

```

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.

```

17 |-----|
    | Utility definitions |

```

```

22 | Range(f)  $\triangleq$  {f[x] : x ∈ DOMAIN f}
24 | AddFirst(seq, x)  $\triangleq$  ⟨x⟩ ∘ seq
26 | Last(seq)  $\triangleq$  seq[Len(seq)]
28 | RemoveN(seq, n)  $\triangleq$  SubSeq(seq, 1, Len(seq) - n)
30 | RemoveLast(seq)  $\triangleq$  RemoveN(seq, 1)

```

```

32 |-----|

```

We begin by specifying the monadic type $M(A)$ through its constructors and deconstructors. 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.)

```

42 | CONSTANTS Constructors(-),
43 |           Deconstructors(-)

```

We assume a monadic value is constructed by a set of constructors. A constructor is a function of a 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)

```

56 | ASSUME  $\forall A : \forall cons \in Constructors(A) : DOMAIN\ cons \subseteq Seq(A)$ 

```

```

59 | Our monadic type
60 |  $M(A) \triangleq UNION \{Range(cons) : cons \in Constructors(A)\}$ 

```

```

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

```

```

65 | Deconstructors' domains are disjoint, but cover all of  $M(A)$ 
66 | ASSUME  $\forall A :$ 

```

```

67       $\wedge \forall des \in Deconstructors(A) : \exists SMA \in SUBSET\ M(A) : des \in [SMA \rightarrow Seq(A)]$ 
68       $\wedge \forall d1, d2 \in Deconstructors(A) : d1 \neq d2 \Rightarrow DOMAIN\ d1 \cap DOMAIN\ d2 = \{\}$ 
69       $\wedge UNION\ \{DOMAIN\ des : des \in Deconstructors(A)\} = M(A)$ 

71  ASSUME  $\forall A :$ 
72       $\wedge \forall des \in Deconstructors(A) : \exists cons \in Constructors(A) : \forall ma \in M(A) : cons[des[ma]] = ma$ 
73       $\wedge \forall cons \in Constructors(A) : \exists des \in Deconstructors(A) : \forall as \in DOMAIN\ cons : des[cons[as]] = as$ 

```

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*

```

85  CONSTANTS Return(-),
86            Compose(-)

88  ASSUME  $\forall A : Return(A) \in [A \rightarrow M(A)]$ 

90  IsUnit(unit, A)  $\triangleq$ 
91       $\forall ma \in M(A) : \wedge Compose(A)[ma, unit] = ma$ 
92       $\wedge Compose(A)[unit, ma] = ma$ 

94  Unit(A)  $\triangleq$  CHOOSE unit  $\in M(A) : IsUnit(unit, A)$ 

96  ASSUME  $\forall A :$ 
97      LET compose  $\triangleq Compose(A)$  IN
98       $\wedge compose \in [M(A) \times M(A) \rightarrow M(A)]$ 
99       $\wedge \exists unit \in M(A) : IsUnit(unit, A)$ 
100      $\wedge \forall x, y, z \in M(A) : compose[compose[x, y], z] = compose[x, compose[y, z]]$ 

```

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?

```

107 THEOREM  $\forall A : \forall xs, ys \in Seq(A), cons \in Constructors(A) :$ 
108      $Compose(A)[cons[xs], cons[ys]] = cons[xs \circ ys]$ 

110 ConsOf(A, ma)  $\triangleq$  CHOOSE cons  $\in Constructors(A) : ma \in Range(cons)$ 

112 Deconstruct(A, ma)  $\triangleq$  LET des  $\triangleq$  CHOOSE des  $\in Deconstructors(A) : ma \in DOMAIN\ des$ 
113     IN des[ma]

115

```

116
MODULE *Bind*

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

$$\text{Bind} : M\ a \rightarrow (a \rightarrow M\ b) \rightarrow M\ b$$

123 CONSTANT A, B

125 CONSTANT F This is the monadic function

126 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. However we can apply F to any of them 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 monadic value input. Ap is not the same as *Deconstruct*, as it can also depend on the constructor.

146 CONSTANT $Ap(-)$

147 ASSUME $\forall T : Ap(T) \in [M(T) \rightarrow Seq(T)]$

149 VARIABLES $x,$

150 $as,$

151 $mb,$

152 y

The monadic input

The values that F will be consecutively applied to

The most recent return value from F

The “current” monadic value in $M(B)$

154 $vars \triangleq \langle x, y, as, mb \rangle$

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

159

160

161

162

$TypeOK \triangleq$

$\wedge x \in M(A)$

$\wedge as \in Seq(A)$

$\wedge mb \in M(B)$

$\wedge y \in M(B)$

When we begin, xs are the deconstruction (with Ap of the first argument to bind.

169 $Init \triangleq$

170 $\wedge \exists ma \in M(A) : as = Ap(A)[ma]$

$\wedge y = Unit(B)$

172 $Next \triangleq$

173 $\wedge as' \neq \langle \rangle$

$\wedge as' = Tail(as)$

174 $\wedge LET\ a \triangleq Head(as)$

175 $IN\ \wedge mb' = F[a]$

176 $\wedge y' = Compose(B)[y, mb']$

3

177 $\wedge \text{UNCHANGED } x$

179 $\text{Spec} \triangleq \text{Init} \wedge \Box[\text{Next}]_{\text{vars}}$

181 THEOREM $\text{Spec} \Rightarrow \Box \text{TypeOK}$

We can talk about the denotation of our spec in the functional denotational semantics of *FP*.
We'll use this opportunity to specify the monad laws

188 $\text{MonadLaws}(\text{bind}) \triangleq$
 189 $\wedge \text{bind} \in [M(A) \times [A \rightarrow M(B)] \rightarrow M(B)]$
 190 $\wedge \forall a \in A, f \in [A \rightarrow M(B)] : \text{bind}[\text{Return}(A)[a], f] = f[a]$
 191 $\wedge A = B \Rightarrow \forall ma \in M(A) : \text{bind}[ma, \text{Return}(A)] = ma$
 192 $\wedge \text{LET } kl[f \in [A \rightarrow M(A)], g \in [B \rightarrow M(B)]] \triangleq [a \in A \mapsto \text{bind}[f[a], g]]$
 193 $\text{IN } \forall C : \forall f \in [A \rightarrow M(A)], g \in [B \rightarrow M(B)], h \in [C \rightarrow M(C)] :$
 194 $kl[kl[f, g], h] = kl[f, kl[g, h]]$

196 $\text{MonadDenotation} \triangleq$
 197 $\exists \text{bind} \in [M(A) \times [A \rightarrow M(B)] \rightarrow M(B)] :$
 198 $\wedge \forall ma \in M(A) : (x = ma) \wedge \text{Spec} \Rightarrow \Diamond \Box (y = \text{bind}[ma, F])$
 199 $\wedge \text{MonadLaws}(\text{bind})$

202

204

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

$p_1|p_2|\dots|p_n$

In this first attempt, a pipes input (and a process's output) is a stream of monadic values. We'll later address that.

Intuitively, you can think of the process as reading values of type A and emitting values of type $M(A)$, and the pipes as doing the converse.

218 Specifies some component that can write to in and read from out

219 $ReaderWriter(A, in, out) \triangleq$

220 $\vee \wedge \exists x \in Seq(A) : in' = x \circ in$

221 $\wedge \text{UNCHANGED } out$

222 $\vee \wedge \exists n \in 0 \dots Len(out) : out' = SubSeq(out, 1, n)$

223 $\wedge \text{UNCHANGED } in$

225 $\text{MODULE } Process1$

227 CONSTANT F

229 VARIABLES in, out LIFO channels

231 $Init \triangleq in \in Seq(\text{DOMAIN } F)$

233 $Compute \triangleq \wedge in \neq \langle \rangle$

234 $\wedge in' = RemoveLast(in)$

235 $\wedge out' = AddFirst(out, F[Last(in)])$

237 $Environment \triangleq ReaderWriter(\text{DOMAIN } F, in, out)$

239 $Next \triangleq Compute \vee Environment$

241 $Spec \triangleq Init \wedge \Box[Next]_{\langle in, out \rangle}$

243

246 $\text{MODULE } Pipe1$

Note that in a composition chaining a single pipe and a single bind don't perform the same function. A pipe performs half the function of the previous bind and half of that of the next.

254 CONSTANT A

256 CONSTANT $Ap(-)$

257 ASSUME $Ap(A) \in [M(A) \rightarrow Seq(A)]$

259 VARIABLES $in,$ The channel

260 $out,$

```

261      state   Our current state
263  vars  $\triangleq \langle in, out, state \rangle$ 
265  TypeOK  $\triangleq \wedge in \in Seq(M(A))$ 
266            $\wedge state \in M(A)$ 
267            $\wedge out \in Seq(A)$ 
269  Init  $\triangleq \wedge in \in Seq(M(A)) \wedge in \neq \langle \rangle$ 
270            $\wedge state = Last(in)$ 
271            $\wedge out \in Seq(A)$ 
273  Receive  $\triangleq \wedge in \neq \langle \rangle$ 
274            $\wedge in' = RemoveLast(in)$ 
275            $\wedge state' = Compose(A)[state, Last(in)]$ 
277  Send  $\triangleq \wedge in = \langle \rangle$ 
278            $\wedge out' = Ap(A)[state] \circ out$ 
279            $\wedge UNCHANGED\ state$ 
281  Environment  $\triangleq \wedge ReaderWriter(A, in, out)$ 
282            $\wedge UNCHANGED\ state$ 
284  Next  $\triangleq Receive \vee Send \vee Environment$ 
286  Spec  $\triangleq Init \wedge \Box[Next]_{vars}$ 
288  THEOREM Spec  $\Rightarrow \Box TypeOK$ 
290  ───────────────────────────────────────────────────────────────────────────────────
292  ─────────────────────────────────── MODULE PipesAndProcess1 ───────────────────────────────────
    We compose two pipes and a process ( | p | )
298  CONSTANT A, B
299  CONSTANT F
300  CONSTANT Ap1(-), Ap2(-)
302  VARIABLES in1, state1,
303           p_in, p_out,
304           out2, state2
306  Pipe1  $\triangleq$  INSTANCE Pipe1   WITH Ap  $\leftarrow Ap1$ , in  $\leftarrow in1$ , state  $\leftarrow state1$ , out  $\leftarrow p\_in$ 
307  Process  $\triangleq$  INSTANCE Process1 WITH in  $\leftarrow p\_in$ , out  $\leftarrow p\_out$ 
308  Pipe2  $\triangleq$  INSTANCE Pipe1   WITH A  $\leftarrow B$ ,
309           Ap  $\leftarrow Ap2$ , in  $\leftarrow p\_out$ , state  $\leftarrow state2$ , out  $\leftarrow out2$ 
311  Spec  $\triangleq Pipe1!Spec \wedge Process!Spec \wedge Pipe2!Spec$ 

```

```

313 |_____|
315 |_____ MODULE MonadsArePipes1 _____|
317 CONSTANTS A, B
318 CONSTANT F
319 CONSTANT Ap1(_), Ap2(_)
321 VARIABLES x, as, mb, y

The main refinement theorem

328 Monad  $\triangleq$  INSTANCE Bind WITH Ap  $\leftarrow$  Ap1
330 Pipes(out2, p_out)  $\triangleq$ 
    We hide out2, because, being part of the function of the next bind
    in the chain, it is not modeled by this monad instance.
    We also hide p_out, as it's not directly modeled by our Bind .
338     INSTANCE PipesAndProcess1 WITH in1  $\leftarrow$   $\langle x \rangle$ ,
339         state1  $\leftarrow$   $\langle x \rangle$ ,
340         p_in  $\leftarrow$  as,
341         state2  $\leftarrow$  y
343 THEOREM MonadsArePipes1  $\triangleq$   $\exists$  out2, p_out : Monad!Spec  $\Rightarrow$  Pipes(out2, p_out)!Spec
345 |_____|
347

```

The pipes so far may have been a little disappointing because we usually think of pipes as letting data values flow, but here processes emit monadic values.

We can do better. Our processes will emit a stream of only simple (nonmonadic) data values, and, in addition, pass a single control value to the pipe (perhaps like a signal or line on stderr).

Intuitively, think of the control value a process emits as a constructor of a monadic value, and the data values it emits as the arguments to that constructor.

362 A tombstone value, which will be used later. Must not be a constructor
 363 $EMPTY \triangleq \text{CHOOSE } x : \forall A : x \notin \text{Constructors}(A)$

365 $\text{MODULE } Process2$

367 CONSTANTS A, B

368 CONSTANT F

370 ASSUME $F \in [A \rightarrow M(B)]$

372 VARIABLES $in, out,$

373 $control$

To simplify matters, we assume that the consumer (pipe) reads all elements from out and sets $control_out$ to $EMPTY$ at once.

380 $Environment \triangleq \wedge \text{ReaderWriter}(\text{DOMAIN } F, in, out)$
 381 $\wedge \vee \text{UNCHANGED } out$
 382 $\vee control' = EMPTY$

384 $TypeOK \triangleq \wedge in \in Seq(A)$
 385 $\wedge out \in Seq(B)$
 386 $\wedge control \in \text{Constructors}(B) \cup \{EMPTY\}$

388 $Init \triangleq \wedge in \in Seq(\text{DOMAIN } F)$
 389 $\wedge out = \langle \rangle$
 390 $\wedge control = EMPTY$

392 $Compute \triangleq \wedge in \neq \langle \rangle$
 393 $\wedge in' = \text{RemoveLast}(in)$
 394 $\wedge \text{LET } mb \triangleq F[\text{Last}(in)]$
 395 $\text{IN } \wedge out' = \text{Deconstruct}(B, mb) \circ out$
 396 $\wedge control' = \text{ConsOf}(B, mb)$

398 $Next \triangleq Compute \vee Environment$

400 $Spec \triangleq Init \wedge \square[Next]_{\langle in, out, control \rangle}$

402 THEOREM $Spec \Rightarrow \square TypeOK$


```

404 |_____|
406 |_____ MODULE Pipe2 _____|

    Note that in a composition chaining a single pipe and a single bind don't perform the same function.
    A pipe performs half the function of the previous bind and half of that of the next.

414 CONSTANT A

416 CONSTANT Ap(-)
417 ASSUME  $Ap(A) \in [M(A) \rightarrow Seq(A)]$ 

419 VARIABLES in,
420             control,
421             collect,
422             state,
423             out

425 vars  $\triangleq \langle in, control, collect, state, out \rangle$ 

427 TypeOK  $\triangleq \wedge control \in Constructors(A) \cup \{EMPTY\}$ 
428            $\wedge in \in Seq(A)$ 
429            $\wedge out \in Seq(A)$ 
430            $\wedge collect \in Seq(A)$ 
431            $\wedge state \in M(A)$ 

433 Environment  $\triangleq \wedge ReaderWriter(A, in, out)$ 
434            $\wedge \vee UNCHANGED in$ 
435            $\vee control' \neq EMPTY$ 
436            $\wedge UNCHANGED collect$ 

438 Init  $\triangleq \wedge in \in Seq(M(A)) \wedge in \neq \langle \rangle$ 
439            $\wedge control = \langle \rangle$ 
440            $\wedge out \in Seq(A)$ 

442 Receive  $\triangleq \wedge control \neq EMPTY$ 
443            $\wedge UNCHANGED control$ 
444            $\wedge \text{IF } in \neq \langle \rangle \text{ THEN } \wedge in' = RemoveLast(in)$ 
445              $\wedge collect' = AddFirst(collect, Last(in))$ 
446           ELSE UNCHANGED in

448 Send  $\triangleq \wedge \text{LET } cons \triangleq control$ 
449            $ma \triangleq cons[collect]$ 
450           IN  $\wedge state' = Compose(A)[state, ma]$ 
451              $\wedge out' = Ap(A)[state']$ 
452            $\wedge collect' = \langle \rangle$ 
453            $\wedge control' = EMPTY$ 
454            $\wedge UNCHANGED in$ 

```

```

456 Next  $\triangleq$  Receive  $\vee$  Send  $\vee$  Environment
458 Spec  $\triangleq$  Init  $\wedge \square[Next]_{vars}$ 
460
462  $\text{MODULE } PipesAndProcess2$ 
464 CONSTANT A, B
465 CONSTANT F
466 CONSTANT Ap1(-), Ap2(-)
468 VARIABLES in1, control1, state1, collect1,
469             p_in, p_out,
470             control2, collect2, state2, out2
472 Pipe1  $\triangleq$  INSTANCE Pipe2 WITH Ap  $\leftarrow$  Ap1, in  $\leftarrow$  in1, control  $\leftarrow$  control1,
473                                     collect  $\leftarrow$  collect1, state  $\leftarrow$  state1, out  $\leftarrow$  p_in
474 Process  $\triangleq$  INSTANCE Process2 WITH in  $\leftarrow$  p_in, out  $\leftarrow$  p_out, control  $\leftarrow$  control2
475 Pipe2  $\triangleq$  INSTANCE Pipe2 WITH A  $\leftarrow$  B,
476                                     Ap  $\leftarrow$  Ap2, in  $\leftarrow$  p_out, control  $\leftarrow$  control2,
477                                     collect  $\leftarrow$  collect2, state  $\leftarrow$  state2, out  $\leftarrow$  out2
479 Spec  $\triangleq$  Pipe1!Spec  $\wedge$  Process!Spec  $\wedge$  Pipe2!Spec
481
483  $\text{MODULE } MonadsArePipes2$ 
485 CONSTANTS A, B
486 CONSTANT F
487 CONSTANT Ap1(-), Ap2(-)
489 VARIABLES x, as, mb, y

```

The main refinement theorem

```

495  Monad  $\triangleq$  INSTANCE Bind WITH  $Ap \leftarrow Ap1$ 

```

In addition to the variables we hide (intermediate ones and ones *Bind* doesn't model, we also quantify over *control2* and *collect2*, as we only look at them at the moment of *Pipe2* send (i.e., when the monadic value is constructed).

```

503   DuringSend(in)  $\triangleq$  in =  $\langle \rangle$ 
505   Pipes(control2, collect2,
506         control1, collect1, p-out, out2)  $\triangleq$ 
507     INSTANCE PipesAndProcess2
508         WITH in1  $\leftarrow \langle x \rangle$ ,
509             p-in  $\leftarrow as$ ,

```

```

510             state1 ← x,
511             control2 ← IF DuringSend(p_out) THEN ConsOf(B, mb) ELSE control1,
512             collect2 ← IF DuringSend(p_out) THEN Deconstruct(B, mb) ELSE collect1,
513             state2 ← y
515 THEOREM MonadsArePipes2  $\triangleq$ 
516      $\exists$  control2, collect2, control1, collect1, p_out, out2 :
517         Monad!Spec  $\Rightarrow$  Pipes(control2, collect2, control1, collect1, p_out, out2)!Spec
518 ┌──────────────────────────────────────────────────────────────────────────────────┐
519 └──────────────────────────────────────────────────────────────────────────────────┘

```