

1 |----- MODULE *MonadsAndPipes* -----|  
 2 | EXTENDS *Sequences, Naturals* |

The goal of this module is to explore the similarity between monads and pipes (as defined below).

The lemmas/theorems/assumptions are provided without proof (neither formal nor informal), and could be wrong. A mistake could invalidate the relationship in some or all cases, but could also be fixable.

There could also be technical mistakes in the spec, as no formal verification has been performed, and I haven't gone over everything carefully, and I may have been sloppy.

The main intuition is this: When a monadic composition is *run*, the monadic function is invoked zero or more times with an input value of type *A*. Note that we're looking at the operation of the monad when it is run – not when it's created – as some monads (*e.g.* *Reader* and *Cont*) never invoke the monadic function when the monad is constructed.

A monadic function would correspond to a process, and an invocation to an *A* value being sent to it over its input channel. This is very similar to pipes, but it is not enough. In order to show that monads “are” pipes, the transformation must be compositional with respect to the syntax, *i.e.*, we must show how a specific monadic function *F* is transformed into a corresponding process.

Note that as since we're showing that monads are pipes but not the converse, we're allowed to expansively generalize monads beyond their precise definition as if all “generalized monads” are pipes, then so are all “true” monads. However, we must not narrow the specification, excluding any true monads. Where I fear I may be narrowing, I explicitly state so.

36 |-----|  
 Utility definitions

41  $Range(f) \triangleq \{f[x] : x \in DOMAIN\ f\}$   
 43  $AddFirst(seq, x) \triangleq \langle x \rangle \circ seq$   
 45  $Last(seq) \triangleq seq[Len(seq)]$   
 47  $RemoveN(seq, n) \triangleq SubSeq(seq, 1, Len(seq) - n)$   
 49  $RemoveLast(seq) \triangleq RemoveN(seq, 1)$   
 51  $Reverse(seq) \triangleq [i \in 1 .. Len(seq) \mapsto seq[Len(seq) - i + 1]]$   
 52 |-----|

We begin by specifying the monadic type  $M(A)$  through its constructors. We do it this way because it will be necessary when looking at the **operational** semantics of a *bind*.

(Constants are module parameters that don't dynamically change over time.)

62 CONSTANTS *Constructors*(-)

We assume a monadic value is constructed by a set of constructors. A constructor is a function of a (possibly empty) finite list of values in *A*. The constructors don't correspond with the data constructors of the monadic value, rather, they are only functions of a list of *A*s, and so “reify” any other arguments, and there can be infinitely many of them.

For example, the *Maybe* monad would only have two constructors here, but the *Either* monad would have one corresponding to *Right*, and infinitely many corresponding to *Left* (one for each possible value). *Reader* also has infinitely many constructors, one for each function  $Env \rightarrow A$ .

77 ASSUME  $\forall A : \forall cons \in Constructors(A) : \text{DOMAIN } cons \subseteq Seq(A)$   
 78  $(Seq(A) \text{ is the set of all finite sequences with elements in } A)$

80 Our monadic type  
 81  $M(A) \triangleq \text{UNION } \{Range(cons) : cons \in Constructors(A)\}$

83 *Constructors'* ranges are disjoint (but cover all of  $M(A)$ )  
 84 ASSUME  $\forall A : \forall c1, c2 \in Constructors(A) : c1 \neq c2 \Rightarrow Range(c1) \cap Range(c2) = \{\}$

There's actually more we can say about the de/constructors and other functions here. That they're parametric means that the parametericity theorem ("free theorems") holds. But we won't bother specifying it.

We'll later define our *bind* using a monoidal *compose* :

96 CONSTANTS *Return*( $\_$ ),  
 97 *Compose*( $\_$ )  
 99 ASSUME  $\forall A : Return(A) \in [A \rightarrow M(A)]$   
 101  $IsUnit(unit, A) \triangleq$   
 102  $\forall ma \in M(A) : \wedge Compose(A)[ma, unit] = ma$   
 103  $\wedge Compose(A)[unit, ma] = ma$   
 105  $Unit(A) \triangleq \text{CHOOSE } unit \in M(A) : IsUnit(unit, A)$   
 107 ASSUME  $\forall A :$   
 108 LET  $compose \triangleq Compose(A)$   
 109 IN  $\wedge compose \in [M(A) \times M(A) \rightarrow M(A)]$   
 110  $\wedge \exists unit \in M(A) : IsUnit(unit, A)$   
 111  $\wedge \forall x, y, z \in M(A) : compose[compose[x, y], z] = compose[x, compose[y, z]]$   
 113  $ConsOf(A, ma) \triangleq \text{CHOOSE } cons \in Constructors(A) : ma \in Range(cons)$   
 114

115

MODULE *RunBind*

We'll specify the operational semantics of a single 'run' of a bind:

$Bind : M\ a \rightarrow (a \rightarrow M\ b) \rightarrow M\ b$

The run is not necessarily the running of the  $>>=$  function, as for some monads (*Reader* , *Cont* ) it does not invoke the monadic function  $F$  at all. Rather, for those monads, this specification describes the operation of their "run" (*i.e.*, *runReader* , *runCont* ).

127 CONSTANT  $A, B$ 129 CONSTANT  $F$  This is the monadic function130 ASSUME  $F \in [A \rightarrow M(B)]$ 

Since the monad is parametric, the only way for us to get values in  $A$  is by deconstructing the monadic value, and then either extracting an  $A$  value directly from it, or performing its "effect" (*e.g.*, for a *Reader* , this is done by applying the function  $e \rightarrow a$  to some ambient environment  $e$  . We then apply  $F$  to any of the created  $A$  values any (finite) number of times.

Bind could rely on the  $M(B)$  value returned from  $F$  to determine further invocations of  $F$  . This may or may not be true for all monads – I haven't thought about it enough (*e.g.* it may or may not break associativity), but if it doesn't, it will require us to add another capability to our pipes.

So in order to express the restriction that we cannot rely on the return value for further invocations, we'll assume there is some function  $Ap$  , that computes a list of  $A$  values to be passed, one by one, to  $F$  from the constructor. The function reifies any environment/effect.

**Note A**

Some special thought must be paid to the *Cont* (continuation) monad. It is sometimes helpful, in order to understand a monad, to expand the definition of the type of the monadic value. For example, for *Reader* ,  $M\ a = e \rightarrow a$  , and so the monadic function  $a \rightarrow M\ b$  becomes  $a \rightarrow e \rightarrow b$  and the function can then be seen as taking a pair of arguments, of type  $\langle a, e \rangle$  . *Reader* poses no special difficulty, but *Cont* does. The monadic value of *Cont* is  $M\ a = (a \rightarrow r) \rightarrow r$  , and so, the monadic function  $a \rightarrow M\ b$  becomes  $a \rightarrow (b \rightarrow r) \rightarrow r$  , for some arbitrary  $r$  . This is the CPS transformation done by *Cont* , as the monadic function can be seen to take an  $a$  argument, as well as a continuation  $b \rightarrow r$  . It is the responsibility of the monadic function to supply the continuation with a  $b$  value (so it's a CPS transformation of the function composition  $(a \rightarrow b) \circ (b \rightarrow r)$  ).

What makes *Cont* special is that it is the monadic function itself, when viewed as taking a pair of a value and a continuation, that decides how many times (if at all) to apply the continuation. However, as the continuation returns an value of type  $r$  unknown to the monadic function, the function cannot use the  $r$  value to decide how many times to apply the continuation – it can only rely on itself, and it communicates this through the constructor, which can be seen as the reification of curried part of the function  $a \rightarrow (b \rightarrow r) \rightarrow r$  .

So, for *Cont* ,  $Ap(B)[mb]$  is the (possibly empty) list of  $B$  values that will be passed to the continuation.

179 CONSTANT  $Ap(-)$ 180 ASSUME  $\forall T : Ap(T) \in [M(T) \rightarrow Seq(T)]$ 182 VARIABLES  $x$ , The monadic input

183  $as$ , The values that  $F$  will be consecutively applied to  
 184  $mb$ , The most recent return value from  $F$   
 185  $y$  The “current” monadic value in  $M(B)$   
 187  $vars \triangleq \langle x, y, as, mb \rangle$

By convention,  $TypeOK$  is a “type” invariant on all variables

192  $TypeOK \triangleq \wedge x \in M(A)$   
 193  $\wedge as \in Seq(A)$   
 194  $\wedge mb \in M(B)$   
 195  $\wedge y \in M(B)$

When we begin,  $xs$  are the “extraction” (with  $Ap$ ) of the first argument to bind.

200  $Init \triangleq \wedge \exists ma \in M(A) : as = Ap(A)[ma]$   
 201  $\wedge y = Unit(B)$   
 203  $Next \triangleq \wedge as' \neq \langle \rangle$   
 204  $\wedge as' = Tail(as)$   
 205  $\wedge LET a \triangleq Head(as)$   
 206  $IN \wedge mb' = F[a]$   
 207  $\wedge y' = Compose(B)[y, mb']$   
 208  $\wedge UNCHANGED x$

210  $Spec \triangleq Init \wedge \Box [Next]_{vars}$

212 THEOREM  $Spec \Rightarrow \Box TypeOK$

But what is the relationship between this operational description of a “running” monad, and the denotational (and ordinary) meaning of (mathematical) monads? It is one of abstraction.

As  $TLA^+$  lets us express any abstraction mathematically, we can show this relationship precisely. Expressing the monad denotationally allows us to specify the monad laws.

222  $MonadLaws(bind) \triangleq$   
 223  $\wedge bind \in [M(A) \times [A \rightarrow M(B)] \rightarrow M(B)]$   
 224  $\wedge \forall a \in A, f \in [A \rightarrow M(B)] : bind[Return(A)[a], f] = f[a]$   
 225  $\wedge A = B \Rightarrow \forall ma \in M(A) : bind[ma, Return(A)] = ma$   
 226  $\wedge LET kl[f \in [A \rightarrow M(A)], g \in [B \rightarrow M(B)]] \triangleq [a \in A \mapsto bind[f[a], g]]$   
 227  $IN \forall C : \forall f \in [A \rightarrow M(A)], g \in [B \rightarrow M(B)], h \in [C \rightarrow M(C)] :$   
 228  $kl[kl[f, g], h] = kl[f, kl[g, h]]$

230 For any initial value of  $x$ ,  $y$  would eventually settle on the result of a monadic  $bind$

231  $MonadDenotation \triangleq$   
 232  $\exists bind \in [M(A) \times [A \rightarrow M(B)] \rightarrow M(B)] :$   
 233  $\wedge MonadLaws(bind)$   
 234  $\wedge \forall ma \in M(A) : (x = ma) \wedge Spec \Rightarrow \Diamond \Box (y = bind[ma, F])$

235  $\boxed{\hspace{15cm}}$   
 236

To talk about pipes, we must first define them. In order to deserve the name, they must resemble POSIX-style shell pipes:

$p_1 | p_2 | \dots | p_n$

We usually think of pipes as letting data values flow, so our processes will emit a stream of only simple (nonmonadic) data values, and, in addition, possibly perform some effect (like writing/reading a shared file).

248  $ReaderWriter(A, in, out) \triangleq$  Specifies an “environment” that can write to  $in$  and read from  $out$  .  
 249  $\vee \wedge \exists x \in Seq(A) : in' = x \circ in$   
 250  $\wedge UNCHANGED\ out$   
 251  $\vee \wedge \exists n \in 0 \dots Len(out) : out' = SubSeq(out, 1, n)$   
 252  $\wedge UNCHANGED\ in$

254  $\boxed{\text{MODULE } Process}$   
 A process is given as a transformation of a monadic function  $F$  and the following  $bind$  .  
 The interpretation of the constructor can be  $a(n\ unmodeled)$  side-effect, such as adding a line to a log file (*Writer* ) , read/write some state in a shared file (*State* ) etc.

263 CONSTANTS  $A, B$   
 264 CONSTANT  $F$   
 265 ASSUME  $F \in [A \rightarrow M(B)]$

To transform  $F$  into a process, the process itself must also make use of the “extraction function”  $Ap(A)$  , for example for  $Cont$  (see **Note A** above).

272 CONSTANT  $Ap(-)$   
 273 ASSUME  $Ap(A) \in [M(A) \rightarrow Seq(A)]$

275 VARIABLES  $in, out$  LIFO channels

277  $TypeOK \triangleq in \in Seq(A) \wedge out \in Seq(B)$

279  $Environment \triangleq ReaderWriter(A, in, out)$

281  $Init \triangleq \wedge in \in Seq(A)$   
 282  $\wedge out \in Seq(B)$

284  $Compute \triangleq \wedge in \neq \langle \rangle$   
 285  $\wedge in' = RemoveLast(in)$   
 286  $\wedge LET\ mb \triangleq F[Last(in)]$  The effect also taked place here  
 287  $IN\ out' = Ap(B)[mb] \circ out$

289  $Next \triangleq Compute \vee Environment$

291  $Spec \triangleq Init \wedge \Box [Next]_{\langle in, out \rangle}$

293 THEOREM  $Spec \Rightarrow \Box TypeOK$

294  $\boxed{\text{END MODULE } Process}$

```

296 |----- MODULE Pipe -----|
    | An example of composing two processes with a pipe – we simply let the first output into the |
    | other's input. |
301 | CONSTANTS A, B, C
303 | CONSTANT F, G, Ap(-)
305 | VARIABLES in, shared, out
307 | Process1  $\triangleq$  INSTANCE Process WITH out  $\leftarrow$  shared
308 | Process2  $\triangleq$  INSTANCE Process WITH in  $\leftarrow$  shared, F  $\leftarrow$  G, A  $\leftarrow$  B, B  $\leftarrow$  C
310 | Spec  $\triangleq$  Process1!Spec  $\wedge$  Process2!Spec
311 |-----|

313 |----- MODULE MonadsArePipes -----|
314 | CONSTANTS A, B
315 | CONSTANT F
316 | CONSTANT Ap(-)
318 | VARIABLES x, as, mb, y

    | The main refinement theorem |

324 | Monad  $\triangleq$  INSTANCE RunBind
326 | Process  $\triangleq$  INSTANCE Process WITH in  $\leftarrow$  Reverse(as),
327 |                                     out  $\leftarrow$  Ap(B)[y] Probably need to reverse something here – details.
329 | THEOREM MonadsArePipes  $\triangleq$  Monad!Spec  $\Rightarrow$  Process!Spec

    | This theorem is provided with no proof, let alone a formal one, just to make the claim clear. |
    | One thing you may notice is that Process does not make use of Compose while RunBind |
    | does. Therefore, the proof of refinement would need to make use of the following theorem, |
    | CompositionOfConstructorArguments , which states that for any constructor the composition of |
    | values yielded by two arg lists is the value yielded from the concatenation of the lists. Is this true? |
    | Partly true? |

343 | THEOREM CompositionOfConstructorArguments  $\triangleq$ 
344 |      $\forall T : \forall xs, ys \in Seq(T), cons \in Constructors(T) :$ 
345 |      $Compose(T)[cons[xs], cons[ys]] = cons[xs \circ ys]$ 
346 |-----|
347 |-----|

```