Programming with Applicative-like expressions

Jan Malakhovski and Sergei Soloviev*

IRIT, University of Toulouse-3 and ITMO University

2015 - 2019

Abstract

The fact that `Applicative` type class allows one to express simple parsers in a variable-less combinatorial style is well appreciated among Haskell programmers for its conceptual simplicity, ease of use, and usefulness for semi-automated code generation (metaprogramming).

We notice that such `Applicative` computations can be interpreted as providing a *mechanism* to construct a data type with "ports" "pluggable" by subcomputations. We observe that it is this property that makes them so much more convenient in practice than the usual way of building the same computations using conventional composition. We distill this observation into a more general algebraic structure of (and/or technique for expressing) "`Applicative`-like" computations and demonstrate several other instances of this structure.

Our interest in all of this comes from the fact that the aforementioned instances allow us to express arbitrary transformations between simple data types of a single constructor (similarly to how `Applicative` parsing allows to transform from streams of `Char`s to such data types) using a style that closely follows conventional `Applicative` computations, thus greatly simplifying (if not completely automating away) a lot of boiler-plate code present in many functional programs.

Contents

*papers@oxij.org; preferably with paper title in the subject line

1 Preliminaries

This paper describes an algebraic structure (and/or a technique) for expressing certain kinds of computations. The presented derivation of said structure (technique) starts from observing Haskell's `Applicative` type class [1] and then generalizing it. The result, however, is language-agnostic (the same way `Applicative` is, as both structures can be applied to most functional programming languages in some shape or form, even if a language in question can not explicitly express required type signatures) and theoretically interesting (as it points to several curious connections to category theory and logic).

Since the idea grows from Haskell and most related literature uses Haskell, it is natural to express the derivation of the structure (technique) in Haskell. Therefore, this paper is organized as a series of Literate Haskell programs in a single Emacs Org-Mode tree [2, 3] (then, most likely, compiled into the representation you are looking at right now).[1] Moreover, for uniformity reasons we shall also use Haskell type class names for the names of the corresponding algebraic structures where appropriate (e.g. "`Applicative`" instead of "applicative functor") as not to cause any confusion between the code and the rest of the text.

2 Introduction

Let us recall the definition of `Applicative` type class [1] as it is currently defined in the `base` [5] package of Hackage [6]

```
infixl 4 <*>
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

One can think of the above definition as simply providing a generic "constant injector" `pure` and a somewhat generic "function application" `(<*>)` operator. (The referenced `Functor` type class and any related algebraic laws can be completely ignored for the purposes of this article.) For instance, an identity on Haskell types is obviously an `Applicative` with `pure = id` and `(<*>)` being the conventional function application (the one that is usually denoted by simple juxtaposition of terms), but there are many more complex instances of this type class (see [7, 8] for comprehensive overviews of this and related algebraic structures), most (for the purposes of this article) notably, including `Applicative` parsing combinators.

Those are very popular in practice as they simplify parsing of simple data types ("simple" in this context means "without any type or data dependencies between different parts") to the point of triviality. For instance, given appropriate `Applicative` parsing machinery like Parsec [9], Attoparsec [10] or Megaparsec [11] one can parse a simple data type like

```
data Device = Device
  { block :: Bool
  , major :: Int
  , minor :: Int }

exampleDevice :: Device
exampleDevice = Device False 19 1
```

from a straightforward serialized representation with just

```
class Parsable a where
  parse :: Parser a
```

[1] The source code is available at https://oxij.org/paper/ApplicativeLike/. It is also embedded straight into the PDF version of this article (click here  or look for "attachments" in your PDF viewer). All runnable code in the paper was tested with GHC [4] version 8.6.

```
instance Parsable Device where
  parse = pure Device <*> parse <*> parse <*> parse
```

While clearly limited to simple data types of a single[2] constructor, this approach is very useful in practice. Firstly, since these kinds of expressions make no variable bindings and all they do is repeatedly apply `parse` it is virtually impossible to make a mistake. Secondly, for the same reason it is exceptionally easy to generate such expressions via Template Haskell and similar metaprogramming mechanisms. Which is why a plethora of Hackage libraries use this approach.

In this paper we shall demonstrate a surprisingly simple technique that can be used to make computations expressing arbitrary transformations between simple data types of a single constructor while keeping the general form of `Applicative` expressions as they were shown above. Since we design our expressions to look similar to those produced with the help of `Applicative` type class but the underlying structure is not `Applicative` we shall call them "`Applicative`-like". Section 3 provides some motivating examples that show why we want to use `Applicative`-like computations to express transformations between data types. Section 4 formalizes the notion of "`Applicative`-like" and discusses the properties we expect from such expressions. Section 5 derives one particular structure for one of the motivating examples using LISP-encoding for deconstructing data types. Section 6 proceeds to derive the rest of motivating examples by applying the same idea, thus showing that section 5 describes a technique, not an isolated example. Section 6 ends by demonstrating the total expressive power of the technique. Section 7 repeats the derivation and the implementations for Scott-encoded data types. Section 8 observes the general structure behind all of the terms used in the paper. Section 9 gives a formal description of the technique and the underlying general algebraic structure. Section 10 refers to related work and wraps everything up.

3 Motivating examples

Consider the following expressions produced with the help of first author's favorite `safecopy` [12] data-type-to-binary serialization-deserialization library which can be used to deserialize-serialize `Device` with the following code snippet (simplified[3])

```
instance SafeCopy Device where
  getCopy = pure Device <*> getCopy <*> getCopy <*> getCopy

  putCopy (Device b x y) = putCopy b >> putCopy x >> putCopy y
```

Note that while `getCopy` definition above is trivial, `putCopy` definition binds variables. Would not it be better if we had an `Applicative`-like machinery with which we could rewrite `putCopy` into something like

```
putCopy = depure unDevice <**> putCopy <**> putCopy <**> putCopy
```

which, incidentally, would also allow us to generate both functions from a single expression? This idea does not feel like a big stretch of imagination for several reasons:

- there are libraries that can do both parsing and pretty printing using a single expression, e.g. [13],
- the general pattern of `putCopy` feels very similar to computations in `(->) a` (the type of "functions from `a`") as it, too, is a kind of computation in a context with a constant value, aka `Reader Monad` [14], which is an instance of `Applicative`.[4]

[2] Two or more constructors can be handled with the help of `Alternative` type class and some tagging of choices, but that is out of scope of this article.

[3] The actual working code for the actual library looks a bit more complex, but the `safecopy` library also provides Template Haskell functions that derive these `SafeCopy` instances automatically, so, in practice, one would not need to write this code by hand in any case.

[4] We shall utilize this fact in the following sections.

Another example is the data-type-to-JSON-to-strings serialization-deserialization part of `aeson` [15] library which gives the following class signatures to its deserializer and serializer from/to JSON respectively.

```
class FromJSON a where
  parseJSON :: Value -> Parser a

class ToJSON a where
  toJSON :: a -> Value
```

In the above, `Value` is a JSON value and `Parser a` is a Scott-transformed variation of `Either ErrorMessage a`. Assuming `(.:)` to be a syntax sugar for `lookup`-in-a-map-by-name function and `(.=)` a pair constructor, we can give the following instances for the `Device` data type by emulating examples given in the package's own documentation

```
instance FromJSON Device where
  parseJSON (Object v) = pure Device
                     <*> v .: "block"
                     <*> v .: "major"
                     <*> v .: "minor"
  parseJSON _          = empty

instance ToJSON Device where
  toJSON (Device b x y) = object
                       [ "block" .= b
                       , "major" .= x
                       , "minor" .= y ]
```

Note that here, again, we have to bind variables in `toJSON`. Moreover, note that in this example even `parseJSON` underuses the `Applicative` structure by ignoring the fact that `Value` can be packed into `Parser` by making the latter into a `Reader`.[5]

Other serialization-deserialization problems, e.g. conventional pretty-printing with the standard `Show` type class [5] are, of course, the instances of the same pattern, as we shall demonstrate in the following sections.

Finally, as a bit more involved example, imagine an application that benchmarks some other software applications on given inputs, records logs they produce and then computes per-application averages

```
data Benchmark a = Benchmark
  { firstApp :: a
  , firstLog :: String
  , secondApp :: a
  , secondLog :: String
  }

type Argv    = [String]
type Inputs  = Benchmark Argv
type Outputs = Benchmark Integer
type Avgs    = Benchmark Double

benchmark :: Inputs -> IO Outputs
average :: [ Outputs ] -> Avgs
```

[5] As noted under footnote 4 and demonstrated in detail in section 5. However, this underuse has a reasonable explanation for `aeson`: `Value`'s definition is *too structured* to have a conventional parser combinator library that can make this trick work in the general case (i.e. not just in the above example). This problem can be solved using indexed `Monad`ic parser combinators but that is out of scope of this article.

Assuming that we have aforementioned machinery for `SafeCopy` we can trivially autogenerate all of the needed glue code to deserialize `Inputs`, serialize `Outputs` and `Avgs`. The `benchmark` is the core of our application, so let us assume that it is not trivial to autogenerate and we have to write it by hand. We are now left with the "`average`" function. Let us assume that for the numeric parts of the `Outputs` type it is just a `fold` with point-wise sum over the list of `Outputs` followed by a point-wise divide by their `length` and for the `String` parts it simply point-wise concatenates all the logs.

Now, do we really want to write those binary operators completely by hand? Note that this `Benchmark` example was carefully crafted: it is not self- or mutually-recursive and, at the same time, it is also not particularly homogeneous as different fields require different operations. In other words, things like SYB [16], Uniplate [17], Multiplate [18] or Lenses [19, 20] are not particularly useful in this case.[6] Of course, in this particular example, it is possible to distill the computation pattern into something like

```
lift2B :: (a -> b -> c) -> (Benchmark a -> Benchmark b -> Benchmark c)
lift2B f (Benchmark a1 l1 a2 l2) (Benchmark b1 l3 b2 l4)
  = Benchmark (f a1 b1) (l1 ++ l3) (f a2 b2) (l2 ++ l4)
```

and then use `lift2B` to implement both functions (with some unsightly hackery for the division part), but would not it be even better if instead we had an `Applicative`-like machinery that would allow us to write the `average` function directly, such as

```
average ls = runMap $ bdivide folded where
  len = fromIntegral $ length ls
  avg = ((/ len) . fromIntegral)

  bappend = depureZip Benchmark unBenchmark unBenchmark
    `zipa` (+) `zipa` (++)
    `zipa` (+) `zipa` (++)

  folded = foldl' (\a b -> runZip $ bappend a b)
                  (Benchmark 0 "" 0 "") ls

  bdivide = depureMap Benchmark unBenchmark
    `mapa` avg `mapa` id
    `mapa` avg `mapa` id
```

similarly to how we would solve similar problems over homogeneous lists?

4 Problem definition

Before going into derivation of the actual implementation let us describe what we mean by "`Applicative`-like" more precisely.

Note that the type of `(<*>)` operator of `Applicative`

```
(<*>) :: f (a -> b) -> f a -> f b
```

at least in the context of constructing data types (of which `Applicative` parsers are a prime example), can be generalized and reinterpreted as

```
plug :: f full -> g piece -> f fullWithoutThePiece
```

where

[6] Strictly speaking, both operations used in the "sum" part of "`average`" are `Monoid` operators, so generalized `zip`s provided by some of the mentioned libraries can be used to implement that part, but the "divide" part is not so homogeneous.

- `f full` is a computation that *provides a mechanism* to handle the `full` structure,
- `g piece` is another kind of computation that *actually handles* a `piece` of the `full` structure (`g == f` for `Applicative` parsers, of course),
- and `f fullWithoutThePiece` is a computation that provided a mechanism to handle the leftover part.

Note that this interpretation, in some sense, reverses conventional wisdom on how such transformations are usually expressed.

For instance, conventionally, to parse (pretty-print, etc) some structure one first makes up computations that handle `piece`s and then composes them into a computation that handles the `full` structure, i.e.

```
compose  :: f fullWithoutThePiece -> g piece -> f full
-- or
compose' :: g piece -> f fullWithoutThePiece -> f full
```

Meanwhile, `Applicative`-like expressions, in some sense, work backwards: they provide up a mechanism to handle (parse, pretty-print, etc) the `full` structure that exposes "ports" that subcomputations `plug` with computations that handle different `piece`s.

Remark 1. *It is rather interesting to think about the conventional function application in these terms: it describes a way to make a computation that produces `b` given a mechanism to construct a partial version of `b` denoted as `a -> b` by plugging its only port with a computation that produces `a`. In other words, this outlook is a reminder that functions can be seen as goals, the same way Haskell's type class instance inference (or Prolog) does. Moreover, note that while such a description sounds obvious for a lazy language, it is also a reminder that, in general, there is a distinction between values and computations.*

To summarize, the crucial part of `Applicative`-like computations is the fact that they compose subcomputations in reverse order w.r.t. the types they handle. This reversal is the cornerstone that provides three important properties:

- A sequence of subcomputations in an expression matches the sequence of parts in the corresponding data type.
- A top-level computation can decide on all data types *first* and then delegate handing of parts to subcomputations without worrying about reassembling their results (which is why we say it "provides a mechanism" that subcomputations use).
- As a consequence, in the presence of type inference, a mechanism for ad-hoc polymorphism (be it type classes, like in Haskell, or something else) can be used to automatically select implementations matching corresponding `piece`s.

It is the combination of these three properties that makes `Applicative`-like expressions (including `Applicative` parsers) so convenient in practice.

5 Deriving the technique

We shall now demonstrate the derivation of the main technique of the paper. Before we start, let us encode reverses to `Device` and `Benchmark` constructors (i.e. "destructors") using the LISP-encoding (see below for motivation, an alternative approach using Scott-encoding is discussed in section 7).

```
unDeviceLISP :: Device -> (Bool, (Int, (Int, ())))
unDeviceLISP (Device b x y) = (b, (x, (y, ())))

unBenchmarkLISP :: Benchmark a -> (a, (String, (a, (String, ()))))
unBenchmarkLISP (Benchmark a b c d) = (a, (b, (c, (d, ()))))
```

Now, let us start by deriving an `Applicative`-like pretty-printer for `Device`. The target expression is as follows

```
showDevice = depureShow unDeviceLISP `showa` show
                                     `showa` show
                                     `showa` show
```

Remember that the type pattern for the `plug` operator from the previous section

```
plug :: f full -> g piece -> f fullWithoutThePiece
```

already prescribes a certain way of implementing the missing operators. Firstly, if we follow the logic for parsing, the `f` type-level function should construct a type that contains some internal state. Secondly, the rest of the expression clearly requires `depureShow` to generate the initial state and `showa` to transform the internal state while chopping away at the parts of the `Device`.

Let us simplify the task of deriving these functions by writing out the desired type and making `Device` argument explicit. Let us also apply the result of the whole computation to `runShow` function to lift the restriction on the return type.

```
showDevice' :: Device -> String
showDevice' d = runShow $ depureShow' (unDeviceLISP d) `showa'` show
                                                       `showa'` show
                                                       `showa'` show
```

What should be the type of `showa'`? Clearly, something like

```
showa' :: (s, (a, b)) -> (a -> String) -> (s, b)
```

should work and match the type pattern of `plug`. The `a -> String` part follows from the expression itself, the `(_ , (a, b))` and `(_ , b)` parts come from chopping away at LISP-encoded deconstructed data type, and `s` plays the role of the internal pretty-printing state. We just need to decide on the value of `s`. The most simple option seems to be to the list of `String`s that is to be concatenated in `runShow`. The rest of the code pretty much writes itself:

```
depureShow' :: a -> ([String], a)
depureShow' a = ([], a)

showa' :: ([String], (a, b)) -> (a -> String) -> ([String], b)
showa' (s, (a, b)) f = ((f a):s, b)

runShow :: ([String], b) -> String
runShow = concat . intersperse " " . reverse . fst

testShowDevice' :: String
testShowDevice' = showDevice' exampleDevice
```

Now, note that `showa'` is actually a particular case of the more generic operator

```
chop :: (s, (a, b)) -> (s -> a -> t) -> (t, b)
chop (s, (a, b)) f = (f s a, b)

showa'' s f = chop s (\s a -> (f a):s) -- == showa'
```

Moreover, `f` parts of that operator can be wrapped into the `(->) r Reader` (see under footnote 4)

```
chopR :: (r -> (s, (a, b))) -> (s -> a -> t) -> (r -> (t, b))
chopR o f r = chop (o r) f
```

thus allowing us to complete the original `showDevice`

```
showDevice :: Device -> ([String], ())

depureShow :: (t -> b) -> t -> ([String], b)
depureShow f r = ([], f r)

showa :: (r -> ([String], (a, b)))
      -> (a -> String)
      -> (r -> ([String], b))
showa st f = chopR st (\s a -> (f a):s)

testShowDevice :: String
testShowDevice = runShow $ showDevice exampleDevice
```

Note that the use of the LISP-encoding (i.e. the `()` in the tails of the deconstructed types and, hence, the use of `fst` in `runShow`) as opposed to using simple stacked tuples is needed to prevent special case handling for the last argument.

Also note that the type of the second argument to `chopR` in the definition of `showa` is `[String] -> a -> [String]` which is `CoState` on a list of `String`s. This makes a lot of sense categorically since `Parser` is a kind of `State` and parsing and pretty-printing are dual. Moreover, even the fact that `String` is wrapped into a list makes sense if one is to note that the above pretty-printer produces *lexemes* instead of directly producing the output string.

The above transformation from `chop` to `chopR` will be a common theme in the following sections, so let us distill it into a separate operator with a very self-descriptive type

```
homWrap :: (s -> a -> t)
        -> (r -> s) -> a -> (r -> t)
homWrap chopper o f r = chopper (o r) f

showa''' = homWrap $ \st f -> chop st $ \s a -> (f a):s -- == showa
```

6 Applying the technique

Turning attention back to `chop` operator, note that both types in the state tuple can be arbitrary. For instance, `s` can be a curried data type constructor, which immediately allows to express an `Applicative`-like step-by-step equivalent of `map`.

```
mapa :: (r -> (x -> y, (a, b)))
     -> (a -> x)
     -> (r -> (y, b))
mapa = homWrap $ \st f -> chop st $ \s a -> s (f a)

depureMap :: a -> (t -> b) -> t -> (a, b)
depureMap c f r = (c, f r)

runMap = fst

mapDevice :: Device -> (Device, ())
mapDevice = depureMap Device unDeviceLISP
  `mapa` not
  `mapa` (+ 100)
  `mapa` (+ 200)
```

```
testMapDevice :: Device
testMapDevice = runMap $ mapDevice exampleDevice
```

Moreover, by extending `chop` with two LISP-encoded representations and repeating the whole derivation we can express an equivalent of `zip`.

```
chop2 :: (s, (a, b), (c, d))
      -> (s -> a -> c -> t)
      -> (t, b, d)
chop2 (s, (a, b), (c, d)) f = (f s a c, b, d)

homWrap2 chopper o f ra rb = chopper (o ra rb) f

zipa :: (ra -> rb -> (x -> y, (a, b), (c, d)))
     -> (a -> c -> x)
     -> (ra -> rb -> (y, b, d))
zipa = homWrap2 $ \st f -> chop2 st $ \s a b -> s (f a b)

depureZip :: a -> (ra -> b) -> (rb -> c)
          -> ra -> rb
          -> (a, b, c)
depureZip c f g ra rb = (c, f ra, g rb)

runZip :: (s, a, b) -> s
runZip (s, _, _) = s

zipDevice :: Device -> Device -> (Device, (), ())
zipDevice = depureZip Device unDeviceLISP unDeviceLISP
  `zipa` (&&)
  `zipa` (+)
  `zipa` (+)

testZipDevice :: Device
testZipDevice = runZip $ zipDevice exampleDevice testMapDevice
```

The above transformations combined with

```
unDevice = unDeviceLISP
unBenchmark = unBenchmarkLISP
```

implement all the examples from section 3, thus solving the problem as it was originally described.

Note, however, that the above technique can be trivially extended to `chop`ping any number of data types at the same time and, moreover, that it is not actually required to match types or even the numbers of arguments of different constructors and destructors used by the desired transformations. For instance, it is trivial to implement the usual stack machine operators, e.g.

```
homWrap0 :: (s -> t)
         -> (r -> s) -> (r -> t)
homWrap0 chopper o r = chopper (o r)

-- syntax sugar
andThen x f = f x

pop :: (r -> (s, (a, b)))
    -> (r -> (s, b))
```

```
pop = homWrap0 $ \(s, (_, b)) -> (s, b)

push = homWrap $ \(s, b) a -> (s, (a, b))

dup = homWrap0 $ \(s, (a, b)) -> (s, (a, (a, b)))
```

and use them to express some mapping function between data types as if Haskell was a stack machine language

```
remapDevice :: Device -> (Device, ())
remapDevice = depureMap Device unDeviceLISP
  `andThen` pop
  `push` True
  `mapa` id
  `andThen` pop
  `andThen` dup
  `mapa` id
  `mapa` id

testRemapDevice :: Device
testRemapDevice = runMap $ remapDevice exampleDevice
```

In other words, in general, one can view `Applicative`-like computations as computations for generalized multi-stack machines with arbitrary data types and/or functions as "stacks".

In practice, though, simple direct transformations in the style of `Applicative` parsers seem to be the most useful to us.

7 Scott-encoded representation

The LISP-encoding used above is not the only generic representation for data types, in this section we shall explore the use of Scott-encoding.

Before we start, let us note that while it is trivial to simply Scott-encode all the pair constructors and destructors in the above transformations to get more complicated terms with exactly equivalent semantics [8], it just complicates things structurally, and we shall not explore that route.

The interesting question is whether it is possible to remake the above machinery directly for Scott-encoded representations of the subject data types

```
unDeviceScott :: Device -> (Bool -> Int -> Int -> c) -> c
unDeviceScott (Device b x y) f = f b x y

unBenchmarkScott :: Benchmark a
                 -> (a -> String -> a -> String -> c) -> c
unBenchmarkScott (Benchmark a b c d) f = f a b c d
```

without reaching for anything else. In other words, would not it be nice if we could work with a Scott-encoded data type `(a -> b -> c -> ... -> z) -> z` as if it was a heterogeneous list of typed values like LISP-encoding is?

Let us start by noticing that we can, in fact, prepend values to Scott-encoded representations as if they were heterogeneous lists or tuples

```
consS :: s
      -> (a -> b)
      -> ((s -> a) -> b)
consS s ab sa = ab (sa s)
```

To see why this prepends `s` to a Scott-encoded `a -> b` substitute, for instance, `x -> y -> b` for `a`. Note, however, that there are some important differences. For instance, Scott-encoded data types, unlike LISP-encoded ones, can not have a generic `unconsS`

```
unconsS :: ((s -> a) -> b) -> (s, a -> b)
unconsS f = (_, _)
```

as, in general, all the pieces of a Scott-encoded data type have to be used all at once. This makes most of our previous derivations unusable. However, very surprisingly, `consS` seems to be enough.

By prepending `s` to the Scott-encoded data type we can emulate pretty-printing code above as follows.[7]

```
chopS :: ((s -> a -> b) -> c)
      -> (s -> a -> t)
      -> ((t -> b) -> c)
chopS i f o = i $ \s a -> o (f s a)

depureShowS f r = consS [] (f r)

showaS :: (r -> ([String] -> a -> b) -> c)
       -> (a -> String)
       -> (r -> ([String] -> b) -> c)
showaS = homWrap $ \st f -> chopS st $ \s a -> (f a):s

runShowS = concat . intersperse " " . reverse . (\f -> f id)

showDeviceS = depureShowS unDeviceScott
  `showaS` show
  `showaS` show
  `showaS` show

testShowDeviceS = runShowS $ showDeviceS exampleDevice
```

The only new parts here are the implementation of `chopS` function, the use of `consS` instead of the pair constructor, and the replacement of `fst` with `\f -> f id`. The rest is produced mechanically by adding `S` suffix to all function calls. The `map` example can be similarly mechanically translated as follows.

```
mapaS :: (r -> ((x -> y) -> a -> b) -> c)
      -> (a -> x)
      -> (r -> (y -> b) -> c)
mapaS = homWrap $ \st f -> chopS st $ \s a -> s (f a)

depureMapS c f r = consS c (f r)

runMapS f = f id

mapDeviceS = depureMapS Device unDeviceScott
  `mapaS` not
  `mapaS` (+ 100)
  `mapaS` (+ 200)

testMapDeviceS :: Device
testMapDeviceS = runMapS $ mapDeviceS exampleDevice
```

[7] We tried our best to make this comprehensible by making the types speak for themselves but, arguably, this and the following listings can only be really understood by playing with the Literate Haskell version in `ghci`.

The most interesting part, however, is the reimplementation of `zip`. By following the terms in the previous section we would arrive at the following translation for `depureZip`

```
depureZipS' :: s -> (ra -> a) -> (rb -> b -> c)
            -> ra -> rb
            -> (s -> a -> b) -> c
depureZipS' c f g r s = consS c (consS (f r) (g s))
```

Frustratingly, there is no `chop2` equivalent for it

```
chop2S' :: ((s -> ((a -> b) -> c) -> d -> e) -> f)
        -> (s -> a -> d -> t)
        -> (t -> (b -> c) -> e) -> f
chop2S' i f o = i $ \s abq d -> o _ _
```

because `a` becomes effectively inaccessible in this order of `consS`ing (as there is no `unconsS`). However, fascinatingly, by simply changing that order to

```
depureZipS c f g r s = consS (consS c (f r)) (g s)
```

we get our `cons2S` and, by mechanical translation, all the rest of `zipDevice` example

```
chop2S :: ((((s -> a -> b) -> c) -> d -> e) -> f)
       -> (s -> a -> d -> t)
       -> (((t -> b) -> c) -> e) -> f
chop2S i f o = i $ \sabc d -> o $ \tb -> sabc $ \s a -> tb $ f s a d

zipaS :: (ra -> rb -> (((((x -> y) -> a -> b) -> c) -> d -> e) -> f))
      -> (a -> d -> x)
      -> (ra -> rb -> (((y -> b) -> c) -> e) -> f)
zipaS = homWrap2 $ \st f -> chop2S st $ \s a b -> s (f a b)

runZipS f = f id id

zipDeviceS = depureZipS Device unDeviceScott unDeviceScott
  `zipaS` (&&)
  `zipaS` (+)
  `zipaS` (+)

testZipDeviceS :: Device
testZipDeviceS = runZipS $ zipDeviceS exampleDevice testMapDeviceS
```

thus, again, implementing all the examples from section 3, but now purely with Scott-encoded data types.

Remark 2. *Note that while the transformation form `b` to `(a, b)` for the LISP-encoding or the plain tuples is regular, the transformation from `(a -> b -> c -> ... -> z) -> z` to `(s -> a -> b -> c -> ... -> z) -> z` is not, the former is not a sub-expression of the latter. Taking that into account, we feel that the very fact that the implementations demonstrated above are even possible is rather fascinating. The fact that Scott-encoding can be used as a heterogeneous list is rather surprising as even the fact that `consS` is possible is rather weird, not to mention the fact that useful things can be done without `unconsS`. We are not aware of any literature that describes doing anything similar directly to Scott-encoded data types.*

8 General Case

Curiously, note that with the aforementioned order of consSing chop2S is actually a special case of chopS

```
chop2S' :: ((((s -> a -> b) -> c) -> d -> e) -> f)
        -> (s -> a -> d -> t)
        -> (((t -> b) -> c) -> e) -> f
chop2S' i f o = chopS i (\sabc d tb -> sabc $ \s a -> tb $ f s a d) o
  -- == chop2S
```

and this pattern continues when consSing more structures

```
depureZip3S :: s -> (ra -> a -> b) -> (rb -> c -> d) -> (rc -> e -> f)
            -> ra -> rb -> rc
            -> (((((s -> a) -> b) -> c) -> d) -> e) -> f
depureZip3S c f g h r s t = consS (consS (consS c (f r)) (g s)) (h t)

chop3S :: ((((((s -> a -> b) -> c) -> d -> e) -> f) -> g -> h) -> i)
       -> (s -> a -> d -> g -> t)
       -> (((((t -> b) -> c) -> e) -> f) -> h) -> i
chop3S i f o = chop2S i (\sabc d g tb -> sabc $ \s a -> tb $ f s a d g) o

-- and so on
```

The same is true for LISP-encoded variant since we can use the same order of consing there, e.g.

```
chop2' :: ((s, (a, b)), (c, d))
       -> (s -> a -> c -> t)
       -> ((t, b), d)
chop2' (sab, (c, d)) f = (chop sab (\s a -> f s a c), d)
  -- ~~ chop2
```

but we think this presentation makes things look more complex there, not less. Though, as we shall see in the next section (in its Literal Haskell version), we could have simplified the general case by using chop2' above.

In other words, if we are to cons LISP-encoded and consS Scott-encoded data types in the right order then all of the Applicative-like operators of this paper and the generalizations of Applicative-like zips to larger numbers of structures can be uniformly produced from just chop and chopS.

9 Formal Account

The derivation of section 5, as demonstrated by the following sections, describes a technique (as opposed to an isolated example) for expressing transformations between simple data types of a single constructor using Applicative-like computations. More formally, that technique consists of

- deconstructing the data type (into its LISP-encoded representation in sections 5 and 6 or Scott-encoded representation in section 7),
- wrapping the deconstructed representation into the Applicative-like structure in question with an operation analogous to Applicative's pure (depureShow, etc),
- followed by spelling out transformation steps to the desired representation by interspersing them with an operator analogous to Applicative's (<*>) (showa, mapa, zipa, etc),

- followed by wrapping the whole structure into `(->) r Reader` that is used to propagate the input argument to the front of the expression without adding explicit argument bindings to the whole expressions.

Note, however, that the last "wrapping" bit of the translation is orthogonal to the rest. It is needed to produce a completely variable-binding-less expression, but that step can be skipped if variable-binding-lessness is not desired: one simply needs to remove the `homWrap` wrapping, add an explicitly bound argument to the function, and then apply it to `depureShow`.

Also remember that section 6 showed that, in general, those expressions can implement any computations for generalized multi-stack machines with arbitrary data types and/or functions as "stacks". For the `show`-, `map`-, and `zip`-like transformations we described in detail, however, the central `chop` operator corresponds to a simple state transformer of the corresponding "step-by-step" `fold`, if we are to view the deconstructed data type as a heterogeneous list.

Finally, note that while `depureMap` and `depureZip` (`depureMapS` and `depureZipS`) take more arguments than `Applicative`'s `pure` this fact is actually inconsequential as in section 8 we noted that we can simply reorganize all our expressions to `cons` to the left (as we had to do for Scott-encoded data types). Thus, only the last argument to the `depure*` functions is of any consequence to the general structure (since it is the argument we are `fold`ing on, inductively speaking), the rest are simply baggage used internally by the corresponding operators.

9.1 Dependently-typed Applicative

Now, the obvious question is how a general structure unifying all those operators would look. Firstly, let us note that the `pure` function of `Applicative` can be separated out into its own type class

```
class Pointed f where
  pure :: a -> f a

infixl 4 <*>
class (Pointed f, Functor f) => Applicative f where
  (<*>) :: f (a -> b) -> f a -> f b
```

Moreover, note that, algebraically speaking, `Applicative` depends on `Pointed` only because their combination gives `Functor`, they are independent otherwise. Since we have no equivalent for `Functor` with `Applicative`-like expressions we can discuss these two parts separately.

Secondly, let us note that `Control.Category` and `Control.Arrow` modules of `base` [5] define `Category` [5] and `ArrowApply` [21] type classes as

```
class Category cat where
  id :: cat a a
  (.) :: cat b c -> cat a b -> cat a c

class Arrow a => ArrowApply a where
  app :: a (a b c, b) c
```

respectively. Both of these type classes denote generalized functions over generalized function types: `cat` and `a` respectively.

Thirdly, if we are to look at the types of our `showa`, `mapa`, and `zipa` operators and their versions for Scott-encoded data types, the most glaring difference from the type of `(<*>)` we will notice is the fact that the types of their second arguments and the types of their results depend on the types of their first arguments (or, equivalently, we can say that all of those depend on another implicit type argument). In other words, if `(<*>)` and `app` are two generalizations of the conventional function application, then the structure that describes our operators is a generalization of the dependently typed function application.

The simplest general encoding we have for our examples for GHC Haskell (with awful lot of extensions) looks like this

```haskell
class ApplicativeLike f where
  type C f a b :: * -- type of arrow under `f`
  type G f a :: *   -- type of argument dependent on `f`
  type F f b :: *   -- type of result dependent on `f`
  (<**>) :: f (C f a b) -> G f a -> F f b

newtype Mapper r f a = Mapper { runMapper :: r -> (f, a) }

instance ApplicativeLike (Mapper e (x -> y)) where
  type C (Mapper e (x -> y)) a b = (a, b)
  type G (Mapper e (x -> y)) a = a -> x
  type F (Mapper e (x -> y)) b = Mapper e y b
  f <**> g = Mapper $ mapa (runMapper f) g

mapDeviceG :: Mapper Device Device ()
mapDeviceG = Mapper (depureMap Device unDeviceLISP)
  <**> not
  <**> (+ 100)
  <**> (+ 200)

testMapDeviceG :: Device
testMapDeviceG = runMap $ runMapper mapDeviceG exampleDevice

newtype MapperS c r f a = MapperS
  { runMapperS :: r -> (f -> a) -> c }

instance ApplicativeLike (MapperS c e (x -> y)) where
  type C (MapperS c e (x -> y)) a b = a -> b
  type G (MapperS c e (x -> y)) a = a -> x
  type F (MapperS c e (x -> y)) b = MapperS c e y b
  f <**> g = MapperS $ mapaS (runMapperS f) g

mapDeviceGS :: MapperS c Device Device c
mapDeviceGS = MapperS (depureMapS Device unDeviceScott)
  <**> not
  <**> (+ 100)
  <**> (+ 200)

testMapDeviceGS :: Device
testMapDeviceGS = runMapS $ runMapperS mapDeviceGS exampleDevice

-- See Literate Haskell version for many more examples.
```

The operator analogous to `pure` simply wraps the result produced by the data type destructor into the corresponding initial state, thus its generalization is not interesting (in general, it is a function `a -> f b`). Moreover, generalizing it actually adds problems because a generic `depure` makes `(<**>)` ambitious in

```haskell
ambitiousExample a = depure unDevice <**> a <**> a <**> a
```

This does not happen for `Applicative` type class since both arguments to `(<*>)` are of the same type family `f` there.

10 Conclusion

From a practical perspective, in this article we have shown that by implementing a series of rather trivial state transformers we called `chop*` and wrappers into a `(->) r Reader` we called `homWrap*` and then composing them one can express operators that can implement arbitrary computations for generalized multi-stack machines using a rather curious form of expressions very similar to conventional `Applicative` parsers. Then, we demonstrated how to use those operators to implement `Applicative`-like pretty-printers, `map`s, and `zip`s between simple data types of a single constructor by first unfolding them into LISP- and Scott-encoded representations and then folding them back with custom "step-by-step" `fold`s. (Where the very fact that Scott-encoded case is even possible is rather fascinating as those terms are constructed using a rather unorthodox technique.)

Remark 3. *By the way, note that Haskell's `GHC.Generics` [22] is not an adequate replacement for LISP- and Scott-encoded representations used in the paper: not only is the `Rep` type family complex, its structure is not even deterministic as GHC tries to keep the resulting type representation tree balanced. Which, practically speaking, suggests another GHC extension.*

From a theoretical perspective, in this article we have presented a natural generalization of the conventional `Applicative`[1] type class (which can be viewed as a generalization of conventional function application) into dependent types with generalized arrow of `Category`/`ArrowApply` [5, 21]. Both `Applicative`s and `Monad`s [23–25] (that can be viewed as a generalization of the conventional sequential composition of actions, aka "imperative semicolon") were similarly generalized to super-applicatives and supermonads in [26]. In particular, [26] starts by giving the following definition for `Applicative`

```
class Applicative m n p where
  (<*>) :: m (a -> b) -> n a -> p b
```

then adds constraints on top to make the type inference work, and then requires all of `m`, `n`, and `p` to be `Functor`s (producing such a long and scary type class signature as the result so that we decided against including it here). In contrast, our `ApplicativeLike` generalizes the arrow under `m`, goes straight to dependent types for `n` and `p` instead of ad-hoc constraints, and doesn't constrain them in any other way.

Remark 4. *Which suggests syntactic (rather than algebraic) treatment of `ApplicativeLike` structure as it seems that there are no new interesting laws about it except for those that are true for the conventional function application (e.g., congruence `a == b => f a == f b`).*

In other words, our `ApplicativeLike` can be viewed as a simpler encoding for generalized super-applicatives of [26] when those are treated syntactically rather than algebraically (since we completely ignore `Functor`s).

Future fork on the subject consists of applying the same ideas to `Alternative` type class to cover the multi-constructor case, which is not clear at the moment since it is not exactly clear how the canonical use of `Alternative` for parsing tagged data types should look like in the first place, as, unlike the `Applicative` case, different libraries use different idioms for this.

References

[1] Conor McBride and Ross Paterson. "Applicative Programming with Effects". In: *Journal of Functional Programming* 18.1 (2008), pp. 1–13. URL: http://www.soi.city.ac.uk/~ross/papers/Applicative.pdf (cit. on pp. 2, 16).

[2] Carsten Dominik, Eric Schulte, Nicolas Goaziou, et al. *Org mode for Emacs*. 2018. URL: https://orgmode.org/ (cit. on p. 2).

[3] Eric Schulte et al. "A Multi-Language Computing Environment for Literate Programming and Reproducible Research". In: *Journal of Statistical Software* 46.3 (Jan. 2012), pp. 1–24 (cit. on p. 2).

[4] GHC Project Authors. *GHC: The Glasgow Haskell Compiler*. 2018. URL: https://www.haskell.org/ghc/ (cit. on p. 2).

[5] *Hackage: The base package, version 4.9.0.0*. 2016. URL: https://hackage.haskell.org/package/base-4.9.0.0 (cit. on pp. 2, 4, 14, 16).

[6] *Hackage: Haskell Central Package Archive*. URL: https://hackage.haskell.org/ (cit. on p. 2).

[7] Haskell Wiki Authors. *Typeclassopedia*. URL: https://wiki.haskell.org/Typeclassopedia (cit. on p. 2).

[8] Jan Malakhovski. "Exceptionally Monadic Error Handling". In: *Journal of Functional Programming* (2018). Under consideration, submitted January 2019. arXiv: 1810.13430. URL: https://oxij.org/paper/ExceptionallyMonadic/ (cit. on pp. 2, 10).

[9] Daan Leijen, Paolo Martini, and Antoine Latter. *Hackage: The Parsec package, version 3.1.11*. 2016. URL: https://hackage.haskell.org/package/parsec-3.1.11 (cit. on p. 2).

[10] Bryan O'Sullivan. *Hackage: The Attoparsec package, version 0.13.1.0*. 2016. URL: https://hackage.haskell.org/package/attoparsec-0.13.1.0 (cit. on p. 2).

[11] Mark Karpov, Paolo Martini, Daan Leijen, et al. *Hackage: The megaparsec package, version 6.3.0*. 2017. URL: https://hackage.haskell.org/package/megaparsec-6.3.0 (cit. on p. 2).

[12] David Himmelstrup, Felipe Lessa, et al. *Hackage: The safecopy package, version 0.9.4.3*. 2018. URL: https://hackage.haskell.org/package/safecopy-0.9.4.3 (cit. on p. 3).

[13] Paweł Nowak. *Hackage: The syntax package, version 1.0.0.0*. 2014. URL: https://hackage.haskell.org/package/syntax-1.0.0.0 (cit. on p. 3).

[14] Andy Gill and Ross Paterson. *Hackage: The transformers package, version 0.5.2.0*. 2016. URL: https://hackage.haskell.org/package/transformers-0.5.2.0 (cit. on p. 3).

[15] Bryan O'Sullivan, Adam Bergmark, et al. *Hackage: The aeson package, version 1.4.2.0*. 2018. URL: https://hackage.haskell.org/package/aeson-1.4.2.0 (cit. on p. 4).

[16] Ralf Lämmel and Simon Peyton Jones. "Scrap your boilerplate: a practical approach to generic programming". In: ACM Press, Jan. 2003, pp. 26–37. URL: https://www.microsoft.com/en-us/research/publication/scrap-your-boilerplate-a-practical-approach-to-generic-programming/ (cit. on p. 5).

[17] Neil Mitchell and Colin Runciman. *Uniplate*. 2007. URL: http://community.haskell.org/~ndm/uniplate/ (cit. on p. 5).

[18] Russell O'Connor. *Hackage: The multiplate package, version 0.0.3*. 2015. URL: https://hackage.haskell.org/package/multiplate-0.0.3 (cit. on p. 5).

[19] Edward Kmett. *Lenses, Folds and Traversals*. 2013. URL: http://lens.github.io/ (cit. on p. 5).

[20] Edward Kmett et al. *Hackage: The lens package, version 4.17*. 2018. URL: https://hackage.haskell.org/package/lens-4.17 (cit. on p. 5).

[21] John Hughes. "Generalising monads to arrows". In: *Science of Computer Programming* 37.1–3 (2000), pp. 67–111. URL: http://www.cse.chalmers.se/~rjmh/Papers/arrows.pdf (cit. on pp. 14, 16).

[22] *GHC 8.6.3: The base package, version 4.12.0.0: GHC.Generics*. 2018. URL: https://downloads.haskell.org/~ghc/8.6.3/docs/html/libraries/base-4.12.0.0/GHC-Generics.html (cit. on p. 16).

[23] Eugenio Moggi. "Computational λ-Calculus and Monads". In: *Logic in Computer Science (LICS)*. June 1989, pp. 14–23. URL: http://www.disi.unige.it/person/MoggiE/ftp/lics89.ps.gz (cit. on p. 16).

[24] Eugenio Moggi. "Notions of computation and monads". In: *Information and Computation* 93.1 (1991). URL: http://www.disi.unige.it/person/MoggiE/ftp/ic91.pdf (cit. on p. 16).

[25] Philip Wadler. "The essence of functional programming". In: *Conference record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Albuquerque, New Mexico, January 19–22, 1992*. Ed. by ACM. ACM order number 54990. New York, NY, USA: ACM Press, 1992, pp. 1–14. ISBN: 0-89791-453-8 (cit. on p. 16).

[26] Jan Bracker and Henrik Nilsson. "Supermonads and superapplicatives". In: *Journal of Functional Programming* (2018). Under consideration, submitted 12 December 2017 (cit. on p. 16).