List-based Monadic Computations for Dynamically Typed Languages

Wim Vanderbauwhede
School of Computing Science,
University of Glasgow
Glasgow, United Kingdom
wim.vanderbauwhede@glasgow.ac.uk

ABSTRACT

In this paper we propose a flexible and elegant design for list-based monadic ("karmic") computations in dynamically typed languages. Our approach is applicable to any dynamically typed language that supports lists, maps and higher-order functions. The design results in a clear, concise and largely language-independent syntax.

We prove that our design adheres to the laws governing monads. We illustrate the use of the list-based monads with three examples: error handling in computations, stateful computations and parser combinators. We provide examples in the popular multi-paradigm language Perl, the object-oriented language Ruby and the functional dynamically typed language LiveScript.

Categories and Subject Descriptors

D.3 [**Programming Languages**]: Language Constructs and Features; D.2 [**Software Engineering**]: Coding Tools and Techniques

General Terms

Languages

Keywords

Dynamic Languages, Monads, Perl, Haskell

1. BACKGROUND

Programming language researchers are familiar with the concept of monads as programming structures in statically typed functional languages [1], as popularised in particular by the functional language Haskell [2]. However, monads have a reputation of being esoteric, hard to understand and the province of academic languages rather than of practical us in day-to-day programming.

Dynamically typed languages ("scripting languages") such as Python, Perl, Ruby or JavaScript are very popular in an increasing number of application domains.

Dyla'14, June 09-11 2014, Edinburgh, United Kingdom

ACM ACM 978-1-4503-2916-3/14/06 ...$15.00.
http://dx.doi.org/10.1145/2617548.2617551.

In this paper, we aim to make the case for the usefulness of monads in dynamically typed languages, and present a practical approach to monadic computations in such languages. We call this approach *karmic* computations, after the Sanskrit word for actions, *karma*, and to separate our approach from the many existing implementations of monads in dynamic languages[1].

For our examples we will use three very different dynamic languages: Perl [3, 4], a language which has many detractors in academia but remains nevertheless very popular in industry, as well as being a very powerful multi-paradigm dynamic language, Ruby [5], an object-oriented language inspired by Smalltalk and Perl, and LiveScript [6], a functional dynamically typed language with a syntax very close to that of Haskell. LiveScript is a fork of CoffeeScript, both languages are compiled into JavaScript [7].

We assume a familiarity with Perl and/or Ruby, and encourage the reader to have a look at LiveScript as it will help with the understanding of the code examples. We also assume the reader to understand the notion of types and lambda functions (a.k.a. anonymous subroutines). Familiarity with Haskell is not required but will make it easier to follow the discussion, as many of the concepts used are borrowed from Haskell.

1.1 Notational Conventions

Throughout the paper, we will use and italic variable-width serif font for type declarations, and a fixed-width serif for actual code examples. Code in blue indicates the code that the programmer actually has to write when using the proposed approach and provided libraries.

1.2 Type Notation

In this paper, we will use a Haskell-style syntax for types, for example:

$$g :: a \rightarrow b$$

represents a function from a type a to a type b, where a and b can be different types

whereas

$$h :: a \rightarrow a$$

represents a function from a type a to a type a, so the argument type can be anything but the return value must be of the same type. Furthermore,

$$f :: a \rightarrow m\,b$$

[1]See e.g. the Wikipedia page on monads in functional programming

Language	Lambda Syntax
Haskell	`\x y -> f x y`
LiveScript	`(x,y) -> f x,y`
Perl	`sub ($x,$y) { f $x,$y }`
Python	`lambda x,y :  f(x,y)`
Ruby	`-> (x,y) { f x,y }`

Table 1: Lambda syntax in various languages

represents a function from a type a to a type $m\ b$, where we can view $m\ b$ as a function m of b. Using Ruby syntax, this signature would apply to the following example function

```
def f(x)
  m.new( g(x) )
end
```

Here, x is of some dynamic type a, and is transformed into the type $m\ b$, i.e. an instance of the object of class m taking a constructor argument of type b, where b is the return type of g.

1.3 Lambda Functions

Karmic computations require higher-order functions, so we need functions-as-values, also know as lambda functions or anonymous subroutines. These are functions that can be assigned to a scalar variable.

Many dynamically typed languages provide these, Table 1 provides a summary.

1.4 What is Karma?

For the purpose of this paper, a karmic computation consists of

- a given type $m\ a$ (for example a list or object or a function),
- the functions *bind* and *wrap*[2],
- and three rules (the "monad laws") governing those functions.

1.4.1 Karmic Types

Typically, a karmic type can be viewed as a "wrapper" around another type. The key point is that the "karmic" type contains more information than the "bare" type.

1.4.2 Combining Functions with bind and wrap

The purpose of the karmic computation is to combine *functions* that operates on karmic types. The *bind* function composes values of type $m\ a$ with functions from a to $m\ b$:

$$bind\ ::\ m\,a\ \rightarrow\ (\,a\ \rightarrow\ m\,b)$$

The *wrap* function takes a given type a and returns m a:

$$wrap\ ::\ a\ \rightarrow\ m\,a$$

With these two functions we can create expressions that consists of combinations of functions.

[2]we use *wrap* instead of the more common name *return* because *return* is a reserved word in most dynamic languages.

1.4.3 The Rules

The three rules governing *bind* and *wrap* ensure that the composition is well-behaved: in order to have a valid monad, the definitions of *bind* and *wrap* must be such that:

1. *bind (wrap x) f* $\equiv$ *f x*
2. *bind m wrap* $\equiv$ *m*
3. *bind (bind m f) g* $\equiv$ *bind m (\x -> bind (f x) g)*

The first rules says that wrapping a value in a monad and then binding it to a function is the same as applying the function to the value.

The second rule says that binding a monadic value to a call to wrap returns the monadic value

The third rule is the associativity rule, which must be satisfied in order to allow expressions without parentheses.

What karmic computations give us is a powerful way of combining function-based computations by abstracting the boilerplate code required for more conventional ways of combining computations.

1.5 Monads in Haskell

In Haskell [8], monads are an integral part of the language: it is not possible to perform I/O in Haskell except through a monadic computation.

One of the main reasons why monads work well in Haskell is the syntax. If one had to write expressions such as (in Haskell syntax):

```
ex x = \x -> (bind (g x) (\y ->
      (f (bind y (\z -> wrap z)))))
```

or the equivalent in Ruby:

```
def ex(x)
  return bind(g( x), lambda {|y|
    f (bind(y,lambda{ |z| wrap(z) }) })
end
```

then monads would be too unwieldy to be practical. Haskell however provides the do-notation as syntactic sugar [9]:

```
ex x = do
  y <- g x
  z <- f y
  wrap z
```

Furthermore, in Haskell the bind function is actually an operator (`>>=`). The do-notation is actually a syntactic sugar over the operator-based syntax:

```
ex x =
  g x >>= \y ->
  f y >>= \z ->
  wrap z
```

1.6 Karmic Computations in Dynamically Typed Languages

As Haskell is a statically typed, pure, lazy functional language, it is not a priori clear that the concept of monads can be transferred to dynamically typed languages, most of which are also impure and eager. We present in this paper a list-based syntax for karmic computations in dynamically typed languages, which allows one to write elegant expressions such as e.g. the following parser (for a Fortran-95 type declaration):

```
[
  identifier,
  maybe parens choice
  natural,
  [
    symbol 'kind',
    symbol '=',
    natural
  ]
]
```

The details of this example are explained in Section 3.3. For the moment, we only present it to illustrate our goal: the above expression is valid syntax in Perl, Ruby and LiveScript; it is easy to read and represents a complex computation.

2. DESIGNING KARMA

As can be seen from the above example, we will use the list syntax to implement our karmic computations. Grouping items in lists provides a natural sequencing mechanism, and allows easy nesting. In most scripting languages, lists are represented using square brackets and commas, which provides us with a portable and non-obtrusive syntax.

We introduce a *chain* function which serves a purpose similar to the *bind* function: to chain computations together. The idea is that we pass *chain* a list of functions and it will combine these functions into a single expression. We now define *chain* in terms of the >>= operator, the *wrap* function and the *foldl* higher-order function. The *foldl* (left fold) function performs a reduction of a list via iterative application of a function to an initial value:

$$foldl :: (a \to b \to a) \to a \to [b] \to [a]$$

A possible implementation e.g. in Ruby would be

```
def foldl (f, acc, lst) {
  for elt in lst
    acc=f.call(acc,elt)
  end
  return acc
end
```

We define *chain* as

```
chain fs =
  \x -> foldl (>>=) (wrap x) fs
```

This definition is valid in Haskell, and consequently, one can use list-based monads in Haskell, but only if the type signature is

$$chain :: [(a \to m\,a)] \to (a \to m\,a)$$

In other words, the karmic type $m\ b$ returned by *bind* does not have to be the same as the karmic type argument $m\ a$. This means that every function in the list passed to *chain* should be allowed to have a different type. This is not allowed in a statically typed language like Haskell, so only the restricted case of functions of type a -> $m\ a$ can be used. However, in a dynamically typed language there is no such restriction, so we can generalise the type signature:

$$chain :: [(a_{i-1} \to m\,a_i)] \to (a_0 \to m\,a_N)\ \forall i \in 1\,..\,N$$

With this definition, we can define computations as

```
comp = chain [
  f,
  g,
  h
]
```

where *f*, *g*, *h* evaluate to lambda functions of type *a* -> *m b*.

Applying the definition of *chain* to this example, we can write it out as

```
comp = \x ->
  wrap x >>= f >>= g >>= h
```

In other words, *chain* defines a karmic computation if *bind* and *wrap* satisfy the monadic laws.

3. LIST-BASED KARMIC COMPUTATIONS BY EXAMPLE

We now illustrate the use of the list-based karmic computations through a number of examples. The first example, in Ruby, shows how to combine computations that can fail. The second, in LiveScript, shows how to do stateful computations without actual mutable state. The final example, in Perl, is a parser combinator library. The full code for the examples can be found at [10]; the proofs of adherence to the monadic laws are given in the Appendix.

3.1 Combining Computations that can Fail

A common pattern in many programs is that of dealing with a computation that can fail. One way to express the success or failure, esp. in an object-oriented language like Ruby, is to create a class *Maybe* with two attributes, *val* and *ok*.

```
class Maybe
  attr_accessor :val,:ok
  def initialize(value,status)
    @val=value
    @ok=status # true or false
    end
end
```

For every operation that can fail, we return a *Maybe* object, with *ok* set to *true* if the computation was succesfull, *false* otherwise. Suppose we have three functions *f*, *g*, *h* that take a value and return an instance of *Maybe*. Apart from this signature (*a* -> *Maybe b*), the functions are completely generic. We will assume these are functions defined with the *def* keyword in Ruby, not lambda functions. We define a value *fail* for convenience:

```
fail = Maybe.new(nil,false)
```

Without karma, we would need to write something like

```
v1 = f(x)
  if (v1.ok)
    v2=g(v1.val)
    if (v2.ok)
      v3 = h(v2.val)
      if (v3.ok)
        v3.val
      else
        fail
      end
    else
      fail
    end
```

```
    else
      fail
    end
  end
```

Using the karmic approach we can write this as

```
comp = chain [
->(x) {f x},
->(y) {g y},
->(z) {h z}
]
v1=comp.call x
```

where *bind* is defined as

```
def chain(x,f)
  if x.ok
    f.call(x)
  else
    fail
  end
end
```

and *wrap* as

```
def wrap(x)
  Maybe.new(x,true)
end
```

Note that if f, g and h had been defined as lambda functions, we could write simply

```
comp = chain [ f,g,h ]
```

3.2 Stateful Computation

The next example, in LiveScript, illustrates how karmic computations can be used to perform stateful computations without actual mutable state. This is particularly relevant in LiveScript because the LiveScript interpreter *lsc* has a *–const* flag which compiles all variables as constants, in other words LiveScript forces immutable variables. However, there is a growing interest in the use of immutable state in other dynamic languages, in particular for web programming, because immutable state greatly simplifies concurrent programming and improves testability of code.

3.2.1 Karmic Computations with State

We use the same approach as in Haskell's Control.State monad [2]: the state is simulated by combining functions that transform the state, and applying the resulting function to an initial state. We create a small API which allows the functions in the list to access a shared state. The API consists of two functions[3]:

```
get = -> ( (s) -> [s, s] )
put = (s) -> ( -> [[],s])
```

The *get* function reads the state, the *put* function writes an updated state. Both are functions that return lambda functions.

Furthermore, we have a top-level function to apply the stateful computation to an initial state:

```
res = evalState comp, init
```

where

```
evalState = (comp,init) ->
  comp!(init)
```

[3]In LiveScript, functions without arguments, e.g. f = -> 42, can be called with a ! instead of (), i.e. f!

3.2.2 A Trivial Interpreter

We use this approach to write an trivial interpreter. We want to interpret a list of instructions of type

$$Instr = (String, ([Int] \to Int, [String]))$$

for example

```
instrs = [
  ["v1",[set,[1]]],
  ["v2",[add,["v1",2]]],
  ["v2",[mul,["v2","v2"]]],
  ["v1",[add,["v1","v2"]]],
  ["v3",[mul,["v1","v2"]]]
]
```

where *add*, *mul* and *set* are functions of type [Int] -> Int

The interpreter will need a context for the variables, we use a simple map { String : Int } to store the values for each variable. This constitutes the state of the computation. We write a function

$$interpret\_instr\_st :: Instr \to [[a], Ctxt]$$

```
interpret_instr_st = (instr) ->
  chain( [
    get,
    (ctxt) ->
      [v,res] =
        interpret_instr(instr, ctxt)
      put (insert v, res, ctxt)
  ] )
```

which gets the context from the state, uses it to compute the instruction, updates the context using the *insert* function and puts the context back in the state. Using *Ctxt* as the type of the context, the type of *interpret_instr* is

$$interpret\_instr :: Instr \to Ctxt \to (String, Int)$$

```
interpret_instr = (instr, ctxt) ->
  [v,rhs] = instr
  [op,args] = rhs
  vals = map (`get_val` ctxt), args
  [v,op vals]
```

Here, *get_val* looks up the value of a variable in the context[4]

The pattern of first calling *get* to get the state, then computing using the state and then updating the state using *put* is very common, so it is often combined in a function called *modify*:

```
modify =
  (f) ->
    chain [
      get,
      (s) -> put( f(s))
    ]
```

Using *modify* we can rewrite *interpret_instr_st* as

```
interpret_instr_st = (instr) ->
  modify (ctxt) ->
    [v,res] =
       interpret_instr(instr, ctxt)
    insert v, res, ctxt
```

[4]In LiveScript, partial application only works for operators, but backticks turn any two-argument function into an operator.

We could in principle use *chain* to write a list of computations on each instruction:

```
chain [
  interpret_instr_st instrs[0],
  interpret_instr_st instrs[1],
  interpret_instr_st instrs[2],
  ...
]
```

but clearly this is not practical as it is entirely static. Instead, we use a monadic variant of the *map* function:

$$mapM \;::\; Monad\, m \Rightarrow (a \to m\, b) \to [a] \to m\, [b]$$

to write

```
ctxt = execState (
    mapM interpret_instr_st, instrs
  ), {}
```

The monadic map *mapM* essentially applies the computations to the list of instructions (using an ordinary *map*) and then folds the resulting list using *bind*.

3.3 Parser Combinators

As a final example we describe the design of a parser combinator library in Perl[5], very similar to Haskell's Parsec library [11]. Each *basic parser* returns a function which operates on a string (i.e. it attempts to parse the string) and returns a tuple containing the status (success or failure), the remaining string and the substring(s) matched by the parser.

$$Parser \;::\; String \to Result$$

$$type\ Result \;=\; (Status, String, [Match])$$

A detailed discussion of the Match type follows; for the moment it can be considered an alias for String.

To create a complete parser, the basic parsers are combined using *combinators*. A combinator takes a list of parsers and returns a parsing function similar to the basic parsers:

$$combinator \;::\; [Parser] \to Parser$$

For the sequential application of the parsers we naturally use the *chain* function, which can be considered a generic combinator. Our aim is to allow the programmer to write expressions such as the following example, a parser for a Fortran-95 type declaration:

```
[
  identifier,
  maybe parens choice
  natural,
  [
    symbol 'kind',
    symbol '=',
    natural
  ]
]
```

As discussed above, in dynamic languages the actual *chain* combinator can be implicit, i.e. a bare list will result in calling of *chain* on the list. For statically typed languages this is of course not possible as all elements in a list must have the same type.

It will surprise some readers that the above example is entirely valid Perl syntax.

To make the parser more user-friendly we want provide the ability to create a parse tree which returns only those parsed strings ("fields") that the user is interested in, with an appropriate label. In Parsec this is achieved by extracting the fields using the "<-" syntax and wrapping them into the returned type, e.g.

```
do
  t <- tag_parser
  return (Tag t)
```

Our approach is to wrap the fields of interest in an anonymous map (key-value pair). The corresponding Perl syntax is

```
{Tag => tag_parser}
```

Applied to the above example:

```
[
  {F95Type => identifier},
  maybe parens choice
    {F95Kind => natural},
    [
      symbol "kind",
      symbol "=",
      {F95Kind => natural}
    ]
]
```

This approach again exploits the fact that lists in a dynamically typed language are heterogeneous. In our case, the list can contain parsers (type *Parser*), anonymous maps *(Map a)* or lists of these. This of course implies that the anonymous maps can contain the same types, so we also rely on the fact that maps in dynamically typed languages are heterogeneous in their value type. As a result, the type *Match* introduced above will actually be a nested list of maps and/or unlabelled strings.

3.4 Parser Implementation

To implement the basic parsers we have opted to use regular expressions, because the engine is very efficient in Perl and Perl-compatible regular expression and available for most dynamic languages. The library has been implemented both in Perl and LiveScript. We provide the implementation examples in LiveScript.

3.4.1 Basic parsers

The basic parsers are functions that take a string return a tuple. We have adopted the nomenclature of Parsec for basic parsers and combinators. The *natural* basic parser parses an unsigned integer. The implementation is

```
sub natural ($str) {
  if ( /^(\d+)/ ) {
      my $matches=$1;
      my $str2 = $str;
      $str2=~s/^\d+\s*//;
      (1, $str2, $matches)
  } else {
      (0, $str, undef)
  }
}
```

[5]Implementations of this library in Perl and LiveScript can be found at `https://github.com/wimvanderbauwhede/Perl-Parser-Combinators` and `https://github.com/wimvanderbauwhede/parser-combinators-ls`.

3.4.2 Combinators

The combinators are functions that take one or more parsers and return a new parser. A typical combinator is *many1*. As in Parsec, *many1* parses 1 or more instances of a given parser.

```
sub many1 ($p) {
    sub ($str) {
        parse_while(
          apply($p,$str,[]) );
    }
}
sub parse_while ($p,$str,$res) {
    (my $st, my $str, my $m) =
       apply($p,$str);
    if ($st) {
        parse_while(
          $p,$str2,[@$res,@$m]);
    } else {
        (1,$str,$res)
    }
}
```

The *apply* function is a generalisation of function application to deal with the bare lists and maps. It checks if the variable $p contains a parser, a list or a map. If it is a list, it calls *chain*; if it is a map, it calls the parser on the value of the map.

3.4.3 Tentative Parsing

One design choice that is different from Parsec's is that our parsers do not consume input on failure. Consequently, there is no need for *try* as in Parsec. However, in order to express an optional part of a grammar, we have introduced *maybe*, which is in fact a simplified version of *many*. The *maybe* combinator applies its parser argument but always succeeds.

3.5 Applying the Parser and Extracting the Parse Tree

Similar to Parsec, and in fact to many monadic computations, to run the parser function constructed from the basic parsers and combinators on a string, we need another function. We are interested in returning the labeled fields as a parse tree. The data structure returned by running the parser consists of nested lists where some of the elements are key-value pairs. We transform this data structure in a nested map (this transformation is actually not straightforward, see the source code for details).

```
my $parse_tree =
runParser $parser, $string;
```

3.6 Application Example: Fortran-95 Variable Declaration Parser

As a practical (and actually real-world) example, we present a parser for variable declarations in Fortran-95. The syntax of a variable declaration is [12]:

For simplicity, we assume that the order of the attributes is fixed. We define separate parsers for each attribute:

```
my $type_parser = chain [
    {Type => identifier},
    maybe parens choice [
        {Kind => natural},
        [
            symbol 'kind',
            symbol '=',
            {Kind => natural}
        ]
      ]
    ];
```

The code for the other parsers (*dim_parser*, *intent_parser*) is listed in the Appendix. The final parser combines these parsers:

```
my $f95_arg_decl_parser =
    chain [
        whiteSpace,
        {TypeTup => $type_parser},
        maybe [
                comma,
                $dim_parser
            ],
        maybe [
                comma,
                $intent_parser
            ],
        symbol '::',
        {Vars => sepBy ',',identifier}
    ];
```

Running this parser e.g.

```
my $vardecl =
"integer(8), dimension(0:ip,-
1:jp+1,kp) :: u,v"
my $res =
runParser $f95_arg_decl_parser, $varsdecl;
```

results in the parse tree

```
{
   'TypeTup' => {
                'Type' => 'integer',
                'Kind' => '8'
            },
   'Dim' => ['0:ip','-1:jp+1','kp'],
   'Vars' => ['u','v']
}
```

4. CONCLUSIONS

We have presented a design for list-based monadic computations, which we call *karmic computations*. We have shown that the karmic computations are entirely equivalent to monadic computations in statically typed languages such as Haskell, but rely essentially on the dynamically typed nature through the use of heterogeneous lists.

There is currently one main limitation for karmic computations in dynamically types languages, compared to Haskell's monads: because of the lack of typeclasses, it is not possible to write a generic *chain* function in terms of a specific implementation of *bind* and *wrap*, in other words *chain* needs to be defined in the same library as *bind* and *wrap*. This also implies that combining different karmic computations can only be achieved through explicit qualification of the *chain* function with its library namespace. We are currently working on an implementation of typeclass-like functionality for dynamically typed languages to allow combining of karmic computations on different types.

Through the examples in the previous section we have shown that karmic computations can be useful for a variety of problems and that the proposed list-based syntax makes them easy to use, and results in concise, elegant and very readable code that is largely language-independent. As a result we think the proposed approach can be applied very widely, especially as a design technique for libraries.

5. REFERENCES

[1] E. Moggi, "Notions of computation and monads," *Information and computation*, vol. 93, no. 1, pp. 55–92, 1991.

[2] B. O'Sullivan, J. Goerzen, and D. B. Stewart, *Real world haskell*. O'Reilly Media, Inc., 2008.

[3] L. Wall and R. L. Schwartz, *Programming perl*. O'Reilly & Associates, Inc., 1991.

[4] M. J. Dominus, *Higher-order perl: transforming programs with programs*. Morgan Kaufmann, 2005.

[5] D. Thomas, C. Fowler, and A. Hunt, *Programming Ruby*. Pragmatic Bookshelf, 2004, vol. 13.

[6] G. Zahariev. (2013) Livescript – a language which compiles to JavaScript. [Online]. Available: http://livescript.net

[7] D. Crockford, *JavaScript: the good parts*. O'Reilly Media, Inc., 2008.

[8] S. L. P. Jones, *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[9] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson *et al.*, "Report on the programming language haskell: a non-strict, purely functional language version 1.2," *ACM SigPlan notices*, vol. 27, no. 5, pp. 1–164, 1992.

[10] W. Vanderbauwhede. (2014) Karmic computations. [Online]. Available: https://github.com/wimvanderbauwhede

[11] D. Leijen and E. Meijer, "Parsec: Direct style monadic parser combinators for the real world," Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, Tech. Rep., 2001.

[12] J. C. Adams, W. S. Brainerd, J. T. Martin, B. T. Smith, and J. L. Wagener, "fortran 90 handbook," *Intertext-McGraw Hill*, 1992.

APPENDIX

A. PROOF OF MONADIC LAWS FOR THE EXAMPLES

A.1 Computations that can Fail

The computation returns a tuple with the first element *true* if successful, *false* if the computation failed. This is equivalent to Haskell's *Maybe* monad.

```
wrap = (x) -> [true,x]

bind = (x,f) ->
    [e,v] = x
    if e
        f v
    else
        [e,v]
```

Monadic laws:

1. (wrap v) 'bind' f == f v

   PROOF. (wrap v) 'bind' f
   = [true,v] 'bind' f'
   = f v (because e = true) □

2. m 'bind' wrap == m

   PROOF. m = [e,v]
   [e,v] 'bind' wrap
   e == false ⇒ [e,v] = m
   e == true⇒ f v = wrap v
   = [true,v]
   = m □

3. (m 'bind' f) 'bind' g == m 'bind' (\x -> f x 'bind' g)

   PROOF. (m 'bind' f) 'bind' g
   m = [false,v] ⇒[false,v]
   m = [true,v] ⇒ f v 'bind' g
   f v = [false,v'] ⇒[false,v']
   f v = [true,v'] ⇒ g v'
   (\x -> f x 'bind' g) v = f v 'bind' g (see above) □

A.2 Stateful Computation

This is analogous to Haskell's *State* monad.

```
wrap = (x) -> ( (s) -> [x,s] )

bind = (f,g) ->
  (st) ->
    [x,st_] = f st
    (g x) st_
```

Monadic laws:

1. (wrap x) 'bind' g == g x

   PROOF. bind ((s) -> [x,s]), g
   = (st) ->
   [x',st'] = ((s) -> [x,s])) st
   (g x) st'
   = (st) ->
   [x',st'] = [x,st]
   (g x') st'
   = (st) ->
   (g x) st
   = g x □

2. m 'bind' wrap == m

   PROOF. m $\stackrel{def.}{=}$ (st) -> [x,st]
   bind ((s) -> [x,s]), (x') -> ((s') -> [x',s'])
   = (st) ->
   [x",st"] = ((s) -> [x,s]) st
   ((x') -> ((s') -> [x',s']) x") st"
   = (st) ->
   [x",st"] = [x,st]
   ((s_) -> [x",s']) st"
   = (st) ->
   [x",st"] = [x,st]
   [x",st"]
   = (st) ->
   [x,st]
   = m □

3. (m 'bind' f) 'bind' g == m 'bind' ((x) -> f x 'bind' g)

PROOF.

1. RHS

bind (bind m, f), g
= (st) ->
  [x",st"] = (bind m, f) st
  (g x") st"

(bind m, f) st =
  [x",st"] = m st
  (f x") st"
⇒
= (st) ->
  [x',st'] = m st
  [x",st"] = (f x') st'
  (g x") st"

2. LHS

m 'bind' ((x) -> f x 'bind' g)
= bind m, ((x) -> f x 'bind' g)

bind (f x), g =
 (st) ->
  [x',st'] = (f x) st
  (g x') st'

((x) -> f x 'bind' g) x'
= bind (f x'), g
⇒
= (st) ->
  [x',st'] = m st
  (bind (f x'), g) st'
= (st) ->
  [x',st'] = m st
  [x",st"] = (f x') st'
  (g x") st"
= RHS □

A.3 Parser Combinators

The parser combinators are inspired by Haskell's Parsec.

```
wrap = (str) -> [true,str,[]]

bind = (t, p) ->
  [st_,r_,ms_] = t
    if st_
      [st,r,ms] = p r_
      if st
        then
          [true,r,ms_ ++ ms]
        else
          [false,r,[]]
    else
      [false,r_,[]]
```

Furthermore,

```
p str =
  | <parse succeeds> => [true, r, ms]
  | <parse fails> => [false, str, []]
```

where r is the remainder of the string *str* and *ms* is a list of matches.

Monadic laws:

1. (wrap str) 'bind' p == p str

   PROOF. bind (wrap str), p =
  [st',r',ms'] = wrap str = [true,str,[]]
  if st'
    [st,r,ms] = p r' = p str
      if st
        then [true,r,[] ++ ms] = [true,r,ms]
        else [false,str,[]]
      else
        [false,str,[]] □

2. t 'bind' wrap == t

   PROOF. *t* can either be *[true, r, ms]* or *[false, str, []]*
  t 'bind' wrap =
    [st',r',ms'] = t
    if st' then
      [st,r,ms] = ((str) -> [false,str,[]]) r'
      if st
        then [true,r,ms_ ++ ms]
        else [false,r,[]]
    else
      [false,r_,[]]
  Case 1: t = [true,r,ms]⇒
  t 'bind' wrap =
    [st',r',ms'] = [true,r,ms]
    [st",r",ms"] = [true,r',[]]
    [true,r",ms' ++ ms"]
  = [true, r, ms]
  = t *(true)*
  Case 2: t = [false,str,[]]⇒
  t 'bind' wrap =
    [st',r',ms'] = [false,str,[]]
    [false,r',[]]
  = [false,str,[]]
  = t *(false)* □

3. (t 'bind' p_1) 'bind' p_2 == t 'bind' ((str) -> (p_1 str) 'bind' p_2)

   This proof is a little more involved because there are several cases depending on the status of t and of the application of p_1 and p_2.

   PROOF. We consider the LHS and RHS expressions separately. *t* can either be *[true, r, ms]* or *[false, str, []]*.

   1. LHS:

  bind (bind t, p_1),p_2=
    [st',r',ms'] = (bind t, p_1)
    if st'
      [st,r,ms] = p_2 r'
      if st
        then [true,r,ms' ++ ms]
        else [false,r,[]]
    else [false,r_,[]]
  bind t, p_1=

[st',r',ms'] = t
if st' then
[st,r,ms] = p_1 r_
if st
then [true,r,ms_ ++ ms]
else [false,r,[]]
else [false,r_,[]]

LHS Case 1: t false or p_1 r' false ⇒ bind t,p_1 = [false,str,[]]⇒

bind (bind t, p_1),p_2=
[st',r',ms'] = [false,str,[]]
if st' then
[st,r,ms] = p_2 r'
if st
then [true,r,ms' ++ ms]
else [false,r,[]]
else [false,r_,[]]

= [false,str,[]] (i.e. t false, whatever p_1 and p_2)

LHS Case 2.1: t true; *bind t,p_1 = [true,r,ms' ++ ms]* $\overset{not.}{=}$ *[true, r_1, ms_1]*⇒

bind (bind t, p_1),p_2=
[st',r',ms'] = [true,r_1,ms_1]
[st,r,ms] = p_2 r'
if st
then [true,r,ms' ++ ms]
else [false,r,[]]

⇒

[st,r,ms] = p_2 r_1
if st
then [true,r,ms_1 ++ ms] $\overset{not.}{=}$ [true,r_2,ms_2]
else [false,r,[]]

(LHS Case 2.1.a)

= [true,r_2,ms_2] *(i.e. t true, p_1 str true, p_2 r_1 true)*

(LHS Case 2.1.b)

= [false,r_1,[]] *(i.e. t true, p_1 str true, p_2 r_1 false)*

LHS Case 2.2: t true; bind t,p_1 = [false,str,[]] ⇒

= [false,r,[]] *(i.e. t true, p_1 str false, whatever p_2)*

2. RHS:

bind (p_1 str), p_2 =

Case 1: p_1 str = [false,str,[]]⇒

[st',r_1,ms'] = [false,str,[]]
if st' then
[st,r_2,ms] = p_2 r_1
if st
then [true,r_2,ms' ++ ms]
else [false,r_1,[]]
else
[false,r_1,[]]

= [false,str,[]]

bind t, ((str) -> [false,str,[]])

= [false, str,[]] *(p_1 str false, whatever t or t false, whatever p_1 str; whatever p_2)*

= LHS Case 2.2 or LHS Case 1

RHS Case 2: p_1 str = [true, r_1, ms_1]⇒

[st',r',ms'] = [true,r_1,ms_1]
[st,r,ms] = p_2 r'
if st
then [true,r,ms' ++ ms]
else [false,r,[]]

⇒

[st,r_2,ms] = p_2 r_1
if st
then [true,r_2,ms_1 ++ ms] $\overset{not.}{=}$ [true, r_2, ms_2]
else [false,r_1,[]]

RHS Case 2.1: bind t, ((str) -> [true, r_2, ms_2]) ⇒

= [true, r_2, ms_2] *(i.e. t true; p_1 str true; p_2 r_1 true)*

= LHS Case 2.1.a

RHS Case 2.2: bind t, ((str) -> [false, r_1, []])⇒

= [false, r_1, []] *(i.e. t true; p_1 str true; p_2 r_1 false)*

= LHS Case 2.1.b □

B. FORTRAN-95 VARIABLE DECLARATION PARSER EXAMPLE

This is the complete code for a "real-world" Fortran-95 variable declaration parser.

```
use Parser::Combinators;

my $dim_parser = chain [
    symbol 'dimension',
    {Dim => parens sepBy ',', regex '[^,\)]+' }
    ];
my $intent_parser = chain [
    symbol 'intent',
    {Intent => parens identifier}
    ];
my $arglist_parser = chain [
    symbol '::',
    {Args => sepBy ',', identifier }
    ];
my $openacc_pragma_parser = chain [
    char('!'),
    whiteSpace,
    choice( symbol('$ACC'), symbol('$acc')),
    {AccPragma => sequence [
    {AccKeyWord => word},
    {AccVal => word}
    ] }
    ]
}
my $f95_arg_decl_parser =
    chain [
        whiteSpace,
        {TypeTup => $type_parser},
        maybe [
                comma,
                $dim_parser
            ],
        maybe [
                comma,
                $intent_parser
            ],
        $arglist_parser,
        maybe($openacc_pragma_parser)
    ];
```