested qualifiers, *e.g.* `ListMonad.Applicative.Fmap.<$`, resulting in less verbose code in the absence of any implicit typing mechanism.

Scott [4] seems to have employed a similar approach to ours, but in OCaml, suggesting that our technique is transferable to other ML dialects. There are also attempts to model type and constructor classes using features from the imperative object oriented programming paradigm. We purposefully avoided these deviations from Standard ML.

## 5. Current Status and Future Work

We previously developed [5] a proof automation tool for Isabelle using this library, and our experience with it was positive. However, every library has room for improvement. We are working to include other constructor classes such as `Arrow` into this framework. In our approach, our `MONAD` module could also generate an instance of `ARROW`, once again eliminating the Haskell use of `newtype` for `Kleisli` arrows.

Furthermore, we plan to support multiple minimal definitions to instantiate some constructor classes. For example, we presented a minimal definition of `MONAD` with `return` and `bind` above, but we could provide a minimal definition of `MONAD` with `return`, `fmap`, and `join` instead. It is up to the user's preference which minimal definition is easier to write. Since they are equivalent, we can write a `functor` that derives one from the other, providing multiple options to users.

## Acknowledgments

We thank Gabriele Keller for stimulating discussions on ways of representing constructor classes with modules. NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

## References


[1] S. Wehr and M. M. T. Chakravarty ML Modules and Haskell Type Classes: A Constructive Comparison (2008) APLAS

[2] D. Dreyer, R. Harper, M. M. T. Chakravarty, and G. Keller. Modular type classes (2007) POPL

[3] O. Kuncar and A. Popescu. A Consistent Foundation for Isabelle/HOL (2015) ITP

[4] P. Scott. `ocaml-monad` library. `https://github.com/Chattered/ocaml-monad`

[5] Y. Nagashima and R. Kumar. A Proof Strategy Language and Proof Script Generation for Isabelle (2016) arXiv.org 1606.02941