A Summary of "On the Expressive Power of Indexed Applicative and Monadic Structures"

Jan Malakhovski*

IRIT, University of Toulouse-3 and ITMO University

Version: 1.1; October 25, 2019

Abstract

It is well-known that very simple theoretic constructs such as `Either` (type-theoretic equivalent of the logical "or" operator), `State` (composable state transformers), `Applicative` (generalized function application), and `Monad` (generalized sequential program composition) structures (as they are named in Haskell) cover a huge chunk of what is usually needed to elegantly express most computational idioms used in conventional programs. However, it is conventionally argued that there are several classes of commonly used idioms that do not fit well within those structures, the most notable examples being transformations between trees (data types, which are usually argued to require ether generalized pattern matching or heavy metaprogramming infrastructure) and exception handling (which are usually argued to require special language and run-time support).

This work aims to show that many of those idioms can, in fact, be expressed by reusing those well-known structures with minor (if any) modifications. In other words, the purpose of this work is to apply the KISS (Keep It Stupid Simple) and/or Occam's razor principles to algebraic structures used to solve common programming problems.

Technically speaking, this work aims to show that natural generalizations of `Applicative` and `Monad` type classes of Haskell combined with the ability to make Cartesian products of them produce a very simple common framework for expressing many practically useful things, some of the instances of which are very convenient novel ways to express common programming ideas, while others are usually classified as effect systems. On that latter point, if one is to generalize the presented instances into an approach to design of effect systems in general, then the overall structure of such an approach can be thought of as being an almost syntactic framework which allows different effect systems adhering to the general structure of the "marriage" framework [28] to be expressed on top of. (Though, this work does not go into too much into the latter, since this work is mainly motivated by examples that can be immediately applied to Haskell practice.)

Note, however, that, after the fact, these technical observation are completely unsurprising: `Applicative` and `Monad` are generalizations of functional and linear program compositions respectively, so, naturally, Cartesian products of these two structures ought to cover a lot of what programs usually do.

1 Introduction

First programmable computers like Colossus (1943-1945) and even the early revisions of ENIAC (1945-1955) were not stored-program computers and could only be programmed using plugboards and mechanical switches.

IBM 650 (1953-1969), the first mass-produced computer, used a magnetic drum as its memory (usually initially loaded from punch-cards) and each instruction's operation code (opcode) had to explicitly specify the address of the next instruction (similarly to how `jump` instructions of modern Assembly languages do).

*papers@oxij.org; preferably with "thesis" as a substring of the subject line

The first computer with proper hardware support for subroutine calls of non-fixed-level nesting depth (that is, supporting recursion and arbitrary modularity) seems to be the PDP-11 (1970-1990), even though the support for simple subroutines was present even on the early ENIAC.

What these early examples show is that the very concept of a mostly linear program organized using modular possibly recursive subroutines had no hardware support until 1970s. Most interestingly, however, as soon as those things got hardware support, the UNIX and the C programming language [6, 7, 22] were born. Both mostly equivalent hardware and those software systems are still ubiquitous even today.

(One could argue that the only big change in the commonly employed computer architecture since 1970s is the popularization of SIMD for numerical computations. That is, the fact that almost all consumer-facing computers now come with GPUs out-of-the box. There is also a revival of hardware virtualization, first introduced on IBM System/370 in 1972 and then forgotten until mid-2000s, but both hardware support for arbitrarily nested virtualization and software use of those features, a good contemporary example of which would be QubesOS [21], are still rather lacking at the moment of writing of this work.)

The history of high-level programming languages starts with FORTRAN initially developed by John Backus for IBM 704 around 1956 and LISP initially developed by John McCarthy at MIT around the same time.

FORTRAN family of imperative compiled strictly-typed languages, including ALGOL, C and their descendants can be viewed as, at first, straightforward attempts to make a universal Assembly language, with later horizontal-gene-transfer/incorporation of structured programming constructs such as `if-then-else` statements, loops (both FORTRAN 77), arrays, modules (both Fortran 90, the later is also C++20), sometimes mixed with some object-oriented constructs from Simula (of which C++ is the prime example), and, after 50-odd years, ideas from functional programming (C++11 and later).

LISP family of functional interpreted dynamically-typed languages, by contrast, was going the other direction by starting from λ-calculus developed by Alonzo Church and his students in 1930s and 1940s with the explicit goal of making a minimalist universal computational formalism [2, 3] and building on top. For the purposes of this discussion two most important features of LISP were the ability to *declare* new language constructs using so called "special forms" (which were, effectively, partially lazy functions in an language with eager evaluation) and the ability to describe its own programs (reflection). The latter property meant that runtime code generation and meta-programming were easy, and, even more importantly, the language could trivially interpret itself, thus allowing arbitrary extensions. The end result is that most variants of LISP to this day can evaluate each other's terms.

Various mixes of the two approaches appeared over the years. Two noteworthy families are

- imperative (usually) interpreted dynamically-typed languages starting with Smalltalk and represented by modern Python, Ruby, JavaScript, among others; and
- functional (usually) compiled statically-typed languages starting with ML and represented by modern OCaml, SML, and Haskell, among others.

Among those, the sequence of languages LISP $\rightarrow$ ML $\rightarrow$ Miranda $\rightarrow$ Haskell is rather interesting because the step from LISP to ML replaced dynamic typing with a polymorphic type system and infix syntax at the cost of loosing both special forms and reflection, the step to Miranda switched to lazy evaluation by default (thus giving most of what special forms did), and the step to Haskell added type classes (thus giving a lot of what dynamic types did) and reintroduced reflection, among many other things.

In other words, Haskell was designed to conveniently express things commonly discussed in Programming Languages Theory (PLT) as its terms look similar to those used in school-level mathematics, strictly-typedness allows (but not guarantees) it to be efficient, and it has enough pieces of LISP and more powerful type systems (like dependent types) to express (or at least hint at how they could be expressed) concepts applicable to whole swaths of programming languages. And indeed, most of the literature cited in this work uses Haskell or a variant of ML.

Haskell is also surprisingly popular for an "academic" language consistently staying in Top-50 of TIOBE Index [26] (measures search requests), with its the most popular public code repository of Hackage [5] listing over 5000 packages.

As a side note, the usual way to explain why imperative languages (like FORTRAN, ALGOL, C) "won" over LISP is to note that the latter required too many transistors to evaluate at agreeable speeds. Where FORTRAN emitted a single Assembly `add`, LISP-machine needed a whole lot of run-time type checking. Then, the resurgence of popularity of Smalltalk descendants like Python, Ruby, JavaScript in late 1990s and early 2000s can be explained by, on the one hand, their general semantic similarity to FORTRAN descendants but with higher levels of programmer satisfaction (simpler syntax without explicit type signatures, automatic memory management, etc), and, on the other hand, the rise of the number of transistors available on an average consumer CPU, followed by the advent of just-in-time (JIT) compilation. Though, note that most high-performance code for systems written in those languages is still implemented in C and FORTRAN to be called by said interpreters via foreign function interface (FFI). For instance, NumPy [19], a Python library for high-performance numerical computations (and probably *the* most well-known Python library in academic circles), is a Pythonic wrapper over a bunch of C (and some FORTRAN, translated into C) code.

The resurgence of interest in the functional programming in the later half of 2000s, on the other hand, comes with the advent of compilation techniques which made them usable in high-performance software systems. Among other things, this allows some of those languages to produce complete or almost complete full-stack mono-language systems. For instance, MirageOS project [13], a modular operating system written entirely in ML. Similarly, Go [24], Haskell [4], and Rust [25] standard libraries also try to limit their use of FFIs. Which, of course, can be seen as either a good thing ("Yay! Readable code in a sane safe language!") when compared to languages that use a lot of C FFIs in their standard libraries (e.g. Python) or a bad thing ("Uhg! Now every language infrastructure reimplements everything from scratch!").

Note, however, that conventional CPUs are, essentially, interpreters for machine code (sequences of opcodes) compiled into hardware (the metal traces and semiconductor gates of which are then "interpreted" by the physical laws of electromagnetism). Which is why languages that are closer to Assembly are easier to compile in such a way that semantically efficient source language programs are compiled into opcode programs that are efficient to evaluate on those machines. GPUs discussed above, first marketed as "graphical accelerators", are now considered an essential piece of modern computing machinery, making modern image rendering and processing techniques, among other things, practically viable. Therefore, it would be interesting to see software systems developed specifically for computers with "FPGA accelerators", since graph reductions performed by interpreters of functional programming languages can be made much more efficient on such machines (e.g., see Reduceron [17, 18] project).

That is to say, it is not entirely obvious that FORTRAN descendants would still be "winning" on the computer systems running in the not so far future, as programs for computers with reversible computations (like raw electromagnetism and quantum computers) are very much functional [1, 23], thus it might be both more efficient and cognitively simpler to implement those systems in functional languages from top to bottom.

In any case, this work deals with somewhat more conventional computations. The main algebraic structures discussed in this work are `Monads` introduced to functional programming from Category theory by Moggi [15, 16] and popularized by Wadler [27] and `Applicative Functor`s introduced by McBride and Paterson [12]. These two structures can be seen as a straightforward generalizations of linear and functional program compositions respectively, that is, generalizations of the "semicolon" and "function call" operators.

2 Extended Abstract

If one is to ask a practicing Haskell programmer to succinctly describe `Applicative` and `Monad` type classes to a practicing programmer in an imperative language, something like "an overloadable function application/call operator" and "an overloadable semicolon operator" would probably be heard.

These structures are useful for a couple of reasons.

- Firstly, using generic operators reduces boilerplate somewhat by allowing for generic combinators (e.g. `mapM`).
- Secondly, and more importantly, those structures provide a convenient level of abstraction that hides irrelevant details (of which `Either Monad` that hides the `Left` half of the computation until it becomes relevant is a prime example).

Think `call` and `ret` operators of most conventional assembly languages, a programmer in CPU microcode (or sufficiently RISC assembly) might ask why do you even need those instructions when you can just `push`/`pop` the instruction pointer and `jump`. Similarly, a programmer for IBM 650 might argue that even linear sequencing of instructions and the instruction pointer are superfluous, each instruction could just explicitly specify the address of the next instruction. Similarly, for `Applicative` and `Monad`, while one could just use particular `(<*>)` and `(>>=)` implementations explicitly, having those operators to represent an even higher level of abstraction can be even more convenient. (Though, it can be problematic to show that convenience to a programmer in a language lacking the means to express it, like with `Either Monad`.)

Interestingly however, after explaining why `Applicative` and `Monad` are useful and pointing that they are indeed very popular in Haskell programs one will be faced with the fact that, apparently, there are not many commonly applicable instances of these structures. In fact, just `Either` and `State` together seem to cover almost everything:

- computations that might fail usually wrap themselves into `Either`,
- a `main` function in a Haskell program, more or less, simply interprets a `State` transformer over a `RealdWorld` that computes program outputs from program inputs (i.e. `IO Monad`, though it can have other interpretations),
- most other things are either particular cases (e.g. `Maybe`), compositions of those two (parsing, for instance, is just a composition of `State` and `Either` with `Stream`s in place of the `RealdWorld`), or mechanical transformations (e.g. Scott-encoding) of them.

The fact that `Either` and `State` `Applicative`s and `Monad`s can express so much makes it even more interesting to carefully look at the frequently used things they, apparently, *can not* express.

Firstly, note that apart from the pure `Either` and its particular cases Haskell provides a bunch of other mechanisms for error handling: most notably, imprecise exceptions and several different type classes claiming to implement generic `throw` and `catch` with slightly different semantics.

Secondly, note that `type State s a = s -> (a, s)` uses a single type `s` on both sides of the arrow. If one is to take a fundamentalist view that all computations are just compositions of state transformers and should be expressed as such, then it is immediately apparent that `State` is too restrictive for the general use case as it can not express state transitions between arbitrary data types.

In other words, while a fundamentalist Haskell programmer could feel content parsing `Stream`s (in particular, `String`s) into data types with the help of a parser combinator library like Parsec [11], to do most other things he/she would have to succumb to using several different approaches to error handling while pattern-matching data types manually or with libraries such as SYB [10], Uniplate [14], Multiplate [20], and Lenses [8, 9].

Which is not to say that doing all those things is inherently bad, but it is interesting to see just how much can be done with just `Either`, `State`, `Applicative`, and `Monad` and their natural extensions, that is to say that it is interesting to see how much can be done with very basic theoretical constructs and their combinations. The purpose of this work is to show that the set of things expressible using these structures is surprisingly large. Or, more specifically, to show that *all* of the problems commonly thought of as requiring special care mentioned above can in fact be solved by reusing those well-known structures with minor (if any) modifications.

3 Contributions

Specifically, every item in the following list, to our best knowledge, is a headline contribution.

- We note that the types of

  ```
  throw :: e -> c a
  catch :: c a -> (e -> c a) -> c a
  ```

  operators are special cases of Monadic `pure` (`return`) and `(>>=)` (`bind`) operators

  ```
  pure :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
  ```

  (substitute $[a \mapsto e, m \mapsto \lambda\_.c\ a]$ into their types).

- Hence, a type of computations `c e a` with two indexes where `e` signifies a type of errors and `a` signifies a type of values can be made a `Monad` twice: once for `e` and once for `a`.

  ```
  class ConjoinedMonads c where
    pure  :: a -> c e a
    (>>=) :: c e a -> (a -> c e b) -> c e b

    throw :: e -> c e a
    catch :: c e a -> (e -> c f a) -> c f a
  ```

  Moreover, for such a structure `throw` is a left zero for `(>>=)` and `pure` is a left zero for `catch`.

- We prove that the type of the above `catch` is most general type for any Monadic structure `\a -> c e a` with additional `throw` and `catch` operators satisfying conventional operational semantics (via simple unification of types for several equations that follow from semantics of said operators). Or, dually, we prove that `(>>=)` has the most general type for expressing sequential computations for Monadic structure `\e -> c e a` (with operators named `throw` and `catch`) with additional `pure` and `(>>=)` operators satisfying conventional operational semantics.

- Substituting a Constant `Functor` for `c` into `ConjoinedMonads` above (i.e., fixing the type of errors) produces the definition of `MonadError`, and, with some equivalent redefinitions, `MonadCatch`. Similarly, `IO` with similar redefinitions is a `ConjoinedMonads` instance too (with the usual caveats).

- `ExceptT` and some other lesser known and potentially novel concrete structures have operators of such types and their semantics matches (or they can be redefined in an equivalent way such that the core part of the resulting structure then matches) the semantics of `Monad` exactly.

- `Monad` type class has a well-known "fish" representation where "`bind`" `(>>=)` operator is replaced by "`fish`" operator

  ```
  (>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
  ```

  and `Monad` laws are just monoidal laws.

  Hence, all those structures can be seen as a pairs of monoids over bi-indexed types with identity elements for respective `bind`s as left zeros for conjoined `bind`s. We find this symmetry to be hypnotic and generalize it further into an observation that it is an instance of Cartesian products of type classes.

- The answer to "Why didn't anyone notice this already?" seems to be that this structure cannot be expressed well in Haskell.

- Meanwhile, it has at least several practically useful instances:

  - Parser combinators that are precise about errors they produce and that reuse common `Monad`ic combinators for both parsing and handling of errors. For instance, the type of `many` for such a parser combinator guarantees that it cannot throw any errors

    ```
    many :: c e a -> c f [a]
    ```

    (since `f` can be anything, it cannot be anything in particular) and

    ```
    choice :: [c e a] -> c e a
    ```

    is an instance of `Monad`ic `sequence` combinator.
  - Conventional exceptions expressed using `Reader Monad` and second-rank `callCC` (the whole idea of which seems to be novel).
  - Error-explicit `IO`, the latter and similar structures with similar motivation were proposed before, but they did not use the fact that their "other half" is a `Monad` too.

- We notice that many practically interesting structures can be described as Cartesian product of a structure handling errors and a structure handling computations, which suggests an interesting direction is programming language design.
- We notice that many `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 (that is, non-`Applicative`) instances of this structure, which includes a curious family of structures that work with Scott-encoded data types as if they are heterogeneous lists of typed values.
- Then, we show that there is, in fact, an infinite family of such "`Applicative`-like" structures. This family can be succinctly described as a family of computations for generalized multi-stack machines with arbitrary data types and/or functions as "stacks".
- Then, we observe that our "`Applicative`-like" is actually a natural generalization of the conventional `Applicative` into dependent types.
- We notice that `Monad`ic parser combinators can be generalized into indexed `Monad`s thus allowing one to "parse" (transform between) arbitrary data types/trees.

References

[1] Thorsten Altenkirch and Jonathan Grattage. "A functional quantum programming language". In: (2005). arXiv: quant-ph/0409065 (cit. on p. 3).

[2] Henk Barendregt. "The Impact of Lambda Calculus in Logic and Computer Science". In: (1997). URL: http://www-users.mat.umk.pl/~adwid/materialy/doc/church.pdf (cit. on p. 2).

[3] Felice Cardone and J. Roger Hindley. *History of Lambda-calculus and Combinatory Logic*. 2006. URL: http://www.users.waitrose.com/~hindley/SomePapers_PDFs/2006CarHin,HistlamRp.pdf (cit. on p. 2).

[4] GHC Project Authors. *Hackage: The base package, version 4.9.0.0*. 2016. URL: https://hackage.haskell.org/package/base-4.9.0.0 (cit. on p. 3).

[5] *Hackage: Haskell Central Package Archive*. 2018. URL: https://hackage.haskell.org/ (cit. on p. 3).

[6] S. C. Johnson and B. W. Kernighan. *The Programming Language B*. Technical report 8. Murray Hill, NJ, USA: Bell Laboratories, 1973 (cit. on p. 2).

[7] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Upper Saddle River, NJ 07458, USA: Prentice-Hall, 1978, pp. x + 228. ISBN: 0-13-110163-3 (cit. on p. 2).

[8] Edward Kmett. *Lenses, Folds and Traversals*. 2013. URL: http://lens.github.io/ (cit. on p. 4).

[9] Edward Kmett et al. *Hackage: The lens package, version 4.17*. 2018. URL: https://hackage.haskell.org/package/lens-4.17 (cit. on p. 4).

[10] 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. 4).

[11] 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. 4).

[12] 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 p. 3).

[13] MirageOS Project Authors. *MirageOS: A programming framework for building type-safe, modular systems*. 2019. URL: https://mirage.io/ (cit. on p. 3).

[14] Neil Mitchell and Colin Runciman. *Uniplate*. 2007. URL: http://community.haskell.org/~ndm/uniplate/ (cit. on p. 4).

[15] 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. 3).

[16] 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. 3).

[17] Matthew Naylor and Colin Runciman. "The Reduceron reconfigured and re-evaluated". In: *Journal of Functional Programming* 22.4–5 (Sept. 2012), pp. 574–613. ISSN: 0956-7968 (print), 1469-7653 (electronic). DOI: 10.1017/S0956796812000214. URL: https://www.cambridge.org/core/product/9818E081664ADAFE9F61F1AEDAD0B043 (cit. on p. 3).

[18] Matthew Naylor, Colin Runciman, and Jason Reich. *The Reduceron*. 2010. URL: https://www.cs.york.ac.uk/fp/reduceron/ (cit. on p. 3).

[19] NumPy Project Authors. *NumPy: Scientific Computing with Python*. 2019. URL: https://numpy.org/ (cit. on p. 3).

[20] 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. 4).

[21] Qubes OS Project Authors. *Qubes OS: A reasonably secure operating system*. 2019. URL: https://www.qubes-os.org/ (cit. on p. 2).

[22] Dennis M. Ritchie and Ken Thompson. "The UNIX time-sharing system". In: *Fourth ACM Symposium on Operating Systems Principles, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 15-17, 1973*. Ed. by ACM. New York, NY, USA: ACM Press, 1973, ??–?? URL: https://www.bell-labs.com/usr/dmr/www/cacm.html (cit. on p. 2).

[23] Amr Sabry. *Topics in Programming Languages: Reversible and Quantum Computing*. 2011. URL: https://www.cs.indiana.edu/~sabry/teaching/b629/s11/ (cit. on p. 3).

[24] The Go Programming Language Project Authors. *The Go Programming Language: Standard Library Packages*. 2019. URL: https://golang.org/pkg/ (cit. on p. 3).

[25] The Rust Programming Language Project Authors. *The Rust Standard Library*. 2019. URL: https://doc.rust-lang.org/std/ (cit. on p. 3).

[26] *TIOBE Index*. 2019. URL: https://www.tiobe.com/tiobe-index/ (cit. on p. 3).

[27] 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. 3).

[28] Philip Wadler and Peter Thiemann. "The marriage of effects and monads". In: *ACM Transactions on Computational Logic* 4.1 (Jan. 2003), pp. 1–32. URL: http://homepages.inf.ed.ac.uk/wadler/papers/effectstocl/effectstocl.ps.gz (cit. on p. 1).