———— MODULE *MonadsAndPipes* ————

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.

Utility definitions

$Range(f) \triangleq \{f[x] : x \in \text{DOMAIN } f\}$

$AddFirst(seq, x) \triangleq \langle x \rangle \circ seq$

$Last(seq) \triangleq seq[Len(seq)]$

$RemoveN(seq, n) \triangleq SubSeq(seq, 1, Len(seq) - n)$

$RemoveLast(seq) \triangleq RemoveN(seq, 1)$

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.)

CONSTANTS $Constructors(\_)$,
$\quad Deconstructors(\_)$

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 As, and so "reify" any other arguments, and there can be infinitely many of them.

For example, the *Maybe* monad would only have two constuctors here, but the *Either* monad would have one corresponding to *Right* , and infinitely many corresponding to *Left* (one for each possible value)

ASSUME $\forall A : \forall cons \in Constructors(A) : \text{DOMAIN } cons \subseteq Seq(A)$

Our monadic type

$M(A) \triangleq \text{UNION } \{Range(cons) : cons \in Constructors(A)\}$

Constructors′ ranges are disjoint (but cover all of $M(A)$)

ASSUME $\forall A : \forall c1, c2 \in Constructors(A) : c1 \neq c2 \Rightarrow Range(c2) \cap Range(c2) = \{\}$

Deconstructors′ domains are disjoint, but cover all of $M(A)$

ASSUME $\forall A :$

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

$\text{ASSUME}\ \forall\, A :$
$\quad \land \forall\, des \in Deconstructors(A) : \exists\, cons \in Constructors(A) : \forall\, ma \in M(A) : cons[des[ma]] = ma$
$\quad \land \forall\, cons \in Constructors(A) : \exists\, des \in Deconstructors(A) : \forall\, as \in \text{DOMAIN}\ cons : des[cons[as]] = as$

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*

$\text{CONSTANTS}\ Return(\_),$
$\quad\quad Compose(\_)$

$\text{ASSUME}\ \forall\, A : Return(A) \in [A \rightarrow M(A)]$

$IsUnit(unit,\ A) \triangleq$
$\quad \forall\, ma \in M(A) : \land\ Compose(A)[ma,\ unit] = ma$
$\quad\quad\quad\quad \land\ Compose(A)[unit,\ ma] = ma$

$Unit(A) \triangleq \text{CHOOSE}\ unit \in M(A) : IsUnit(unit,\ A)$

$\text{ASSUME}\ \forall\, A :$
$\quad \text{LET}\ compose \triangleq Compose(A) \text{IN}$
$\quad \land\ compose \in [M(A) \times M(A) \rightarrow M(A)]$
$\quad \land \exists\, unit \in M(A) : IsUnit(unit,\ A)$
$\quad \land \forall\, x,\ y,\ z \in M(A) : compose[compose[x,\ y],\ z] = compose[x,\ compose[y,\ z]]$

For any constuctor the composition of values yielded by two arg lists is the value yielded from the concatenation of the lists. Is this true? Partly true?

$\text{THEOREM}\ \forall\, A : \forall\, xs,\ ys \in Seq(A),\ cons \in Constructors(A) :$
$\quad\quad Compose(A)[cons[xs],\ cons[ys]] = cons[xs \circ ys]$

$ConsOf(A,\ ma) \triangleq \text{CHOOSE}\ cons \in Constructors(A) : ma \in Range(cons)$

$Deconstruct(A,\ ma) \triangleq \text{LET}\ des \triangleq \text{CHOOSE}\ des \in Deconstructors(A) : ma \in \text{DOMAIN}\ des$
$\quad\quad\quad\quad \text{IN}\quad des[ma]$

MODULE *Bind*

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

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

CONSTANT $A,\ B$

CONSTANT F This is the monadic function

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.

CONSTANT $Ap(\_)$

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

VARIABLES x, The monadic input
as, The values that F will be consecutively applied to
mb, The most recent return value from F
y The "current" monadic value in $M(B)$

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

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

$$
\begin{aligned}
TypeOK \triangleq\ & \land\ x \in M(A) \\
& \land\ as \in Seq(A) \\
& \land\ mb \in M(B) \\
& \land\ y \in M(B)
\end{aligned}
$$

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

$$
\begin{aligned}
Init \triangleq\ & \land\ \exists\, ma \in M(A) : as = Ap(A)[ma] \\
& \land\ y = Unit(B)
\end{aligned}
$$

$$
\begin{aligned}
Next \triangleq\ & \land\ as' \neq \langle\rangle \\
& \land\ as' = Tail(as) \\
& \land\ \text{LET } a \triangleq Head(as) \\
& \quad\ \text{IN } \land\ mb' = F[a] \\
& \qquad\quad \land\ y' = Compose(B)[y,\ mb']
\end{aligned}
$$

$$\quad \land \text{UNCHANGED } x$$

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

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

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

$$
\begin{array}{l}
MonadLaws(bind) \triangleq \\
\quad \land\ bind \in [M(A) \times [A \rightarrow M(B)] \rightarrow M(B)] \\
\quad \land\ \forall\, a \in A, f \in [A \rightarrow M(B)] : bind[Return(A)[a], f] = f[a] \\
\quad \land\ A = B \Rightarrow \forall\, ma \in M(A) : bind[ma, Return(A)] = ma \\
\quad \land\ \text{LET } kl[f \in [A \rightarrow M(A)], g \in [B \rightarrow M(B)]] \triangleq [a \in A \mapsto bind[f[a], g]] \\
\quad\quad \text{IN } \forall\, C : \forall\, f \in [A \rightarrow M(A)], g \in [B \rightarrow M(B)], h \in [C \rightarrow M(C)] : \\
\quad\quad\quad kl[kl[f, g], h] = kl[f, kl[g, h]]
\end{array}
$$

$$
\begin{array}{l}
MonadDenotation \triangleq \\
\quad \exists\, bind \in [M(A) \times [A \rightarrow M(B)] \rightarrow M(B)] : \\
\quad\quad \land\ \forall\, ma \in M(A) : (x = ma) \land Spec \Rightarrow \Diamond\Box(y = bind[ma, F]) \\
\quad\quad \land\ MonadLaws(bind)
\end{array}
$$

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

$p_1|p_2|...|p_n$

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.

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

$$\begin{aligned}
&ReaderWriter(A, in, out) \triangleq \\
&\quad \lor \land \exists x \in Seq(A) : in' = x \circ in \\
&\quad \land \text{UNCHANGED } out \\
&\quad \lor \land \exists n \in 0 \,..\, Len(out) : out' = SubSeq(out, 1, n) \\
&\quad \land \text{UNCHANGED } in
\end{aligned}$$

MODULE *Process1*

CONSTANT F

VARIABLES in, out *LIFO* channels

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

$$\begin{aligned}
Compute \triangleq\ &\land in \neq \langle\rangle \\
&\land in' = RemoveLast(in) \\
&\land out' = AddFirst(out, F[Last(in)])
\end{aligned}$$

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

$$Next \triangleq Compute \lor Environment$$

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

MODULE *Pipe1*

Note that in a composition chaing 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.

CONSTANT A

CONSTANT $Ap(\_)$

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

VARIABLES $in,$ The channel

$out,$

$state$ Our current state

$vars \triangleq \langle in, out, state \rangle$

$$\begin{aligned} TypeOK \triangleq\ & \land\ in \in Seq(M(A)) \\ & \land\ state \in M(A) \\ & \land\ out \in Seq(A) \end{aligned}$$

$$\begin{aligned} Init \triangleq\ & \land\ in \in Seq(M(A)) \land in \neq \langle\rangle \\ & \land\ state = Last(in) \\ & \land\ out \in Seq(A) \end{aligned}$$

$$\begin{aligned} Receive \triangleq\ & \land\ in \neq \langle\rangle \\ & \land\ in' = RemoveLast(in) \\ & \land\ state' = Compose(A)[state, Last(in)] \end{aligned}$$

$$\begin{aligned} Send \triangleq\ & \land\ in = \langle\rangle \\ & \land\ out' = Deconstruct(A, state) \circ out \\ & \land\ \text{UNCHANGED}\ state \end{aligned}$$

$$\begin{aligned} Environment \triangleq\ & \land\ ReaderWriter(A, in, out) \\ & \land\ \text{UNCHANGED}\ state \end{aligned}$$

$Next \triangleq Receive \lor Send \lor Environment$

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

THEOREM $Spec \Rightarrow \Box TypeOK$

MODULE $PipesAndProcess1$

We compose two pipes and a process (| p |)

CONSTANT A, B
CONSTANT F
CONSTANT $Ap1(\_), Ap2(\_)$

VARIABLES $in1, state1,$
$p\_in, p\_out,$
$out2, state2$

$$\begin{aligned} Pipe1 &\triangleq \text{INSTANCE}\ Pipe1\ \text{WITH}\ Ap \leftarrow Ap1, in \leftarrow in1, state \leftarrow state1, out \leftarrow p\_in \\ Process &\triangleq \text{INSTANCE}\ Process1\ \text{WITH}\ in \leftarrow p\_in, out \leftarrow p\_out \\ Pipe2 &\triangleq \text{INSTANCE}\ Pipe1\ \text{WITH}\ A \leftarrow B, \\ &\qquad Ap \leftarrow Ap2, in \leftarrow p\_out, state \leftarrow state2, out \leftarrow out2 \end{aligned}$$

$Spec \triangleq Pipe1!Spec \land Process!Spec \land Pipe2!Spec$

MODULE *MonadsArePipes1*

CONSTANTS A, B
CONSTANT F
CONSTANT $Ap1(\_)$, $Ap2(\_)$

VARIABLES x, as, mb, y

The main refinement theorem

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

$Pipes(out2, p\_out) \triangleq$

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 $Bind$.

INSTANCE $PipesAndProcess1$ WITH $in1 \leftarrow \langle x \rangle$,
$state1 \leftarrow \langle x \rangle$,
$p\_in \leftarrow as$,
$state2 \leftarrow y$

THEOREM $MonadsArePipes1 \triangleq \exists\, out2, p\_out : Monad!Spec \Rightarrow Pipes(out2, p\_out)!Spec$

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.

```
  A tombstone value, which will be used later. Must not be a constructor
EMPTY ≜ CHOOSE x : ∀ A : x ∉ Constructors(A)

┌──────────────────── MODULE Process2 ────────────────────┐

CONSTANTS A, B
CONSTANT F

ASSUME F ∈ [A → M(B)]

VARIABLES in, out,
          control
```

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

```
Environment ≜ ∧ ReaderWriter(DOMAIN F, in, out)
              ∧ ∨ UNCHANGED out
                ∨ control′ = EMPTY

TypeOK ≜ ∧ in      ∈ Seq(A)
         ∧ out     ∈ Seq(B)
         ∧ control ∈ Constructors(B) ∪ {EMPTY}

Init ≜ ∧ in ∈ Seq(DOMAIN F)
       ∧ out = ⟨⟩
       ∧ control = EMPTY

Compute ≜ ∧ in ≠ ⟨⟩
          ∧ in′ = RemoveLast(in)
          ∧ LET mb ≜ F[Last(in)]
            IN   ∧ out′ = Deconstruct(B, mb) ∘ out
                 ∧ control′ = ConsOf(B, mb)

Next ≜ Compute ∨ Environment

Spec ≜ Init ∧ □[Next]_⟨in, out, control⟩

THEOREM Spec ⇒ □TypeOK
```

```
────────────────────────────── MODULE Pipe2 ──────────────────────────────
```

Note that in a composition chaing 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.

```
CONSTANT A

CONSTANT Ap(_)
ASSUME   Ap(A) ∈ [M(A) → Seq(A)]

VARIABLES in,
          control,
          collect,
          state,
          out

vars ≜ ⟨in, control, collect, state, out⟩

TypeOK ≜ ∧ control ∈ Constructors(A) ∪ {EMPTY}
         ∧ in      ∈ Seq(A)
         ∧ out     ∈ Seq(A)
         ∧ collect ∈ Seq(A)
         ∧ state   ∈ M(A)

Environment ≜ ∧ ReaderWriter(A, in, out)
              ∧ ∨ UNCHANGED in
                ∨ control′ ≠ EMPTY
              ∧ UNCHANGED collect

Init ≜ ∧ in ∈ Seq(M(A)) ∧ in ≠ ⟨⟩
       ∧ control = ⟨⟩
       ∧ out ∈ Seq(A)

Receive ≜ ∧ control ≠ EMPTY
          ∧ UNCHANGED control
          ∧ IF in ≠ ⟨⟩ THEN ∧ in′ = RemoveLast(in)
                            ∧ collect′ = AddFirst(collect, Last(in))
                       ELSE UNCHANGED in

Send ≜ ∧ LET cons ≜ control
             ma   ≜ cons[collect]
         IN   ∧ state′ = Compose(A)[state, ma]
              ∧ out′ = Ap(A)[state′]
       ∧ collect′ = ⟨⟩
       ∧ control′ = EMPTY
       ∧ UNCHANGED in
```

$Next \triangleq Receive \lor Send \lor Environment$

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

MODULE *PipesAndProcess2*

CONSTANT A, B
CONSTANT F
CONSTANT $Ap1(\_), Ap2(\_)$

VARIABLES $in1, control1, state1, collect1,$
$\quad p\_in, p\_out,$
$\quad control2, collect2, state2, out2$

$Pipe1 \triangleq$ INSTANCE $Pipe2$ WITH $Ap \leftarrow Ap1, in \leftarrow in1, control \leftarrow control1,$
$\quad collect \leftarrow collect1, state \leftarrow state1, out \leftarrow p\_in$
$Process \triangleq$ INSTANCE $Process2$ WITH $in \leftarrow p\_in, out \leftarrow p\_out, control \leftarrow control2$
$Pipe2 \triangleq$ INSTANCE $Pipe2$ WITH $A \leftarrow B,$
$\quad Ap \leftarrow Ap2, in \leftarrow p\_out, control \leftarrow control2,$
$\quad collect \leftarrow collect2, state \leftarrow state2, out \leftarrow out2$

$Spec \triangleq Pipe1!Spec \land Process!Spec \land Pipe2!Spec$

MODULE *MonadsArePipes2*

CONSTANTS A, B
CONSTANT F
CONSTANT $Ap1(\_), Ap2(\_)$

VARIABLES x, as, mb, y

The main refinement theorem

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

In addition to the variables we hide (intermediate ones and ones *Bind* doesn't model, we also quntify over *control*2 and *collect*2, as we only look at them at the moment of *Pipe*2 send (*i.e.*, when the monadic value is constructed).

$DuringSend(in) \triangleq in = \langle\rangle$

$Pipes(control2, collect2,$
$\quad control1, collect1, p\_out, out2) \triangleq$
$\quad$ INSTANCE $PipesAndProcess2$
$\quad\quad$ WITH $in1 \leftarrow \langle x \rangle,$
$\quad\quad\quad p\_in \leftarrow as,$

```
                          state1   ← x,
                          control2 ← IF DuringSend(p_out) THEN ConsOf(B, mb) ELSE control1,
                          collect2 ← IF DuringSend(p_out) THEN Deconstruct(B, mb) ELSE collect1,
                          state2   ← y

THEOREM MonadsArePipes2 ≜
    ∃ control2, collect2, control1, collect1, p_out, out2 :
        Monad!Spec ⇒ Pipes(control2, collect2, control1, collect1, p_out, out2)!Spec
```