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.

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 moands "are" pipes, the transformation must be compositional with respect to the syntax, *i.e.*, we must show how a specific monadic function F is trsansformed 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 moands 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 explictly state so.

Utility definitions

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

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

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

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

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

$$Reverse(seq) \triangleq [i \in 1 \,..\, Len(seq) \mapsto seq[Len(seq) - i + 1]]$$

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.)

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 constuctors 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 $Env \rightarrow A$.

```
ASSUME $\forall A : \forall cons \in Constructors(A) :$ DOMAIN $cons \subseteq Seq(A)$
              ($Seq(A)$ is the set of all finite sequences with elements in $A$ )
```

Our monadic type

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

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

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

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* :

```
CONSTANTS $Return(\_)$,
          $Compose(\_)$

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

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

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

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

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

MODULE *RunBind*

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

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

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*).

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, 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.

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

```
$TypeOK \triangleq$ $\land\ x \in M(A)$
                    $\land\ as \in Seq(A)$
                    $\land\ mb \in M(B)$
                    $\land\ y \in M(B)$
```

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

```
$Init \triangleq$ $\land\ \exists\, ma \in M(A) : as = Ap(A)[ma]$
                  $\land\ y = Unit(B)$

$Next \triangleq$ $\land\ as' \neq \langle\rangle$
                  $\land\ as' = Tail(as)$
                  $\land$ LET $a \triangleq Head(as)$
                      IN  $\land\ mb' = F[a]$
                          $\land\ y' = Compose(B)[y,\ mb']$
                  $\land$ UNCHANGED $x$

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

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 preisely. Expressing the monad denotationally allows us to specify the monad laws.

```
$MonadLaws(bind) \triangleq$
  $\land\ bind \in [M(A) \times [A \rightarrow M(B)] \rightarrow M(B)]$
  $\land\ \forall\, a \in A,\ f \in [A \rightarrow M(B)] : bind[Return(A)[a],\ f] = f[a]$
  $\land\ A = B \Rightarrow \forall\, ma \in M(A) : bind[ma,\ Return(A)] = ma$
  $\land$ LET $kl[f \in [A \rightarrow M(A)],\ g \in [B \rightarrow M(B)]] \triangleq [a \in A \mapsto bind[f[a],\ g]]$
     IN  $\forall\, C : \forall\, f \in [A \rightarrow M(A)],\ g \in [B \rightarrow M(B)],\ h \in [C \rightarrow M(C)] :$
                 $kl[kl[f,\ g],\ h] = kl[f,\ kl[g,\ h]]$
```

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

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

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).

$\mathit{ReaderWriter}(A, \mathit{in}, \mathit{out}) \triangleq$ Specifies an "environment" that can write to *in* and read from *out* .

```
    ∨ $\land\ \exists x \in \mathit{Seq}(A) : \mathit{in}' = x \circ \mathit{in}$
      $\land$ UNCHANGED $\mathit{out}$
    ∨ $\land\ \exists n \in 0\,..\,\mathit{Len}(\mathit{out}) : \mathit{out}' = \mathit{SubSeq}(\mathit{out}, 1, n)$
      $\land$ UNCHANGED $\mathit{in}$
```

MODULE *Process*

A process is given as a transformation of a monadic function F *and* the folliwing *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.

CONSTANTS A, B

CONSTANT F

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).

CONSTANT $Ap(\_)$

ASSUME $Ap(A) \in [M(A) \rightarrow \mathit{Seq}(A)]$

VARIABLES $\mathit{in}, \mathit{out}$ LIFO channels

$\mathit{TypeOK} \triangleq \mathit{in} \in \mathit{Seq}(A) \land \mathit{out} \in \mathit{Seq}(B)$

$\mathit{Environment} \triangleq \mathit{ReaderWriter}(A, \mathit{in}, \mathit{out})$

```
Init ≜ $\land\ \mathit{in} \in \mathit{Seq}(A)$
       $\land\ \mathit{out} \in \mathit{Seq}(B)$

Compute ≜ $\land\ \mathit{in} \neq \langle\rangle$
          $\land\ \mathit{in}' = \mathit{RemoveLast}(\mathit{in})$
          $\land$ LET $mb \triangleq F[\mathit{Last}(\mathit{in})]$   The effect also taked place here
               IN $\mathit{out}' = Ap(B)[mb] \circ \mathit{out}$
```

$\mathit{Next} \triangleq \mathit{Compute} \lor \mathit{Environment}$

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

THEOREM $\mathit{Spec} \Rightarrow \Box\,\mathit{TypeOK}$

MODULE *Pipe*

An example of composing two processes with a pipe – we simply let the first output into the other's input.

CONSTANTS $A,\ B,\ C$

CONSTANT $F,\ G,\ Ap(\_)$

VARIABLES $in,\ shared,\ out$

$Process1 \triangleq$ INSTANCE $Process$ WITH $out \leftarrow shared$
$Process2 \triangleq$ INSTANCE $Process$ WITH $in \leftarrow shared,\ F \leftarrow G,\ A \leftarrow B,\ B \leftarrow C$

$Spec \triangleq Process1!Spec \land Process2!Spec$

MODULE *MonadsArePipes*

CONSTANTS $A,\ B$
CONSTANT F
CONSTANT $Ap(\_)$

VARIABLES $x,\ as,\ mb,\ y$

<u>The main refinement theorem</u>

$Monad \triangleq$ INSTANCE $RunBind$

$Process \triangleq$ INSTANCE $Process$ WITH $in \leftarrow Reverse(as),$
$\qquad out \leftarrow Ap(B)[y]$ Probably need to reverse something here – details.

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 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?

THEOREM $CompositionOfConstructorArguments \triangleq$
$\quad \forall\, T : \forall\, xs,\ ys \in Seq(T),\ cons \in Constructors(T) :$
$\qquad Compose(T)[cons[xs],\ cons[ys]] = cons[xs \circ ys]$
