The first part is mostly for ATS programmers that maybe come from a more C-like background and the second part is more geared for newbie converts from Haskell, like myself.

1 Monads and comonads in general

The role of monads in programming can be introduced in many ways, e.g., as a way to simulate mutable state and/or side effects in a pure language with immutable data, such as Haskell. While these perspectives do have merit, I instead want to push the perspective that we use monads because we are interested in Kleisli categories.

Consider something like

```
typedef M(a: type) = ...
```

and a function type `a -> M b` . The trick to understanding Kleisli categories is to (in one's mind) parse this expression as `a (-> M) b`, that is to say, to think of `M` as annotating the arrow. (ATS has several arrow annotations, like `-<cloref1>` or `-<wrt>`; perhaps we should write `-<M>`.) What this means more concretely is that a term of type `a -> Option(b)`, for example, is not to be interpreted as an ordinary function from `a` to `Option(b)`, but as a particular flavor of function *to* `b`. If we insist on the "`M`-flavored" functions to behave like ordinary functions (in particular, they should be possible to compose and to compose associatively), then, clearly, `M` must carry some extra structure: that of a monad. One definition is the following: there should be functions

```
fn bind{a, b: type}(x: M(a), f: a -> M(b)): M(b)
```

and

```
fn return{a: type}(x: a): M(a)
```

satisfying the three monad laws:

```
bind(return(x), f) = f(x)
bind(mx, return) = mx
bind(mx, lam(x) => bind(f(x), g)) = bind(bind(mx, f), g)
```

Then we can compose functions `f: a -> M(b)` and `g: b -> M(c)` to a function `h: a -> M(c)` by the formula `h(x) = bind(f(x), g)`. Ok, so why would one be interested in monads, or more to the point, why would one want to consider "`M`-flavored" functions? The `Option`-monad offers an accessible example. Taking the head of a list is sensibly interpreted as a function that possibly returns "none". Another example is that monads can be used to translate impure functions into pure functions. Consider the function (in slight pseudocode) which uses a single fixed memory cell `l` referencing an integer, given by

```
f(x) = let val y = !l in l := x; y end
```

It depends on a "state" variable and has the side-effect that it updates the state. A more transparent formulation would be to type the function as a function of type `int -> (int -> (int, int))`, that maps a value `x` to the "stateful computation" that takes an initial state `s0` to a pair consisting of a new state and value `(y, s1)`, where `s1 = x`. This is the so-called "state passing translation" and it is based on a monad. The state monad (with integers as state values) has the following presentation:

```
typedef State(a: type) = int -> (a, int)
```

Hence, the (state passing translation of the) function `f` is a function

```
f: int -> State(int)
```

thus; a function of "`State`-flavour" from integers to integers.

Another example of a monad is the writer monad. It depends on a choice of monoid type, i.e., something with an append-like function. If we choose `string` as our monoid, then:

```
typedef Writer(a: type) = (a, string)
```

The bind operation concatenates strings. We can now define functions such as (again pseudo-code)

```
fn add_two(x: int): Writer(int) =
  (x+2, append("I added 2 to", to_string(x)))
```

This is a "`Writer`-function". Kleisli composition will record a log of what the functions "write". There are numerous other, sometimes surprising examples, of how monadic flavors of function occur naturally in functional programming. Wether they really do make coding clearer or not can be argued. Before ending this section, lets turn briefly to comonads. A comonad W is again a typeconstructor, but now it is asked to flavor functions by putting it on the domain rather than the codomain: `W(a) -> b`. For it to be possible to compose such functions in a sensible way (the buzz word is "category theoretical way") one again needs two functions:

```
extend: (W(a) -> b, W(a)) -> W(b)
extract: W(a) -> a
```

Essentially, one just reverses the directions of function arrows in the operations and laws governing monads, e.g., `extract` is dual to `return` and so on. Whereas monads give flavors of function that in some way enlarge the possible function values (encoding things such as optional "none"-values, updating state or writing a log), comonads enlarge the notion of input dependency. Briefly, one could say that monads are for effectful functions and comonads are for context-dependent functions.

2 (Co)monads in ATS?

I do not believe there is any need for more monads in ATS. It is a language that admits arbitrary effects and mutable data and as such it has no real reliance on monadic solutions, the way a language like Haskell does. But note that the expressive type system of ATS allows some monads to be better implemented (both semantically better and with better optimization) than in Haskell, Scala and most other functional languages. For example, state is probably in many cases best regarded as irreversible, so that the state monad

```
State(a) = s -> (a, s)
```

would be modelled on a linear type `s`, with state actions `lam(z) => (f(z), g(z))` where `g` is a linear function.

However, it seems to me that comonads could potentially find better use in ATS than in most other functional languages. Programming with views and viewtypes can incur some expenditure, yet most of it is book-keeping – and possibly book-keeping that can be automated by packaging it in comonadic terms. Recall that comonads are for enlarging what counts as inputs: proof-variables seem like just such a thing. In fact, every view `v` defines a comonad, which is a sort of peculiar version of what one other languages is called the writer comonad. Consider a type construction of the form `W(a) = (v| a)` for a fixed view `v`, with `extract` operation given by projection onto `a`. (We want to allow `a` to be a viewtype, so that, really, `(v| a) = (v, w| b)` where `a = (w| b)` and `b` is nonlinear.) The extend function is given by the trivial formula:

```
extend(f, (pf| x)) = (pf| f(pf| x))
```

The caveat here, which seems philosophically wrong, is that the function call `f(pf| x)` cannot destroy the view-term `pf`, or rather, that it must be duplicated/cloned by extend. Note that the regular version of the writer comonad, often called the `Env`-comonad is to some extent already implicitly incorporated in ATS: a function `Env(a) = (env, a) -> b` is a function depending on an implicitly passed "environment" type.

Let me give another basic example. (I am a total beginner at ATS and haven't come up with something fancier yet.)

```
vtypedef P(a: vt0ype) = [l: agz](a@l, mfree_gc_v(l)| ptr(l))

fn extract{a: vt0p}(xi: P(a)): a = let
    val (pfat, pfgc| p) = xi
    val x = !p
  in ptr_free(pfgc, pfat| p); x
  end

fn extend{a, b: vt0p}(f: P(a) -> b, xi: P(a)): P(b) = let
    val (pfat, pfgc| p) = ptr_alloc<b>()
    val y = f(xi)
```

```
in $effmask_wrt(ptr_set<b>(pfat| p, y)); (pfat, pfgc| p)
end
```

These definitions constitute a comonad. The type `P(a)` is a viewtype, meaning it is linear, and it parametrizes "safe pointers". The syntax for its definition can be read as saying that a term of this type is something "for which there exists a nonzero memory address `l` (the syntax `[l: agz]` means existential quantification over the dependent type `agz = [l: addr | l > null]`) such that we have a triple `(pfat, pfgc| p)`, where `pfat` is a (linear) proof variable witnessing that the address points to something of type `a` (the syntax `a@l`), `pfgc` is a witness that the memory can be deallocated, and `p` is a pointer to the address. The function `extract` extracts the term of `a` pointed to (and frees the pointer), while the function `extend` lifts a value to a pointer equipped with corresponding proof-terms. Some remarks:

The function `extract` is a prime example of how I see comonadic functions `W(a) -> b` entering in a uniquely ATS way. Instead of "just" the function that maps a pointer to the value pointed to, `extract` requires extra input encapsulated in proof-terms. Indeed, proof-terms are erased after compilation, solidifying the justification for regarding them as "flavoring" what is meant by function rather than as truly changing the type of input.

The formula

```
W_map(f)(wx) = extend(lam(wy) => f(extract(wy)), wx)
```

shows that every comonad `W` is a functor. By using the induced functoriality of `P` one lift operations on base types to pointers, thereby reducing the need to unpack and repackage the pointers and proof-terms. Admittedly, the ATS compiler can accomplish many implicit such conversions on its own.

The type `P` can also be made into a monad. However, this seems less natural to me, because, as explained earlier, proof-terms extend what we consider as inputs to functions (indeed, some functions require them!); thus the Kleisli arrows should be comonadic.