Monad code generation and extension

Anton Antonov
MathematicaForPrediction at WordPress
MathematicaForPrediction at GitHub
MathematicaVsR project at GitHub
June 2017

Introduction

This document (blog post) aims to introduce monadic programming in Mathematica / Wolfram Language (WL) in a concise and code-direct manner. The monad codes discussed are derived from ”first principles” of Mathematica / WL.

The usefulness of the monadic programming approach comes from several angles:

1) easy to construct, read, and modify pipeline of commands,
2) easy to program data-polymorphic behaviour,
3) easy to program context utilization.

Speaking informally,

> Monad programming provides an interface to sequentially structured computations that allows data polymorphic and contextual behavior to handled by the constructed sequences of operators.

The main theoretic background for this document (blog post) is the Wikipedia article on Monadic programming, [1]. The code in is document uses the definition given there and in this document that definition is referred to as the ***“Haskell definition”***.

The Haskell definition in Mathematica terms

Here are operators for a monad associated with a certain symbol `M`:

1. monad *unit* function ("return" in Haskell notation) is `Unit[x_] := M[x]`;

2. monad *bind* function (">>=" in Haskell notation) is a rule like `Bind[M[x_], f_] := f[x]` with `MatchQ[f[x],M[_]]` giving `True`.

Note that the code definitions in 2 according to the general definition:

- the function `Bind` unwraps the content of `M[_]` and gives it to the function `f`;
- the functions f_i are responsible to return results wrapped with the monad symbol `M`.

Here is an illustration formula showing a ***monad pipeline***:

$$M[data] \xrightarrow[Bind[M[\_],f\_]]{} f_1 \xrightarrow[Bind[M[\_],f\_]]{} f_2 \xrightarrow[Bind[M[\_],f\_]]{} f_3 \xrightarrow[Bind[M[\_],f\_]]{} \ldots \xrightarrow[Bind[M[\_],f\_]]{} f_k \tag{1}$$

From the definition and formula it should be clear that if for the result `f[x]` of `Bind` the test `MatchQ[f[x],_M]` is `True` then the result is ready to be fed to the next binding operation in monad's pipeline. Also, it is easy to program the pipeline functionality with `Fold`:

In[1]:=
```
Fold[Bind, M[x], {f1, f2, f3}]
```

Out[1]=
```
Bind[Bind[Bind[M[x], f1], f2], f3]
```

Expected monadic programming features

Looking at formula (1) -- and having certain programming experiences -- we can expect the following monadic programming features.

- Computations that can be expressed with monad pipelines are easy to construct and read.
- By programming the binding function we can tuck-in certain particular monad behaviours -- this the so called “programmable semicolon” feature of monadic programming.
- Monad pipelines can be constructed with `Fold`, but with suitable definitions of infix operators like `DoubleLongRightArrow` (⟹) we can produce code that resembles the pipeline in formula (1).
- A monad pipeline can have a polymorphic behaviour by overloading the signatures of f_i (and if we have to, `Bind`.)

These points are clarified below. For a more complete discussion see [1].

First monadic programming examples

It is fairly easy to program the basic monads Maybe and State discussed in [1].

Maybe monad

The goal of the Maybe monad is to provide easy exception handling in a sequence of chained computational steps. If one of the computation steps fails then the whole pipeline returns a designated failure symbol, say `None`; otherwise the result after the last step is wrapped in another designated symbol, say `Maybe`.

Here is the special version of the generic pipeline formula (1) for the Maybe monad:

$$\texttt{Maybe[data]} \xrightarrow[\texttt{Bind[m\_,f\_]}]{} \cdots \xrightarrow[\texttt{Bind[m\_,f\_]}]{} \left(\begin{cases} \texttt{None} & \texttt{m} \equiv \texttt{None} \\ \texttt{f}_i\texttt{[x]} & \texttt{m } \textit{is} \texttt{ Maybe[x\_]} \end{cases} \right) \xrightarrow[\texttt{Bind[m\_,f\_]}]{} \cdots \tag{2}$$

Here is the minimal code to get a functional Maybe monad (for a more detailed exposition of code and explanations see [8]):

In[2]:=
```
MaybeUnitQ[x_] := MatchQ[x, None] || MatchQ[x, Maybe[___]];
```

In[3]:=
```
MaybeUnit[None] := None;
MaybeUnit[x_] := Maybe[x];
```

In[5]:=
```
MaybeBind[None, f_] := None;
MaybeBind[Maybe[x_], f_] := Block[{res = f[x]}, If[FreeQ[res, None], res, None]];
```

In[7]:=
```
MaybeEcho[x_] := Maybe@Echo[x];
MaybeEchoFunction[f___][x_] := Maybe@EchoFunction[f][x];
```

In[9]:=
```
MaybeOption[f_][xs_] := Block[{res = f[xs]}, If[FreeQ[res, None], res, Maybe@xs]];
```

In order to make the pipeline form of the code we are going to write below let us give definitions to suitable infix operator (like "⟹") to use `MaybeBind`:

In[10]:=
```
DoubleLongRightArrow[x_?MaybeUnitQ, f_] := MaybeBind[x, f];
DoubleLongRightArrow[x_, y_, z__] := DoubleLongRightArrow[DoubleLongRightArrow[x, y], z];
```

Here is an example of a Maybe monad pipeline using the definitions so far:

In[12]:=
```
data = {0.61, 0.48, 0.92, 0.90, 0.32, 0.11};
```

Out[14]=
```
MaybeUnit[data]⟹                          (* lift data into the monad *)
 (Maybe@Join[#,RandomInteger[8,3]]&)⟹     (* add more values *)
 MaybeEcho⟹                               (* display current value *)
 (Maybe@Map[If[#<0.4,None,#]&,#]&)        (* map values that are too small to None *)
```

» `{0.61, 0.48, 0.92, 0.9, 0.32, 0.11, 2, 8, 6}`

Out[15]= None

The result is `None` because:

1. the data has a number that is too small, and

2. the definition of `MaybeBind` stops the pipeline aggressively using a `FreeQ[_,None]` test.

Monad laws verification

Let us convince ourselves that the current definition of `MaybeBind` gives a monad. The monad laws definitions are taken from https://wiki.haskell.org/Monad_laws .

Laws

In the monad laws given below "⟹" is for monad's binding operation and ($x \mapsto$ expr) is for a function in anonymous form.

Here is a table with the laws:

Out[19]=

⌗	name	LHS		RHS
1	Left identity	`unit a ⟹ f`	≡	`f a`
2	Right identity	`m ⟹ unit`	≡	`m`
3	Associativity	`(m ⟹ f) ⟹ g`	≡	`m ⟹ (x ↦ f x ⟹ g)`

Verification

The verification is straightforward to program and shows that the implemented Maybe monad adheres to the monad laws.

Out[23]=

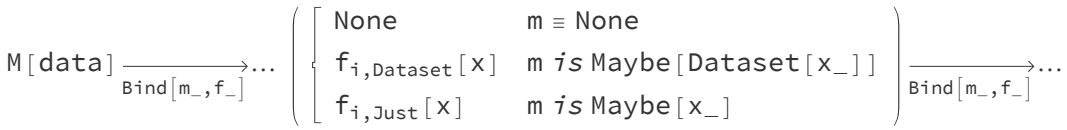
⌗	name	Input	Output
1	Left identity	`MaybeUnit[a]⟹f`	`f[a]`
2	Right identity	`Maybe[a]⟹MaybeUnit`	`Maybe[a]`
3	Associativity LHS	`(Maybe[a]⟹(Maybe[f1[⌗1]] &))⟹(Maybe[f2[⌗1]] &)`	`Maybe[f2[f1[a]]]`
4	Associativity RHS	`Maybe[a]⟹Function[{x}, Maybe[f1[x]]⟹(Maybe[f2[⌗1]] &)]`	`Maybe[f2[f1[a]]]`

Extensions with polymorphic behavior

We can see from formulas (1) and (2) that the monad codes can be easily extended through overloading the pipeline functions.

For example the extension of the Maybe monad to handle of `Dataset` objects is fairly easy and straightforward.

Here is the formula of the Maybe monad pipeline extended with `Dataset` objects:



Here is an example of a polymorphic function definition for the Maybe monad:

In[25]:=
```
MaybeFilter[filterFunc_][xs_] := Maybe@Select[xs, filterFunc[#] &];
```

In[26]:=
```
MaybeFilter[critFunc_][xs_Dataset] := Maybe@xs[Select[critFunc]];
```

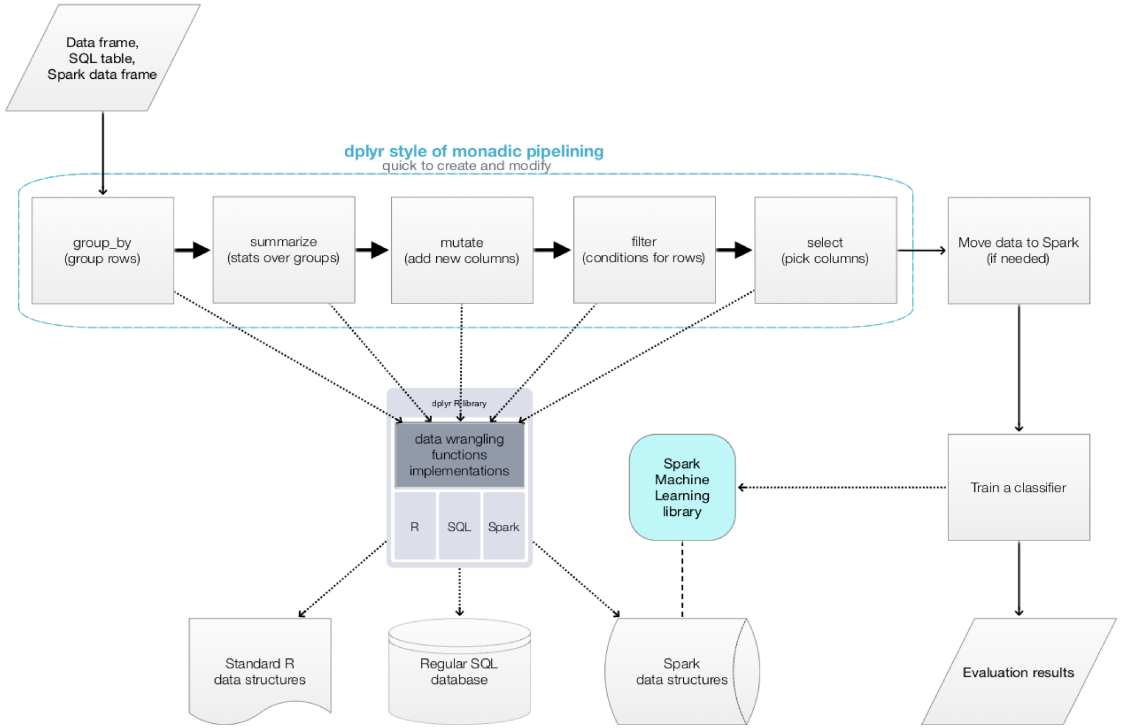
See [8] for more detailed examples of polymorphism in monadic programming with Mathematica / WL.

Polymorphic monads in R's dplyr

The R package dplyr has implementations centered around monadic polymorphic behavior. The pipelines using dplyr can work on R data frames, SQL tables, and Spark data frames without changes.

Here is a diagram of typical work-flow with dplyr:

In[27]:=



The diagram shows how a pipeline made with dplyr can be re-run (or reused) for different data, placed in different data structures.

Monad code generation

We can see monad code definitions like the ones for Maybe as some sort of initial templates for monads that can be extended in specific ways depending on their applications. Mathematica / WL can easily provide code generation for such templates.

In this section are given examples with package that generate codes. The case study sections have examples of packages that utilize generated

monad codes.

Maybe monad code generation

The package [3] provides Maybe code generator for a given prefix of the generated functions. (The monad code generation is discussed below in greater detail.)

Here is an example:

In[52]:=
```
Import["https://raw.githubusercontent.com/antononcube/MathematicaForPrediction/master/MonadicProgramming/
    MaybeMonadCodeGenerator.m"]
```

In[29]:=
```
GenerateMaybeMonadCode["AnotherMaybe"]
```

Out[31]=
```
AnotherMaybeUnit[data]⟹                          (* lift data into the monad *)
 (AnotherMaybe@Join[#,RandomInteger[8,3]]&)⟹     (* add more values *)
AnotherMaybeEcho⟹                                (* display current value *)
 (AnotherMaybe@Map[If[#<0.4,None,#]&,#]&)        (* map values that are too small to None *)
```

» {0.61, 0.48, 0.92, 0.9, 0.32, 0.11, 1, 5, 8}

» AnotherMaybeBind: Failure when applying: Function[AnotherMaybe[Map[Function[If[Less[Slot[1], 0.4], None, Slot[1]]], Slot[1]]]]

Out[32]= None

We see we get the same result as above.

State monad code generation

The State monad is also basic and its programming in Mathematica / WL is not that difficult. (See [4].)

Here is the special version of the generic pipeline formula (1) for the State monad:

$$\texttt{State[data, context]} \xrightarrow[\texttt{Bind[m\_,f\_]}]{} \cdots \left(\begin{cases} \texttt{f}_i\texttt{[None]} & \texttt{m} \equiv \texttt{None} \\ \texttt{f}_i\texttt{[x\_, c\_Association]} & \texttt{m } \textit{is} \texttt{ State[x\_, c\_Association]} \\ \texttt{None} & \text{otherwise} \end{cases} \right) \xrightarrow[\texttt{Bind[m\_,f\_]}]{} \cdots$$

Note since the State monad pipeline caries both a value and a state, it is a good idea to have functions that manipulated in a separately. For example, we can have functions for context modification and context retrieval. (These are done in [4].)

Let us demonstrate with the State monad with a code generation example.

In[34]:=
```
Import["https://raw.githubusercontent.com/antononcube/MathematicaForPrediction/master/MonadicProgramming/
    StateMonadCodeGenerator.m"]
```

In[35]:=
```
GenerateStateMonadCode["StMon"]
```

The following pipeline code starts with a random matrix and then replaces values in the pipeline value according to a threshold parameter kept in the context. Several times context deposit and retrieval functions are invoked.

In[36]:=
```
SeedRandom[34]
StMonUnit[RandomReal[{0, 1}, {3, 2}], <|"mark" → "TooSmall", "threshold" → 0.5|>] ⟹
  StMonEchoValue ⟹
  StMonEchoContext ⟹
  StMonAddToContext["data"] ⟹
  StMonEchoContext ⟹
  (StMon[#1 /. (x_ /; x < #2["threshold"] :↦ #2["mark"]), #2] &) ⟹
  StMonEchoValue ⟹
  StMonRetrieveFromContext["data"] ⟹
  StMonEchoValue ⟹
  StMonRetrieveFromContext["mark"] ⟹
  StMonEchoValue;
```

» value: {{0.789884, 0.831468}, {0.421298, 0.50537}, {0.0375957, 0.289442}}

» context: ⟨|mark → TooSmall, threshold → 0.5|⟩

» context: ⟨|mark → TooSmall, threshold → 0.5, data → {{0.789884, 0.831468}, {0.421298, 0.50537}, {0.0375957, 0.289442}}|⟩

» value: {{0.789884, 0.831468}, {TooSmall, 0.50537}, {TooSmall, TooSmall}}

» value: {{0.789884, 0.831468}, {0.421298, 0.50537}, {0.0375957, 0.289442}}

» value: TooSmall

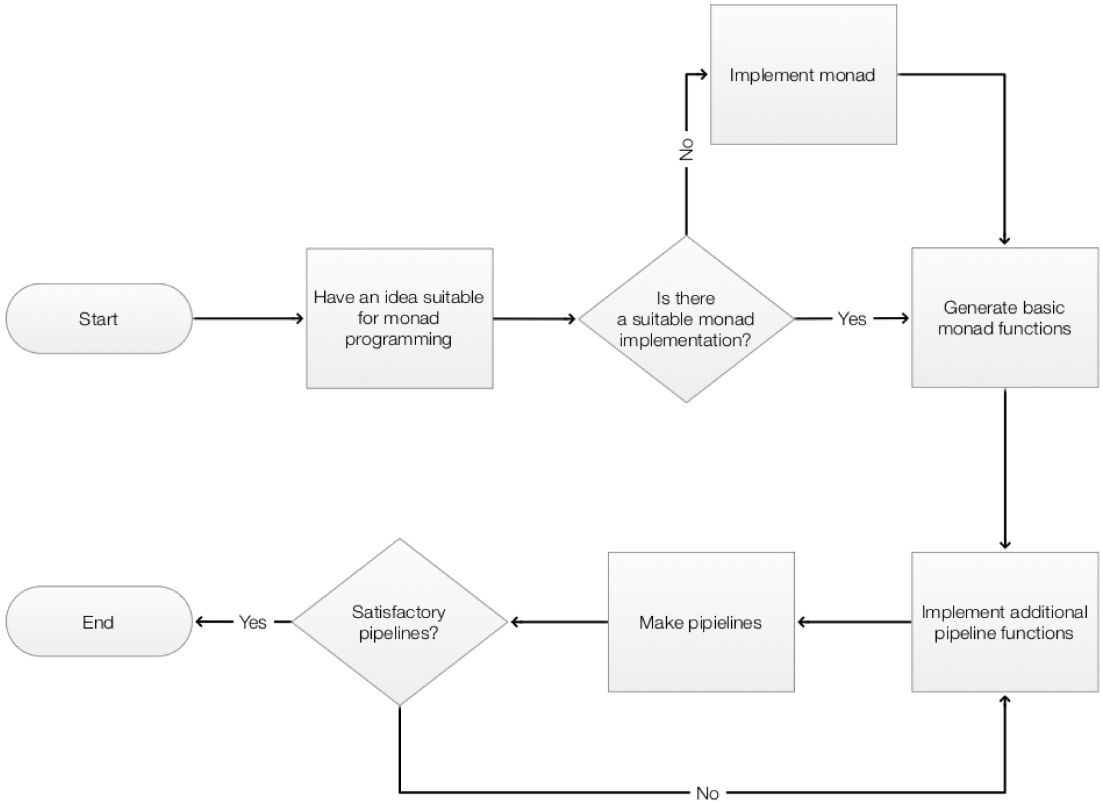
General work-flow

With the abilities to generate and utilize monad codes it is natural to consider the following work flow. (Also shown in the diagram below.)

1. Come up with an idea that can be expressed with monadic programming.

2. Look for suitable monad implementation.
3. If there is no such implementation, make one (or two, or five.)
4. Having a suitable monad implementation, generate the monad code.
5. Implement additional pipeline functions addressing envisioned use cases.
6. Start making pipelines for the problem domain of interest.
7. Are the pipelines are satisfactory? If not go to **5.**

In[38]:=



The template nature of the general monads can be exemplified with the group of functions in the package StateMonadCodeGenerator.m, [4].

They are in five groups:

1. base monad functions (unit testing, binding),
2. display of the value and context,
3. context manipulation (deposit, retrieval, modification),
4. flow governing (optional new value, conditional function application),
5. other convenience functions.

We can say that all monad implementations will have their own versions of these groups of functions. The more specialized monads will have functions specific to their intended use. Such special monads are discussed in the next case study sections.

Contextual classification *(case study)*

In this section we show a State based and extending monad made to be used in machine learning classification work-flows.

This loads the package [5]:

In[41]:=
```
Import["https://raw.githubusercontent.com/antononcube/MathematicaForPrediction/master/MonadicProgramming/
   MonadicContextualClassification.m"]
```

This gets some test data (the Titanic dataset):

In[43]:=
```
dataName = "Titanic";
ds = Dataset[Flatten@*List @@@ ExampleData[{"MachineLearning", dataName}, "Data"]];
varNames = Flatten[List @@ ExampleData[{"MachineLearning", dataName}, "VariableDescriptions"]];
ds = ds[All, AssociationThread[varNames → #] &];
```

This monadic pipeline goes through several stages: data summary, classifier training, evaluations, acceptance test, and if results are rejected a bnew classifier is made with a different algorithm using the same data. The context keeps track of the data and its splitting. That allows the conditional classifier switch to be concisely specified.

In[47]:=
```
res =
  ClConUnit[ds] ⟹
   ClConSplitData[0.75] ⟹
   ClConEchoFunctionValue["summary:", RecordsSummary /@ # &] ⟹
   ClConEchoFunctionValue["xtabs:", MatrixForm[CrossTensorate[Count == 3 + 4, #]] & /@ # &] ⟹
   ClConEchoFunctionValue["classifier:", "NearestNeighbors" &] ⟹
   ClConMakeClassifier["NearestNeighbors"] ⟹
   ClConEchoFunctionContext["training time:", ClassifierInformation[#["classifier"], "TrainingTime"] &] ⟹
   ClConClassifierMeasurements[{"Accuracy", "Precision", "Recall"}] ⟹
   ClConEchoValue ⟹
   (If[#1["Accuracy"] > 0.8, Echo["Good accuracy!", "Success:"];
      ClConFail, ClConUnit[#1, #2]] &) ⟹
   ClConEchoFunctionValue["classifier:", "RandomForest" &] ⟹
   ClConMakeClassifier["RandomForest"] ⟹
   ClConEchoFunctionContext["training time:", ClassifierInformation[#["classifier"], "TrainingTime"] &] ⟹
   ClConClassifierMeasurements[{"Accuracy", "Precision", "Recall"}] ⟹
   ClConEchoValue;
```

» summary: ⟨|

trainData → { 1 passenger class: 3rd 534, 1st 246, 2nd 201 , 2 passenger age: Missing[] 197, 21. 34, 22. 33, 24. 32, 18. 29, 30. 27, (Other) 629 , 3 passenger sex: male 630, female 351 , 4 passenger survival: died 601, survived 380 },

testData → { 1 passenger class: 3rd 175, 1st 77, 2nd 76 , 2 passenger age: Missing[] 66, 24. 15, 30. 13, 31. 13, 18. 10, 19. 10, (Other) 201 , 3 passenger sex: male 213, female 115 , 4 passenger survival: died 208, survived 120 }

|⟩

» xtabs: ⟨|

trainData →

	died	survived
female	99	252
male	502	128

, testData →

	died	survived
female	28	87
male	180	33

|⟩

» classifier: NearestNeighbors

» training time: 0.113982 s

» value:

⟨|Accuracy → 0.778626, Precision → ⟨|died → 0.761658, survived → 0.826087|⟩, Recall → ⟨|died → 0.924528, survived → 0.553398|⟩|⟩

» classifier: RandomForest

» training time: 0.141532 s

» value:

⟨|Accuracy → 0.835878, Precision → ⟨|died → 0.825843, survived → 0.857143|⟩, Recall → ⟨|died → 0.924528, survived → 0.699029|⟩|⟩

Tracing monad pipelines *(case study)*

The monadic implementations in the package [6] allow tracking of the pipeline execution of functions with other monads.

The primary reason for developing the package was the desire to have the ability to print a tabulated trace of code and comments using the usual monad pipeline notation. (I.e. without conversion to strings etc.)

This loads the package [6]:

In[49]:=
```
Import[
 "https://raw.githubusercontent.com/antononcube/MathematicaForPrediction/master/MonadicProgramming/MonadicTracing.
   m"]
```

This generates a Maybe monad to be used in the example (for the prefix “Perhaps”):

In[58]:=
```
GenerateMaybeMonadCode["Perhaps"]
GenerateMaybeMonadSpecialCode["Perhaps"]
```

In this example we can see that pipeline functions of the Perhaps monad are interleaved with comment strings.
Producing the grid of functions and comments happens “naturally” with the monad function `TraceMonadEchoGrid`.

Note that :

1. the tracing is initiated by just using `TraceMonadUnit`;
2. pipeline functions (actual code) and comments are interleaved;
3. putting a comment string after a pipeline function is optional.

In[54]:=
```
data = RandomInteger[10, 15];
```

In[60]:=
```
TraceMonadUnit[PerhapsUnit[data]]⟹"lift to monad"⟹
  TraceMonadEchoContext⟹
  PerhapsFilter[# > 3 &]⟹"filter current value"⟹
  PerhapsEcho⟹"display current value"⟹
  PerhapsWhen[#[[3]] > 3 &, PerhapsEchoFunction[Style[#, Red] &]]⟹
  (Perhaps[# / 4] &)⟹
  PerhapsEcho⟹"display current value again"⟹
  TraceMonadEchoGrid[Grid[#, Alignment → Left] &];
```

» context: ⟨|data → PerhapsUnit[data], binder → DoubleLongRightArrow, commands → {}, comments → {lift to monad}|⟩

» {10, 5, 7, 4, 6, 4, 8, 7, 9, 7, 9}

» {10, 5, 7, 4, 6, 4, 8, 7, 9, 7, 9}

» $\left\{\frac{5}{2}, \frac{5}{4}, \frac{7}{4}, 1, \frac{3}{2}, 1, 2, \frac{7}{4}, \frac{9}{4}, \frac{7}{4}, \frac{9}{4}\right\}$

```
»  PerhapsUnit[data] ⟹                                              lift to monad
     PerhapsFilter[#1 > 3 &] ⟹                                      filter current value
     PerhapsEcho ⟹                                                  display current value
     PerhapsWhen[#1[[3]] > 3 &, PerhapsEchoFunction[Style[#1, Red] &]] ⟹
     Perhaps[#1/4 &] ⟹
     PerhapsEcho                                                    display current value again
```

Summary

References

[1] Wikipedia entry: Monad (functional programming), URL: https://en.wikipedia.org/wiki/Monad_(functional_programming) .

[2] Anton Antonov, “Implementation of Object-Oriented Programming Design Patterns in Mathematica”, (2016) MathematicaForPrediction at GitHub, https://github.com/antononcube/MathematicaForPrediction, folder Documentation.

[3] Anton Antonov, Maybe monad code generator Mathematica package, (2017), MathematicaForPrediction at GitHub.
URL: https://github.com/antononcube/MathematicaForPrediction/blob/master/MonadicProgramming/MaybeMonadCodeGenerator.m .

[4] Anton Antonov, State monad code generator Mathematica package, (2017), MathematicaForPrediction at GitHub.
URL: https://github.com/antononcube/MathematicaForPrediction/blob/master/MonadicProgramming/StateMonadCodeGenerator.m .

[5] Anton Antonov, Monadic contextual classification Mathematica package, (2017), MathematicaForPrediction at GitHub.
URL: https://github.com/antononcube/MathematicaForPrediction/blob/master/MonadicProgramming/MonadicContextualClassification.m .

[6] Anton Antonov, Monadic tracing Mathematica package, (2017), MathematicaForPrediction at GitHub.
URL: https://github.com/antononcube/MathematicaForPrediction/blob/master/MonadicProgramming/MonadicTracing.m .

[7] Anton Antonov, MathematicaForPrediction utilities, (2014), MathematicaForPrediction at GitHub.
URL: https://github.com/antononcube/MathematicaForPrediction/blob/master/MathematicaForPredictionUtilities.m .

[8] Anton Antonov, Simple monadic programming, (2017), MathematicaForPrediction at GitHub.

(*Preliminary version, 40% done.)*
URL: https://github.com/antononcube/MathematicaForPrediction/blob/master/Documentation/Simple-monadic-programming.pdf .