Simple monadic programming

Implementations in Mathematica / Wolfram Language

Preliminary, unfinished version, 0.4

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

Introduction

The goal of this document is to provide monad implementations in Mathematica / Wolfram Language (WL) that are both simple to do and useful.

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.

Theoretical and computational background

Monad definition used

My main source for writing this document was 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"***.

Here is the "informal" definition in [1]:

A monad is created by defining a type constructor *M* and two operations, ***bind*** and ***return*** (where ***return*** is often also called ***unit***):

- The unary ***return*** operation takes a value from a plain type (a) and puts it into a container using the constructor, creating a monadic value (with type *M a*).
- The binary ***bind*** operation ">>=" takes as its arguments a monadic value with type *M a* and a function (*a* → *M b*) that can transform the value.
  - The bind operator unwraps the plain value with type a embedded in its input monadic value with type *M a*, and feeds it to the function.
  - The function then creates a new monadic value, with type *M b*, that can be fed to the next bind operators composed in the pipeline.

Note: There are at least two programming monads definitions. The Haskell definition uses "return" and "bind". Another definition -- used in Scala -- is based the operators "fmap" and "join" ("flatmap", "map").

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***:

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

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`:

```
Fold[Bind, M[x], {f1, f2, f3}]
Bind[Bind[Bind[M[x], f1], f2], f3]
```

Premonitions (from the definitions)

Looking at formula (1) we can expect the following points.

- 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` (⟹) or `NonCommutativeMultiply` (**) 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.

Programming the Maybe monad

The Maybe monad is a simple example that shows one of the fundamental aspects of monad programming application. (See the description of the monad Maybe in [1].)

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 the whole pipeline returns a designed 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:

$$\text{Maybe}[\text{data}] \xrightarrow[\text{Bind}[m\_,f\_]]{} \cdots \xrightarrow[\text{Bind}[m\_,f\_]]{} \begin{pmatrix} \text{None} & m \equiv \text{None} \\ f_i[x] & m \ \textit{is}\ \text{Maybe}[x\_] \end{pmatrix} \xrightarrow[\text{Bind}[m\_,f\_]]{} \cdots \tag{2}$$

First definitions

Since we consider the Maybe monad to work on a certain new type, let us define a test for that new type:

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

Here is a definition for the binding operation that fits the formula (2):

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

Note that `MaybeBind` bails-out more aggressively than prescribed in (2) -- the pipeline goes to `None` if the current value *has* `None`, not just when the current value *is* `None`.

Let us define for completeness the function `MaybeUnit` which takes an ordinary value and "lifts" it into the monad:

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

Here are a few definitions of monad pipeline functions:

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

```
MaybeEcho = Maybe@*Echo;
MaybeEchoFunction[f_][x_] := Maybe@EchoFunction[f][x];
```

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`:

```
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:

```
data = {0.61, 0.48, 0.92, 0.90, 0.32, 0.11};
Maybe[data]⟹                          (* lift data into the monad *)
 MaybeFilter[#>0.3&]⟹                 (* filter current value *)
 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}

None
```

As it was mentioned in the "Premonitions" sub-section the pipeline code of the above example is equivalent to using `Fold`:

```
Fold[
 MaybeBind,
 Maybe[data],                          (* lift data into the monad *)
 {MaybeFilter[#>0.3&],                 (* filter current value *)
  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}

None

Pipeline flow control definitions

Next it is a good idea to provide some pipeline functions that can control the flow of the data.

Consider the function `MaybeOption[f_]` that would attempts to continue the flow with the result of `f`:

- if `f` produces failure then `MaybeOption` passes current pipeline value;
- if `f` produces a successful result then `MaybeOption` passes that result.

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

Similar simple functions are `MaybeIfElse` and `MaybeWhen`. The function `MaybeIfElse` has three arguments: a test function applied to the current value, a yes-function if the test gives `True`, and no-function if the test gives `False`.

```
MaybeIfElse[testFunc_, fYes_, fNo_][xs_] := Block[{testRes = testFunc[xs]}, If[TrueQ[testRes], fYes[xs], fNo[xs]]];
```

The function `MaybeWhen` is based on `MaybeIfElse` with a no-function that is the monad identity. (See the monad laws below.)

```
MaybeWhen[testFunc_, f_][xs_] := MaybeIfElse[testFunc, f, Maybe][xs];
```

Consider the following example:

```
data={0.61,0.48,0.92,0.90,0.32,0.11};
MaybeUnit[data]⟹                                  (* lift data to the monad *)
 MaybeFilter[#>0.3&]⟹                             (* filter-out numbers ≤ 0.3 *)
 MaybeEcho⟹                                       (* display current value *)
 MaybeOption[(Maybe@Map[If[#<0.4,None,#]&,#]&)]⟹  (* attempt mapping values that are < 0.4 *)
 MaybeEcho⟹                                       (* display current value *)
 MaybeWhen[Total[#]>2&,Maybe@Style[#,Red]&]       (* if the total is > 2 color in red *)
```

```
» {0.61, 0.48, 0.92, 0.9, 0.32}
» {0.61, 0.48, 0.92, 0.9, 0.32}
Maybe[{0.61, 0.48, 0.92, 0.9, 0.32}]
```

```
ToExpression[code]
```

```
» {0.61, 0.48, 0.92, 0.9, 0.32}
» {0.61, 0.48, 0.92, 0.9, 0.32}
Maybe[{0.61, 0.48, 0.92, 0.9, 0.32}]
```

We can see in the example that `MaybeOption` avoided the failure, and since the total is greater than 2 the pipeline value is colored in red.

There are other control flow functions that can be defined. But that might be more convenience not so much because they are needed. We can always use standard pipeline pure function application to control the flow. Let us demonstrate this with an example.

Consider a control flow function for that applies each of a list of the functions to the current pipeline value and selects a result according to certain scoring function. We can define a function like:

```
MaybeChoice[scoreFunc_, fs_List][xs_] :=
  Block[{res = Through[fs[xs]]}, Maybe@Last[SortBy[res, scoreFunc]]];
```

Consider these data and functions

```
data = {1, 9, 5, 1, 8};
fs = {Sqrt@*Total, Total@*Sqrt, Times @@ (Most[#] / Rest[#]) &, Times @@ (Rest[#] / Most[#]) &};
```

Then make this kind of pipelines:

```
MaybeUnit[data]⟹MaybeEchoFunction[Through[fs[#]] &]⟹MaybeChoice[N, fs]
```

```
» {2 √6 , 5 + 2 √2 + √5 , 1/8, 8}
Maybe[5 + 2 √2 + √5 ]
```

Or directly use a pure function (with the same effect) instead of `MaybeChoice`:

```
MaybeUnit[data] ⟹ MaybeEchoFunction[Through[fs[#]] &] ⟹ (Maybe@Last@SortBy[Through[fs[#]], N] &)
```

» $\left\{2\sqrt{6}, 5+2\sqrt{2}+\sqrt{5}, \frac{1}{8}, 8\right\}$

$\text{Maybe}\left[5+2\sqrt{2}+\sqrt{5}\right]$

It is a monad indeed

Let us convince ourselves that the current definition of `MaybeBind` gives a monad.

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:

#	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.

#	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]]]`

Every symbol has a monad with List and Apply

The monad laws are satisfied for every symbol in Mathematica with `List` being the unit and `Apply` being the binding operation.

If we program `Unit` and `DoubleLongRightArrow` like this:

```
Unit = List;
DoubleLongRightArrow[x_, f_] := f @@ x;
```

we can see that monad laws hold:

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

Special functions

Let us define several special functions that are useful for certain applications we have in mind (but not of general interest in Maybe monads.)

```
MaybeRandomChoice[n_][xs_] :=
  Maybe@Block[{res = RandomChoice[xs, n]}, If[TrueQ[Head[res] === RandomChoice], None, res]];

MaybeMapToFail[critFunc_][xs_] := MaybeMapToFail[critFunc, 1][xs];
MaybeMapToFail[critFunc_, lvl_][xs_] :=
  If[AtomQ[xs], If[critFunc[xs], None, xs], Maybe@Map[If[critFunc[#], None, #, #] &, xs, lvl]];

MaybeRandomReal[xs_] := Block[{res = RandomReal[Sequence @@ xs]}, If[NumberQ[res] || ListQ[res], Maybe@res, None]];
```

Here is an example using these functions.

```
MaybeUnit[{{-10, 100}, {30, 4}}] ⟹
 MaybeRandomReal ⟹
 MaybeEchoFunction[Grid[Quartiles /@ Transpose[#]] &] ⟹
 MaybeMapToFail[# < -1 &, {-1}]
```

```
17.8388  46.4547  74.2239
13.4514  47.6028  88.0834
5.55279  57.3342  79.1495
12.0126  34.2422  65.2693
```

```
None
```

The MaybeDivide special function

Since in [1] special attention is given to using the Maybe monad to handle division by 0, here we are going to discuss implementation of such a function.

Here we define the `MaybeDivide` function to divide the argument with the current pipeline value:

```
MaybeDivide[x_?MaybeUnitQ, y_?MaybeUnitQ] :=
  Block[{yres = MaybeBind[y, MaybeMapToFail[# == 0 &]]},
   If[! FreeQ[yres, None], None, Maybe[x[[1]] / y[[1]]]]
  ];
MaybeDivide[y_][xs_] := If[FreeQ[xs, None], MaybeDivide[y, Maybe[xs]], None];
```

This pipeline demonstrates a division success:

```
MaybeUnit[{2, 1, 2}] ⟹
 MaybeEcho⟹
 MaybeOption[MaybeDivide[Maybe[{1, 3, 9}]]]
```

» `{2, 1, 2}`

$$\text{Maybe}\left[\left\{\frac{1}{2}, 3, \frac{9}{2}\right\}\right]$$

This pipeline demonstrates a division failure -- the final result is unchanged because of `MaybeOption`.

```
MaybeUnit[{0, 2, 1}] ⟹
 MaybeEcho⟹
 MaybeOption[MaybeDivide[Maybe[{0, 0, 0}]]]
```

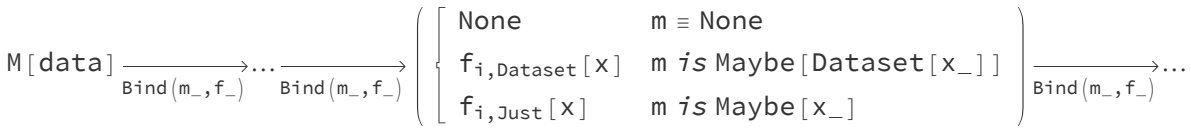
» `{0, 2, 1}`

`Maybe[{0, 2, 1}]`

Polymorphic behavior extensions

Let us extend the Maybe monad developed so far with handling of `Dataset` objects. This turns out is fairly easy and straightforward.

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



Maybe monad Dataset extension definitions

The only reason we have to provide a another definition of `MaybeBind` is because of how `FreeQ` works over `Dataset`.

```
ClearAll[MaybeBind]
MaybeBind[Maybe[x_], f_] :=
  Block[{res = f[x]},
   If[MatchQ[res, Maybe[_Dataset]],
    If[FreeQ[res[[1]], None], res, None],
    If[FreeQ[res, None], res, None]
   ]
  ];
```

Next we just provide the Dataset versions of the Maybe monad special definitions (that work with data):

```
MaybeRandomChoice[n_][xs_Dataset] := Maybe@RandomChoice[xs, n];
```

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

```
MaybeMapToFail[critFunc_][xs_Dataset] := Maybe@Dataset@Map[If[critFunc[#], None, #] &, xs];
```

```
MaybeRandomReal[_Dataset] := Maybe@RandomReal[];
```

Titanic data

Let us load the “Titanic” dataset with the following commands:

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

Here is a sample:

```
RandomSample[ds, 6]
```

passenger class	passenger age	passenger sex	passenger survival
3rd	30.	male	died
3rd	20.	male	survived
3rd	17.	male	died
1st	19.	female	survived
3rd	19.	male	survived
1st	21.	female	survived

Demonstrating polymorphic behavior

In order to demonstrate the polymorphic behavior we going to use the functions `RecordsSummary` and `CrossTabulate` from the package [7].

Dataset

```
SeedRandom[7]
MaybeUnit[ds]⟹
 MaybeFilter[#[[2]] > 45 &]⟹
 MaybeEchoFunction["\t\tSummary after filtering:" &]⟹
 MaybeEchoFunction[RecordsSummary]⟹
 MaybeEchoFunction[MatrixForm@CrossTabulate[#[[All, {1, 4}]] &]⟹
 MaybeEchoFunction["\t\tPick 5 rows:" &]⟹
 MaybeRandomChoice[5]⟹
 MaybeEcho⟹
 MaybeEchoFunction["\t\tTest result:" &]⟹
 MaybeEchoFunction[MaybeFilter[#[[4]] == "died" &]]⟹
 MaybeMapToFail[#[[4]] == "died" &]
```

» Summary after filtering:

» 1 passenger class

1st	102
2nd	32
3rd	21

2 passenger age

Min	45.5
1st Qu	49.
Median	53.
Mean	54.629
3rd Qu	60.
Max	80.

3 passenger sex

male	103
female	52

4 passenger survival

died	93
survived	62

»

	died	survived
1st	50	52
2nd	24	8
3rd	19	2

» Pick 5 rows:

»

passenger class	passenger age	passenger sex	passenger survival
2nd	50.	male	died
3rd	49.	male	died
1st	47.	male	died
1st	58.	male	died
1st	54.	male	died

» Test result:

» Maybe[

passenger class	passenger age	passenger sex	passenger survival
2nd	50.	male	died
3rd	49.	male	died
1st	47.	male	died
1st	58.	male	died
1st	54.	male	died

]

None

Array

The same pipeline can be executed over an array instead of a `Dataset` object. The results are the same, but have a different form in some cases.

```
tbl = Normal[ds[All, Values]];
```

```
SeedRandom[7]
MaybeUnit[tbl] ⟹
 MaybeFilter[#[[2]] > 45 &] ⟹
 MaybeEchoFunction["\t\tSummary after filtering:" &] ⟹
 MaybeEchoFunction[RecordsSummary] ⟹
 MaybeEchoFunction[MatrixForm@CrossTabulate[#[[All, {1, 4}]] &] ⟹
 MaybeEchoFunction["\t\tPick 5 rows:" &] ⟹
 MaybeRandomChoice[5] ⟹
 MaybeEcho ⟹
 MaybeEchoFunction["\t\tTest result:" &] ⟹
 MaybeEchoFunction[MaybeFilter[#[[4]] == "died" &]] ⟹
 MaybeMapToFail[#[[4]] == "died" &]
```

```
» Summary after filtering:
```

1 column 1	
1st	102
2nd	32
3rd	21

2 column 2	
Min	45.5
1st Qu	49.
Median	53.
Mean	54.629
3rd Qu	60.
Max	80.

3 column 3	
male	103
female	52

4 column 4	
died	93
survived	62

	died	survived
1st	50	52
2nd	24	8
3rd	19	2

```
» Pick 5 rows:

» {{2nd, 50., male, died}, {3rd, 49., male, died}, {1st, 47., male, died}, {1st, 58., male, died}, {1st, 54., male, died}}

» Test result:

» Maybe[{{2nd, 50., male, died}, {3rd, 49., male, died}, {1st, 47., male, died}, {1st, 58., male, died}, {1st, 54., male, died}}]

  None
```

Programming and using the State monad

Here we generate the basic functions of a State monad for the symbol “StMon”:

```
Get["~/MathFiles/MathematicaForPrediction/MonadicProgramming/StateMonadCodeGenerator.m"]
Get["~/MathFiles/MathematicaForPrediction/MonadicProgramming/MonadicTracing.m"]

Get["~/MathFiles/MathematicaForPrediction/MathematicaForPredictionUtilities.m"]

GenerateStateMonadCode["StMon", "StringContextNames" → False, "FailureSymbol" → None]
```

```
SeedRandom[232]
Quiet[
 res =
   TraceMonadUnit[StMonUnit[RandomReal[{0, 1}, {3, 2}], <|"color" → Red, "threshold" → 0.5|>]] ⟹
    "(* generate a random matrix and lift into the monad *)"⟹
    StMonEchoValue⟹
    (StMon[#1 /. (x_ /; x < #2["threshold"] :↦ Style[x, #2["color"]]), Join[#2, <|"data" → #1|>]] &) ⟹
    "(* if less that the context threshold: color in red *)"⟹
    StMonEchoValue⟹
    StMonModifyContext[Join[#1, <|"color" → Blue, "threshold" → 0.8|>] &]⟹"(* modify the context *)"⟹
    (StMon[#2["data"] /. (x_ /; x < #2["threshold"] :> Style[x, #2["color"]]), #2] &) ⟹
    "(* if less that the new context threshold: color in blue *)"⟹
    StMonEchoValue⟹
    TraceMonadEchoGrid[
     GridTableForm[#, TableHeadings → {"code", "comment"}, Dividers → None, Background → Automatic] &];
]
```

```
» {{0.0813545, 0.643847}, {0.700647, 0.996779}, {0.861355, 0.287696}}
» {{0.0813545, 0.643847}, {0.700647, 0.996779}, {0.861355, 0.287696}}
» {{0.0813545, 0.643847}, {0.700647, 0.996779}, {0.861355, 0.287696}}
```

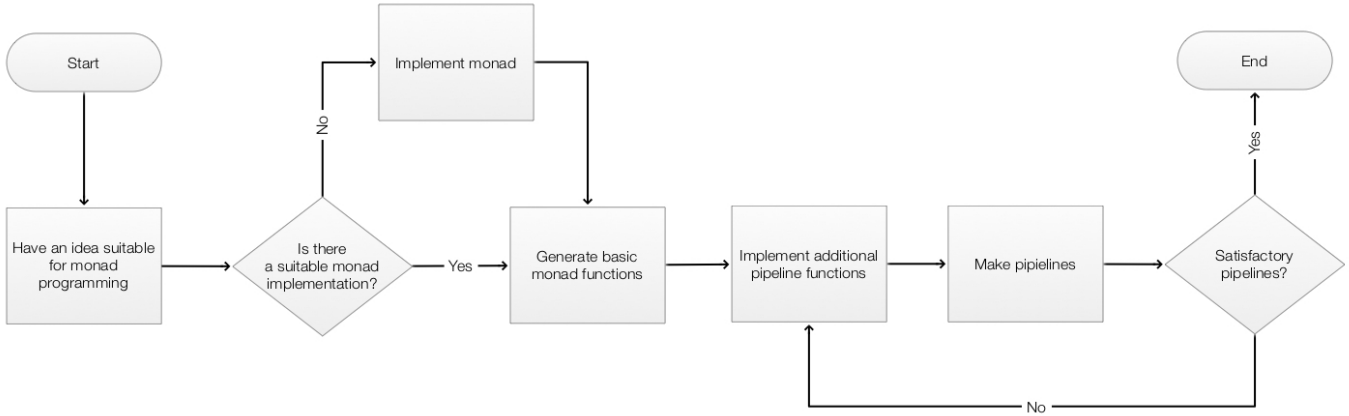
#	code	comment
1	`StMonUnit[RandomReal[{0, 1}, {3, 2}], Association[color → Red, threshold → 0.5`]] ⟹`	`(* generate a random matrix and lift into the monad *)`
2	`StMonEchoValue ⟹`	
3	`StMon[#1 /. x_ /; x < #2[threshold] :↦ Style[x, #2[color]], Join[#2, Association[data → #1]]] & ⟹`	`(* if less that the context threshold: color in red *)`
4	`StMonEchoValue ⟹`	
5	`StMonModifyContext[Join[#1, Association[color → Blue, threshold → 0.8`]] &] ⟹`	`(* modify the context *)`
6	`StMon[#2[data] /. x_ /; x < #2[threshold] :↦ Style[x, #2[color]], #2] & ⟹`	`(* if less that the new context threshold: color in blue *)`
7	`StMonEchoValue`	

```
Fold[StMonBind, ReleaseHold[res[[2]]["data"]], res[[2]]["commands"]]
» {{0.19695, 0.760793}, {0.105671, 0.550989}, {0.112365, 0.0993635}}
» {{0.19695, 0.760793}, {0.105671, 0.550989}, {0.112365, 0.0993635}}
» {{0.19695, 0.760793}, {0.105671, 0.550989}, {0.112365, 0.0993635}}
StMon[{{0.19695, 0.760793}, {0.105671, 0.550989}, {0.112365, 0.0993635}},
 ⟨|color → ■, threshold → 0.8, data → {{0.19695, 0.760793}, {0.105671, 0.550989}, {0.112365, 0.0993635}}|⟩]
```

How to use monad programming in Mathematica / WL

General work-flow

The following diagram shows my current vision of how to use monadic programming in Mathematica.



Monad code generators and monad completions (packages)

Case study: Motivation and implementation of TraceMonad

Retrospect and summary

Personal biases

Statements like:

> "Monads are the leading design pattern for functional programming."

tend to annoy me -- I see them as a tunnel vision perspective derived from monadic programming.

My biased view on monads is that they are loudly advertised in Haskell, Scala, and F#, but I kind of do not see what is the big deal. Granted, similar things are said about Object-Oriented Programming Design Patterns, but I like them and use them a lot, [2]. In part this article was written to convince myself that monads are not a big deal.

Monad programming is more important for functional programming languages like Haskell and Scala because of their strong type function definition. Not so much in LISP-like languages.

Monad programming style is not so hard to apply/use in LISP or Mathematica. That can be done in several ways. I think the approach presented here is the simplest in WL using “first principles.”

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 .