



**HAL**  
open science

# Sur le pouvoir expressif des structures applicatives et monadiques indexées

Ian Malakhovski

► **To cite this version:**

Ian Malakhovski. Sur le pouvoir expressif des structures applicatives et monadiques indexées. Analyse numérique [cs.NA]. Université Paul Sabatier - Toulouse III, 2019. Français. NNT : 2019TOU30118 . tel-02735749

**HAL Id: tel-02735749**

**<https://theses.hal.science/tel-02735749>**

Submitted on 2 Jun 2020

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



# THÈSE

## En vue de l'obtention du DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par l'Université Toulouse 3 - Paul Sabatier  
Cotutelle internationale : ITMO University

Présentée et soutenue par  
**Ian MALAKHOVSKI**

Le 15 octobre 2019

Sur le pouvoir expressif des structures applicatives et  
monadiques indexées

Ecole doctorale : **EDMITT - Ecole Doctorale Mathématiques, Informatique et  
Télécommunications de Toulouse**

Spécialité : **Informatique et Télécommunications**

Unité de recherche :  
**IRIT : Institut de Recherche en Informatique de Toulouse**

Thèse dirigée par  
**Sergei SOLOVIEV et Nikolay VASILYEV**

Jury

M. Thomas EHRHARD, Rapporteur  
M. Christian RETORE, Rapporteur  
Mme Ileana OBER, Examinatrice  
M. Maxim BUZDALOV, Examineur  
M. Aleksy SCHUBERT, Examineur  
M. Zhaohui LUO, Examineur  
M. Sergei SOLOVIEV, Directeur de thèse  
M. Nikolay VASILYEV, Co-directeur de thèse

IRIT, University of Toulouse-3 Paul Sabatier and  
Saint Petersburg National Research University of Information Technologies, Mechanics and  
Optics

---

## On the Expressive Power of Indexed Applicative and Monadic Structures

by  
Jan Malakhovski

---

A dissertation submitted to the graduate school of Mathematics, Computer Science and Telecommunications of University of Toulouse-3 Paul Sabatier in partial fulfillment of the requirements for the degree of Doctor of Philosophy

**Scientific advisors:**

Sergei Soloviev  
Nikolay Vasilyev

**Thesis Committee:**

Thomas Ehrhard  
Christian Retoré  
Ileana Ober  
Maxim Buzdalov  
Aleksy Schubert  
Zhaohui Luo  
Sergei Soloviev  
Nikolay Vasilyev

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 other 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 [111] 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.

Unless you are reading the document you yourself recently fetched from <https://oxij.org/thesis/PhD/> there is likely to be a better version of this document there. It is not too late to switch to reading that one yet.

# Acknowledgments

This work would have been impossible without Sergei Soloviev who patiently read and meticulously commented numerous drafts of this document, generously gave invaluable advice, encouraged me to pursue ideas I was ready to discard, spent weeks of his netto-time discussing topics in science, literature, politics, and other facets of intellectual life with me, and helped me to carry through numerous bureaucratic hurdles. These four years of close work with Sergei thoroughly demonstrated to me what “scientific advisor” actually meant to mean and even managed to soften my general outlook on academic life when I thought I was completely disillusioned.

This work would have also been impossible without numerous people, including but not limited to [Sergey Baranov](#), Nikolay Vasilyev, Georgiy Korneev, Ralph Matthes, Maxim Buzdalov, Alexey Sergushichev, and Anatoly Shalyto, who encouraged me to pursue these topics after I described the general ideas to them. Of those, discussions with Georgiy Korneev and Sergey Baranov were of most tremendous help and influence: discussions with Georgiy convinced me to pursue these topics in the first place, while discussions on related topics and critiques of ideas presented in this work by Sergey steered the first half of this document into its current form.

I also want to thank Thomas Ehrhard and Christian Retoré for their helpful feedback that pointed to the parts of the manuscript that needed further clarifications and emphasis.

★

There are numerous people whose efforts produced intellectual environments that steered my general thought patterns into their current states. For most of them I won’t ever know their names, nor the degree to which their work and/or ideas influenced me, but there are some I can immediately remember:

- developers and contributors to GHC Haskell compiler, Coq, Agda, and, most recently, Idris and Lean theorem provers, their underlying calculi, as well as their numerous libraries, whose monumental efforts created tools that allowed computers to be taught enough mathematics and algebra for them to teach it back to me, who started as an undergraduate uninterested in anything but the most “practical” of computing;
- developers and contributors to Emacs and Org-Mode without whom neither the planning, nor the writing of the actual text of this document, nor the steering of the rest of my intellectual life would have been manageable;
- developers and contributors to NixOS and sibling projects without whom I would have gone insane managing my computing devices;
- Aleksandr Lubischev, Daniel Dennett, Noam Chomsky, Scott Aaronson, and Eliezer Yudkowsky whose numerous works on self-discipline, philosophy, rationality, numerous pointers to other fascinating literature, and, for the latter author, even his fictional stories, formed my recent years (I wish I started reading those things when I was nine).

★

Last but not least, I want to thank my family that supports me regardless of anything I do.

# To the Reader

Historiography (a set of, usually documented, events) in general, and the historiography of programming languages in particular, is rather messy. Any attempt at making it into a *history*, that is, a story, necessarily means selecting a starting point, throwing most of the relevant events out and then smoothing transitions between rather discrete points that are still left as to make the whole look a set of interwoven continuous streams following some internal or external logic, the most popular choice being causality.<sup>1</sup>

In regards to literary scientific works this process requires supplying them with historical introductions and organizing new discoveries to be presented in some “logical” fashion to make the work in question look as a nice logically consistent extension of those historical accounts. However, it is important to remember that the reality of both historiography of things preceding a given work and historiography of discoveries described in it is often just a set of accidents. Which is to say that history in general, and presentation of results of a given work in particular, is fiction whose only purpose is to confuse reader’s mind to redirect his/her attention from inconvenient messy facts to convenient seemingly logically consistent stories of a kind that is easy to retain in human memory.<sup>2</sup> Or, as they say, “stories trump data”. This work, being a Ph.D. thesis, shall follow the tradition and amply spice up the technical results with those fictional accounts.


However, the author boldly assumes that there are readers uninterested all these fictional wrappers some of whom are also convinced that a programming language like Haskell and/or its libraries can not be learned by reading a document without doing any actual programming practice. Such readers might be interested in the following facts.

- Firstly, note that this work uses Haskell syntax extensively for the purposes of precise expression of thought. In particular, most proofs use Haskell notation for equation reasoning and normal descriptive text uses Haskell type class names for the names of the respective algebraic structures where appropriate (e.g. “**Monad**” instead of “monad”).
- Secondly, assuming some familiarity with Haskell, this work can be read by starting from the abstract above, followed by chapter 3, followed by jumping to the “meat” of the matters by following the intra-document hyperlinks of section 3.1. Following this regime effectively cuts the number of pages in this work in half. Note, however, that the document organized in such a way so that one could seamlessly fallback to the basics in case some algebraic structure is not familiar by following links to sections of chapter 4 and chapter 5 on by-need basis. Anything lacking there can usually be answered by referencing Diehl’s web-page [20], GHC’s `base` package [26], especially the types and descriptions of functions from the `Prelude` module, and Typeclassopedia [40].
- Thirdly, this work is organized as a series of Literate Haskell programs in a single Emacs Org-Mode tree [21, 98] (then, most likely, compiled into the representation you are looking at right now). The literate source itself is available at <https://oxij.org/thesis/PhD/> and embedded

---

<sup>1</sup> As a fun example, Brownian motion was described by Lucretius in his scientific poem “On the Nature of Things” circa 60 BCE, and Nobel prize in physics for works about this phenomenon was awarded to Jean Perrin in 1926. Which is not to say that either contribution or any of the unknown preceding and known intermediate events that were ignored in the previous sentence are inconsequential (in fact, on the contrary, anything of consequence is usually independently rediscovered at least twice, which itself can be used as a metric of importance of results in a particular work), but to say that history of physics of matter presented as a smooth sequence of discoveries is pure fiction.

<sup>2</sup> Which, by the way, is also very much the case for autobiographic history and thus self-authoring part of what is usually called “personality”. But we digress.

straight into the PDF version of this work (click  or look for “attachments” in your PDF viewer).

In author’s opinion, technical details of sections involving Scott-encoded data types and CPS-transformed terms are incomprehensible without some hacking (playing, experimenting) with terms and types of the Literate version loaded into `ghci`. The two most essential techniques for interacting with `ghci` employed by the author himself are:

- replacing random terms in the source with “\_” holes, reloading, and looking at compiler output, and
- erasing and then trying to interactively reinvent the terms in question using “\_” holes.

All runnable sources were tested with GHC [28] version 8.6.

- Finally, the honest account of the historiography of this work is as follows. All of the results presented in this work are either side-effects of conscious efforts to reuse algebraic structures of author’s liking to solve author’s common programming problems or things accidentally discovered while trying to explain those efforts in talking or writing. Everything else, including the presented sequencing from simpler to more complex algebraic structures and the story and the overall philosophical picture of where this work falls within modern programming practice and language design are after-the-fact fictions.

For other readers, let us note that this work is written in such a way so that familiarity with Haskell (or even  $\lambda$ -calculus) is not actually required to understand the outline of ideas this work presents. The minimal effective dose of Haskell (and  $\lambda$ -calculus) can be learned by osmosis by reading this work in a linear fashion.

Moreover, we are obliged to specifically point out that despite the heavy use of Haskell notation, most of the results of this work are actually **language-agnostic** and can be applied (if not straight to practice, then at least to inform design choices) to almost any programming language.

# Contents

<b>I</b>	<b>Introductions</b>	<b>7</b>
<b>1</b>	<b>General Introduction</b>	<b>8</b>
<b>2</b>	<b>Short Introduction to Haskell</b>	<b>11</b>
2.1	Term Notation . . . . .	11
2.2	Type Notation . . . . .	14
2.3	Data Types . . . . .	15
2.4	Type Classes . . . . .	17
2.5	Applicatives and Monads . . . . .	18
<b>3</b>	<b>Extended Abstract</b>	<b>22</b>
3.1	Headline Contributions . . . . .	23
<b>4</b>	<b>Introduction to the Basic Structures of Haskell</b>	<b>25</b>
4.1	Before-Monadic . . . . .	25
4.1.1	Monoid . . . . .	25
4.1.2	Functor, Pointed, Applicative . . . . .	26
4.1.3	Alternative . . . . .	28
4.2	Purely Monadic . . . . .	28
4.2.1	Monad . . . . .	29
4.2.2	MonadFish . . . . .	30
4.2.3	Monad's fail and MonadFail . . . . .	31
4.2.4	Identity Monad . . . . .	32
4.2.5	Maybe Monad . . . . .	32
4.2.6	Either Monad . . . . .	33
4.3	An Intermission on Monadic Boilerplate . . . . .	33
4.4	MonadTransformers . . . . .	34
4.4.1	Identity . . . . .	34
4.4.2	Maybe . . . . .	35
4.4.3	Except . . . . .	35
4.4.4	Reader . . . . .	37
4.4.5	State . . . . .	37
4.5	Imprecise Exceptions . . . . .	38
4.5.1	IO . . . . .	38
4.5.2	raise# and catch# . . . . .	39
4.5.3	Typeable . . . . .	39
4.5.4	Exception . . . . .	40
4.5.5	throw and catch . . . . .	40
4.5.6	error and undefined . . . . .	41
4.6	Precise raiseIO# and throwIO . . . . .	41
4.7	Non-exhaustive Patterns . . . . .	42
4.8	Monadic Generalizations . . . . .	43
4.8.1	MonadError . . . . .	43
4.8.2	MonadThrow and MonadCatch . . . . .	43



<b>5</b>	<b>Introduction to Some Non-basic Structures of Haskell</b>	<b>45</b>
5.1	Continuations . . . . .	45
5.1.1	Continuation-Passing Style . . . . .	45
5.1.2	Scott-encoding . . . . .	47
5.1.3	Cont . . . . .	49
5.1.4	Delimited callCC . . . . .	50
5.1.5	Scheme's call/cc and ML's callcc . . . . .	51
5.2	Monadic Parser Combinators . . . . .	51
5.2.1	Simple Monadic Parser Combinator . . . . .	52
5.2.2	... with Full Access to the State . . . . .	53
5.2.3	Rollback vs. Not, Programmer-provided Error Types . . . . .	54
5.3	Indexed Monads . . . . .	55
5.3.1	Indexed State Monad . . . . .	56
5.4	Other Variants of MonadCatch . . . . .	57
<b>II</b>	<b>Exceptionally Monadic Error Handling</b>	<b>58</b>
<b>6</b>	<b>Motivation</b>	<b>59</b>
<b>7</b>	<b>Derivation</b>	<b>62</b>
7.1	The Nature of an Error . . . . .	62
7.2	The Type of Error Handling Operator . . . . .	63
7.3	Formal Account: Conjoinedly Monadic Algebra . . . . .	64
7.4	Instance: Either . . . . .	65
7.4.1	Logical Perspective . . . . .	66
7.5	Formal Account: Haskell Encoding . . . . .	66
<b>8</b>	<b>Non-trivial Instances</b>	<b>69</b>
8.1	Constant Functors . . . . .	69
8.1.1	MonadError . . . . .	69
8.1.2	MonadThrow and MonadCatch . . . . .	69
8.2	Parser Combinators . . . . .	70
8.2.1	The Boring Part . . . . .	70
8.2.2	The Interesting Part . . . . .	72
8.3	Conventional throw and catch via callCC . . . . .	73
8.3.1	Second-rank callCC . . . . .	73
8.3.2	ThrowT MonadTransformer . . . . .	73
8.4	Error-explicit IO . . . . .	75
8.5	Conventional IO . . . . .	76
<b>9</b>	<b>Cartesian Products</b>	<b>78</b>
<b>10</b>	<b>Discussion</b>	<b>80</b>
<b>III</b>	<b>Transforming Trees with Generalized Applicative Expressions</b>	<b>82</b>
<b>11</b>	<b>Motivation</b>	<b>83</b>
11.1	Motivating Examples . . . . .	84
<b>12</b>	<b>Derivation</b>	<b>87</b>
12.1	Problem Definition . . . . .	87
12.2	Deriving the Technique . . . . .	88
12.3	Applying the Technique . . . . .	90
12.4	Scott-encoded Representation . . . . .	92
12.5	General Case . . . . .	94
12.6	Formal Account . . . . .	95
12.6.1	Dependently-typed Applicative . . . . .	96
12.6.2	Haskell Encoding . . . . .	96
<b>13</b>	<b>Cartesian Products</b>	<b>100</b>

<b>14 Discussion</b>	<b>103</b>
<b>IV Transforming Trees with Indexed Monads</b>	<b>104</b>
<b>15 Motivation</b>	<b>105</b>
15.1 Motivating Examples . . . . .	105
<b>16 Derivation (and Cartesian Products)</b>	<b>108</b>
16.1 Instance: Simple Indexed Monadic Parser Combinator . . . . .	108
16.2 Non-instance: ... with Full Access to the State . . . . .	110
16.3 Instance: Twice Conjoinedly Indexed Monadic Parser Combinator with Full Access to the State . . . . .	111
<b>17 Examples</b>	<b>113</b>
<b>18 Discussion</b>	<b>116</b>
<b>Conclusions and Future Work</b>	<b>118</b>

**Part I**

**Introductions**

# Chapter 1

## General 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).

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 [47, 53, 96] 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 [93], 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 at IBM (a compiler for IBM 704) around 1956 (first compiler delivered in 1957) and LISP initially developed by John McCarthy at MIT around the same time (first specified in 1958, first universal interpreter implemented by Steve Russell for IBM 704 around 1960, first compiler written in LISP in 1962).

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  $\lambda$ -calculus developed by Alonzo Church and his students in 1930s and 1940s with the explicit goal of making a minimalist universal computational formalism [7, 16] 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  $\text{LISP} \rightarrow \text{ML} \rightarrow \text{Miranda} \rightarrow \text{Haskell}$  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 losing 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 [108] (measures search requests), with its the most popular public code repository of Hackage [39] 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 [86], 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 [75], a modular operating system written entirely in ML. Similarly, Go [106], Haskell [26], and Rust [107] 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 [83, 84] 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, 97], 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 [79, 80] and popularized by Wadler [110] and **Applicative Functors** introduced by McBride and Paterson [74]. 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. To explain those generalizations, however, we need to start talking in Haskell.

## Chapter 2

# Short Introduction to Haskell

### 2.1 Term Notation

Firstly, to shortly explain Haskell term notation, let us consider school-level arithmetical expressions such as

$$a \sin(2\alpha) + b \cos(\alpha) + \text{mod}(n, m)$$

Haskell notation, essentially, is a modification of plain-text arithmetical notation that requires all operators except the function call operator to be explicit, does not require parentheses around function calls and arguments, and does not separate them with commas. In other words, in Haskell, multiplication of `a` and `b` can be written as `a * b` while the expression `a b` would denote an application of the argument “`b`” to the function “`a`”, that is, a function call. Thus, the most straightforward translation of the above expression into Haskell notation would look as follows

```
a * sin (2 * alpha) + b * cos alpha + mod n m
```

Note, however, that this means that in Haskell some expressions will have parentheses in somewhat unexpected places. For instance, conventionally notated

$$\text{mod}(n, m + n)$$

becomes

```
mod n (m + n)
```

as `mod n m + n` would be parsed as `(mod n m) + n` since term juxtaposition, which denotes argument application, grammatically binds more strongly than anything else.

(The reader might feel these conventions to be an instance of useless non-conformism, but they actually make a lot of sense for a functional programming language where most programs spends most of their code doing function calls, as opposed to assigning results of computations of expressions to variables like most imperative languages do.)

Then, Haskell allows to make infix functions like `(+)` into prefix functions by wrapping them into parentheses and the reverse by wrapping them with ``` symbols. For instance, the above school-level term notation example can also be rewritten as

```
(+) (a * sin (2 * alpha)) (b * cos alpha) + n `mod` m
```

Finally, both constant variables and function declarations can be made by simply defining them in blocks of `f a_1 a_2 ... a_n = b` expressions starting at the same indent, with `f` denoting definition’s name, `a_1 ... a_n` denoting the names of its arguments (zero for constant variable and one or more for a function), and `b` denoting a “function body”, i.e. a term the expression in question should evaluate to when supplied with values for all of the arguments. Let us consider, for instance, the following set of definitions

```

x = 2
y = 3
id z = z
foo c
  = c * x + y
bar a b =
  a * foo b

```

Such expressions should be read as left-to-right rewrite rules. For example,

- `x = 2` above says “in an expression, all instances of `x` should be replaced with `2`”,
- and `foo c = c * x + y` above says “in an expression, all instances of `foo c` should be replaced with `c * x + y` for all values of `c`”.

For instance, under the above definitions, the expression

```
x + foo 10
```

can be reduced to

```
2 + 10 * 2 + 3
```

which can then be reduced further by applying reduction rules for `(+)` and `(*)` operators.

Similarly, the whole set of the five definitions above, assuming `x`, `y`, and `id` are not used anywhere else, can be reduced to

```

foo c = c * 2 + 3
bar a b = a * foo b

```

Moreover, function definition can also be inlined by using “ $\lambda$ -expression” notation from  $\lambda$ -calculus [5]

```
bar a b = a * (\c -> c * 2 + 3) b
```

where `\a_1 ... a_n -> b` denotes an anonymous (unnamed) function with an arguments `a_1 ... a_n` and term `b` as the body. Thus, the normal `f a_1 ... a_n = b` block notation can be viewed as a syntax sugar for `f = \a_1 ... a_n -> b` blocks.

This, of course, means that Haskell can trivially provide a mechanism for “incomplete function calls” (also known as *currying*), that is, give simple semantics to expressions that call functions with some of their arguments missing by simply wrapping them with  $\lambda$ -expressions. For instance,

```
z = bar 2
```

is equivalent to

```
z = (\b -> bar 2 b)
```

(in the theory of  $\lambda$ -calculus this transformation is called “ $\eta$ -expansion” and its reverse “ $\eta$ -reduction”) which can then be reduced to

```
z = \b -> 2 * (\c -> c * 2 + 3) b
```

Then, function definitions can also use *pattern matching* syntax, by spelling out the possible combinations of argument values. For instance,



```

lnot 0 = 1
lnot 1 = 0

natMinus 0 = 0
natMinus n = n - 1

```

Outside of function definitions the same thing can also be written using the `case` syntax construct, e.g.

```

lnot' x = case x of
  0 -> 1
  1 -> 0

```

Finally, there is also a builtin `if-then-else` construct

```

lnot'' x = if x == 0 then 1
           else if x == 1 then 0
           else -1

```

Note that the above description of the notation implies that all expressions in Haskell are referentially transparent (i.e. there's no distinction between “value” and “reference” types, everything is a value) and all functions are pure (i.e. without side-effects). Thus, the act of evaluating a given Haskell expression (term) consists of repeated inlinings of all the definitions the term uses followed by repeated substitutions of applied function arguments into the corresponding function bodies (possibly with introductions of some intermediate  $\lambda$ -expressions) until nothing else can be reduced. Therefore, working with “references” requires separate mechanisms, which we are going to discuss in the following sections.

Meanwhile, note that if one wants to evaluate a single term of a Haskell program (when compiling a Haskell program into machine code such a term named `main`) then it and the rest of the program can be equivalently expressed as a bunch of nested  $\lambda$ -expressions and applications. For instance, the five definitions above, when evaluating term `bar`, can be equivalently expressed as

```

(\x \y ->
  (\foo -> \a b -> a * foo b) (\c -> c * x + y)
) 2 3

```

In other words, when working with a single expression, named definitions given on separate lines can be viewed as a syntax sugar for a  $\lambda$ -expression introducing the name as an argument immediately followed by application of the body of that definition to that  $\lambda$ -expression.

Thus, the model of evaluation of Haskell expressions can be simplified even more: all Haskell does is it de-syntax-sugars its terms into something very similar to  $\lambda$ -calculus and then repeatedly substitutes function arguments into anonymous functions until nothing else can be reduced, i.e. this is exactly  $\beta$ -reduction of  $\lambda$ -calculus.

Note, however, that when dealing with function arguments, in Haskell, as in any other sane programming language, variables are never “captured”. For instance,

```

a = 2
foo x = a * x
bar a b = foo a * b

```

can be reduced to

```

bar a b = (2 * a) * b

```

not `bar a b = a * a * b`, nor `bar a b = 2 * 2 * b`, nor anything else. That is to say, arguments of both named and anonymous functions can be freely renamed without changing the meaning of an expression, and hence proper definition of inlining and function argument substitution should respect that property.

Technically speaking, one can simply forget about variable names and consider variable references to be pointers to the corresponding function arguments, with a variable reference denoting a pointer to the closest enclosing argument with the same name introduced by a  $\lambda$ -expression. Or, equivalently, one can annotate  $\lambda$ -expressions with numbers of variables they introduce (instead of their names) and make variable references into expressions denoting “an argument number  $n$  of the  $m$ ’s enclosing  $\lambda$ -expression”. Such an encoding bears a name of de Bruijn indicies [13].

Finally, we have to discuss recursive functions. The simplest way to explain them is to think them as being non-inlineable and calling themselves using function pointers (which is essentially what compilation from Haskell to machine code does). Alternatively, one can imagine that Haskell has a builtin fixed-point operator, which we could define as having the following rewrite rule

```
fix f = f (fix f)
```

and then to think of all recursive functions as using `fix` while taking themselves (i.e. a pointer to themselves) as their first argument. For instance,

```
fib' fib 0 = 1
fib' fib n = n + fib (n - 1)
```

```
fib = fix fib'
```

As a side note, note that the above shows that, basically, any pure Haskell program can be expressed in terms of just  $\lambda$ -expressions, `fix`, and some primitive types (like integers) and their operators like (+). The beauty of the untyped  $\lambda$ -calculus comes from the fact that it can also express `fix` (and other fixed point operators), numbers, trees, operators over them, and, in fact, anything computable, using cleverly constructed  $\lambda$ -terms and nothing else. That is, untyped  $\lambda$ -calculus is a truly minimalist universal programming language. A reader interested in such things is referred to [5, 6]. Visitors from the outer space culturally unrelated to the human race are very unlikely to know anything about x86 assembly (unless they specifically tried to learn it), alien races anatomically dissimilar to humans might even not know about Turing machines (since a Turing machine does not make much sense unless you are accustomed to manipulating your own hands), but they will probably know about prime numbers and  $\lambda$ -calculus (in the de Bruijn form).

## 2.2 Type Notation

In theory, Haskell is a strictly typed language and, hence, a programmer coming from an imperative language would probably expect to see a lot of type annotations. In practice, however, one can write a lot of Haskell code without ever mentioning types. Since Haskell supports type inference almost all type annotations can usually be left out. However, spelling out the types of expressions can be useful for documentation and debugging purposes, it is also required in some syntactic expressions, like data types and type classes discussed below.

In plain Haskell 98, to attach a type to an expression one uses `a :: t` syntax, where `a` denotes a name and `t` denotes a type expression. Most Haskell programs usually use those expressions to attach types to top-level definitions. For instance,

```
x :: Int
x = 2

foo :: Int -> Int
foo x = a * x
```

In type expressions simple types are denoted with their names, function types are denoted with arrows, which associate to the right ( $a \rightarrow b \rightarrow c == a \rightarrow (b \rightarrow c)$ ). Which, of course, means that a function of  $n + 1$  arguments can be seen as a function of one argument that returns a function of  $n$  arguments. Which is just currying discussed in the previous section.

Polymorphic functions (“template functions” in C++/Java-speak) are denoted by using `forall` operator

```
id :: forall x . x -> x
id x = x * x
```

```
consF :: forall b . b -> (forall a . a -> b) -> b
consF b _ = b
```

Moreover, “naked” (not under parentheses) leading `forall`s can be omitted, thus the above is equivalent to

```
id :: x -> x
id x = x * x
```

```
consF :: b -> (forall a . a -> b) -> b
consF b _ = b
```

In fact, Haskell 98 does not allow any explicit `forall`s, explicit `forall`s are a GHC extension named `RankNTypes`.

Finally, while Haskell 98 only allows to attach types to named definitions, GHC with `ScopedTypeVariables` extension allows to attach them to arbitrary expressions. For instance,

```
foo :: Int -> Int
foo x = a * (x :: Int)
```

## 2.3 Data Types

While pure  $\lambda$ -calculus can express all computable programs [5, 6], in practice, constructing all values from  $\lambda$ -terms is very inefficient. Haskell follows the conventional way to solve this problem by extending  $\lambda$ -calculus with builtin types, values, and operations that can be efficiently represented and manipulated on conventional hardware, combined with providing ways to combine those types to produce more complex ones.

In Haskell, those builtins include

- arbitrarily long integers `Integer` like `118 :: Integer` or `2 ** 256 :: Integer` (two to the power 256), there are also types that cover subsets of `Integer` like `Int`, `Word`, `Int32`, `Word32`, etc;
- rational numbers `Rational` like `-1.5 :: Rational` and `1/3 :: Rational` and subsets like `Double`, `Float`, etc;<sup>1</sup>
- characters `Char` like `'A' :: Char`; and
- strings `String` like `"Hello, World!" :: String`.

Haskell also provides two builtin type constructors

- tuples of arbitrary types, like `(1, 2, "Hello") :: (Integer, Rational, String)`,
- lists of values of a given type, like `[1, 2, 3] :: [Integer]` and `1 : 2 : 3 : [] :: [Int]` (which are the same thing).

---

<sup>1</sup> Note how Haskell provides the ideal versions of arbitrary precision out of the box.

It also provides a way to define tagged unions of Cartesian products of types denoted with `data` expressions, for instance

```
data OptionalIntChar = Null | AnInt Int | AChar Char | Both Int Char
x1 = Null           :: OptionalIntChar
x2 = AnInt 1        :: OptionalIntChar
x3 = AChar 'A'      :: OptionalIntChar
x4 = Both 1 'A'     :: OptionalIntChar
```

The very first lexeme in each of the blocks between the “|” symbols is a constructor name (which is a tag to be used for that case in the union), the following ones are types of fields (which are then taken into a Cartesian product). The names of constructors must be capitalized.

Moreover, data expressions can also take types as arguments, thus making them generic, e.g.

```
data Maybe a = Nothing | Just a
x1 = Nothing  :: Maybe Int
x2 = Just 1    :: Maybe Int
x3 = Just 1    :: Maybe Float
x4 = Just 'A'  :: Maybe Char
x5 = Just (\x -> x) :: Maybe (Int -> Int)
```

Functions involving these data types can also be used with pattern matching as if they are builtin types

```
foo :: Maybe Int -> a -> Int
foo Nothing b = b
foo (Just a) b = a + b
```

Thus, tuples are just a syntax sugar for

```
data Pair a b = Pair a b

appair f (a, b) = f a b
appair' f (Pair a b) = f a b

data Triple a b c = Triple a b c
-- ...
```

Data type syntax also allows to name to elements of the Cartesian products, for instance

```
data Quadruple a b c d = Quadruple
  { first :: a
  , second :: b
  , third :: c
  , fourth :: d
  }
```

which also defines those names into functions extracting corresponding elements from the product

```
sumQuadruple q = first q + second q + third q + fourth q
```

Data types can also be recursive, for instance

```
data List a = Nil -- an empty list
            | Cons a (List a) -- prepend @a as the head of @rest
```

In fact, builtin lists can be defined as

```
data ([]) a = []
  | (:) a (List a)
```

and `String` type is actually defined as `[Char]` (and string literals are just a special syntax for such data types).

Finally, type expressions can be given names similarly to normal expressions using the `type` keyword, and those expressions, too, can include arguments (thus producing “template types” in C++/Java-speak), for instance

```
type State' s a = s -> (a, s)
```

and data types of a single constructor and single field can be defined using `newtype` keyword instead of `data`, for instance

```
data State' s a = State' { runState' :: s -> (a, s) }
newtype State s a = State { runState :: s -> (a, s) }
```

The only difference between these definitions is the fact that for `newtype` the `State` tag is purely syntactic, while `State'` of the definition using the `data` keyword is going to be stored in memory (which semantically also means that the `data` definition introduces lazyness, as one can pattern match on the tag and not on the field, which is impossible with `newtype`).

## 2.4 Type Classes

The `forall` universal quantifier provides a way to implement universally polymorphic functions, that is, functions that use the same terms for different types. Unfortunately, this universality means that the functions in question can not do much with those arguments of universally quantified types, since those operations must apply to all the possible types. In fact, they can do nothing except ignoring them and propagating them into subcomputations (that is, copying). For instance, naively, there cannot be a universal addition operator such that a function

```
f a b = a + b
```

could accept arguments of both, say, `Int` and `Float` types.

Note, however, that in functional programming language nothing prevents us from supplying such a (+) operator as an argument to the function in question, which would allow us to type it as

```
f :: (a -> a -> a) -> a -> a -> a
f (+) a b = a + b
```

Then, we can organize sets of such operators and constants into algebraic structures (an object-oriented programming language would call them *interfaces*) and supply functions that need those operators and constants with such structures as arguments, thus getting ad-hoc polymorphism essentially for free

```
data Monoid a = Monoid
  { zero :: a
  , plus :: a -> a -> a
  }
```

```
intMonoid :: Monoid Int
intMonoid = Monoid intZero intPlus
```

```
f :: Monoid a -> a -> a -> a
f (Monoid _ (+)) a b = a + b
```

The only problem with the above is that we have to explicitly construct, apply, and pattern-match arguments of those interface types, which is rather inconvenient. Which is why Haskell provides “type classes” mechanism which does most of those things automatically.

With type classes one uses `class` syntax construct instead of `data` to define the type class, `instance` syntax construct instead of plain terms to define its instances, separates them with `=>` symbol in type signatures, and does not pattern match on those arguments at all. For instance, the above example would be translated as

```
class Monoid a where
  zero :: a
  plus :: a -> a -> a

instance Monoid Int where
  zero = intZero
  plus = intPlus

f :: Monoid a => a -> a -> a
f a b = a + b
```

Type classes can also extend each other

```
class Monoid a => Group a where
  inverse :: a -> a

instance Group Int where
  inverse = intInverse
```

and instances can be defined as being derived from other instances (thus, effectively, defining functions over type classes)

```
class Invertible a where
  inv :: a -> a

instance (Monoid a, Invertible a) => Group a where
  inverse = inv
```

Haskell compiler will then try to apply appropriate instances into all function calls, thus, effectively performing a simple Prolog-like witness search. Moreover, as with `data` keywords GHC Haskell with `MultiParamTypeClasses` extension enabled allows type classes have arbitrary number of arguments. `FunctionalDependencies` extension also adds a syntax that allows specifying some parameters as dependent on others. Thus, in fact, one can encode some rather complicated Prolog programs with type classes.

## 2.5 Applicatives and Monads

This work primarily concerns itself with `Applicative` and `Monad` type classes that, ideally (the reality of Haskell’s standard library is somewhat less pretty, see chapter 4), make up the following type class hierarchy

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

class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

```

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

infixl 1 >>=
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b

```

All of those are discussed in detail in chapter 4. In this section we shall only try to give the overall description of the latter two structures so that the following sections would make some sense.

The above `Applicative` type class, essentially, defines an algebraic structure denoting a generalized function application. For instance, an identity on Haskell types is obviously an `Applicative` with `pure = id = \x -> x` and `(<*>) = \f x -> f x` 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, we shall discuss those in later sections.

The `Monad` type class is a bit harder to explain, but essentially, it is a generalization of imperative “semicolon” operator. To explain why, consider the following pseudo-C program

```

x = foo(args);
rest

```

where `rest` is a subprogram that potentially involves other semicolons. The above pseudo-code, essentially, just binds the result of evaluation of `foo(args)` to a name `x` in a subprogram named `rest`. For instance, the following C program would be an instance of the above construction

```

something_t main () {
  x = foo();
  y = bar(x);
  return baz(x, y);
}

```

Assuming `foo` is a pure function (without side-effects), the original expression can be encoded in Haskell as

```
(\x -> rest) (foo args)
```

Thus, the body of the `main` function in the latter program, again, assuming `foo`, `bar`, and `baz` are pure functions, can be encoded in Haskell as

```
(\x -> (\y -> baz x y) (bar x)) foo
```

or, equivalently, if we are to denote a flipped application with `andThenContinueTo` operator and add some newlines

```

foo `andThenContinueTo` (\x ->
bar x `andThenContinueTo` (\y ->
baz x y))

```

The `(>>=)` operator of the `Monad` type class is just one possible way to type such a `andThenContinueTo` operator (there are others, which we shall extensively discuss in this work).

Note, however, that the type of `(>>=)` actually removes the purity requirements we had to constantly mention above. In Haskell, impure functions are encoded as pure functions that generate impure “actions” an impure run-time system would eventually *run* (as opposed to *evaluate*). In other words, those “actions” are to be interpreted outside of the language, inside the language they are just values (e.g. think system call numbers with attached data or similar).

For instance, assuming `newIORef` is such an action that encodes an allocation of a piece of memory on a heap and `assign x y` is an action that encodes an assignment of *value* `y` into a piece of memory *pointed by value* `x` (note, however, that `assign` then, is a function that takes two arguments and produces such an action), and assuming `(>>=)` denotes an opaque function provided by the run-time system that *runs* an impure action given as its first argument, applies its result to a function given as its second argument, then *runs* the *result of evaluating* that as an impure action, the result of which it then returns, then the following impure C program

```
void foo(int * x) {
    x = 1;
    x = 2;
}

void bar() {
    int * x = malloc(sizeof(int));
    foo(x);
}
```

can be, more or less, equivalently encoded in Haskell as

```
foo x =
    assign x 1 >>= \_ ->
    assign x 2

bar = do
    newIORef >>= \x ->
    foo x
```

inlining of which then produces the following expression

```
newIORef >>= \x ->
assign x 1 >>= _ ->
assign x 2
```

Then, to make this expression evaluate similarly to the C version one needs to invent some values that can be used as encoding for `newIORef` and `assign x y` and implement an appropriate `(>>=)` for them. That is to say, one needs to make an interpreter that would interpret all references to those symbols adhering to the desired semantics. That is to say, inside the program one can think of `newIORef` and `assign` as being elements of a data type run-time system will later interpret.

The point in all of the above is that by generalizing the “semicolon” operator of C into `(>>=)` of Haskell one can get many useful benefits. For instance, note that `bar` in the C code above ignores the possible error of `malloc`. The conventional way to resolve this problem is to either explicitly check for those errors with something like

```
void bar() {
    int * x = malloc(sizeof(int));
    if (x == NULL) abort();
    foo(x);
}
```

or use a library that provides a wrapper around `malloc` that performs such a check and calls `abort` inside. Safer languages usually only provide the second kind of `malloc` that `throw` Out-Of-Memory *exceptions* or similar.

Which is usually fine for `malloc`, since a failure to `malloc` usually means that the program can not continue. But consider, for instance, a less obviously deadly issue of failed logging. Clearly, a failure



to log an event might be a critical problem or non-issue depending on context. Thus, usually, logging libraries provide several sets of interfaces with different semantics and/or a way to globally configure which of the several logging failure semantics (ignore failures, repeat until success, fail on failure, etc) is desired. The latter approach, of course, has all the usual problems involving computations using global variables. The former approach means that computations using an interface with one logging failure semantics can not be reused in computations requiring the other.

By contrast, in Haskell, one could instead have a single logging library with a single zero-configuration interface and several different **Monads** that provide different ( $>>=$ ) operators (which either fail, repeat, ignore, etc on logging failures) and switch between those **Monads** depending on context in a way transparent to subcomputations, thus greatly improving in modularity.

In practice, however, defining separate **Monads** for different kinds of computations is so useful that almost every little thing has its own **Monad** in Haskell. The rest of the work will provide numerous examples.

As a final note on the topic we have to mention that this `foo >>= \x -> rest` construct is frequent enough that Haskell has a special syntax sugar for it, called **do**-syntax, that allows one to write

```
do
  x <- foo
  y <- bar x
  baz x y
```

instead of `foo >>= \x -> bar x >>= \y -> baz x y`, thus making programs involving **do**-syntax look very similar to those written in an imperative languages like C.

## Chapter 3

# 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` (section 4.2.6),
- a `main` function in a Haskell program, more or less, simply interprets a `State` transformer (section 4.4.5) over a `RealdWorld` that computes program outputs from program inputs (i.e. `IO Monad` of section 4.5.1, though it can have other interpretations, see remark 4),
- 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 `Streams` in place of the `RealdWorld`, see section 5.2), or mechanical transformations (e.g. Scott-encoding, see section 5.1.2) of them.

The fact that `Either` and `State Applicatives` and `Monads` 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 (see section 4.5) and several different type classes claiming to implement generic `throw` and `catch` with slightly different semantics (see section 4.8).

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 `Streams` (in particular, `Strings`) into data types with the help of a parser combinator library like `Parsec` [66], 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` [64], `Uniplate` [78], `Multiplate` [87], and `Lenses` [58, 61].

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.

### 3.1 Headline 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, see sections 7.1 and 7.2).

- 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` (see sections 7.3 and 7.4.1).

- 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, see section 7.2). 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 (see footnote 5).
- 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` (see section 8.1). Similarly, `IO` with similar redefinitions and with the usual caveats of remark 4 is a `ConjoinedMonads` instance too (see section 8.5).

- **ExceptT** (section 7.4) and some other lesser known and potentially novel concrete structures (see chapter 8, most interestingly, section 8.3) 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 **binds** as left zeros for conjoined **binds** (section 7.3). We find this symmetry to be hypnotic and generalize it in chapter 9.

- The answer to “Why didn’t anyone notice this already?” seems to be that this structure cannot be expressed well in Haskell (see section 7.5).
- Meanwhile, it has at least several practically useful instances:
  - Parser combinators that are precise about errors they produce and that reuse common **Monadic** 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 **Monadic sequence** combinator (see section 8.2).
  - Conventional exceptions expressed using **Reader Monad** and second-rank `callCC` (the whole idea of which seems to be novel, see section 8.3).
  - Error-explicit **IO** (section 8.4), 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 (chapter 9), which suggests an interesting direction is programming language design (see conclusions).
- We notice that many **Applicative** computations can be interpreted as providing a *mechanism* to construct a data type with “ports” “pluggable” by subcomputations (chapter 11 and section 12.1). 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 (sections 12.2 and 12.3), which includes a curious family of structures that work with Scott-encoded data types as if they are heterogeneous lists of typed values (section 12.4).
- Then, we show that there is, in fact, an infinite family of such “**Applicative-like**” structures (section 12.5). 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” (sections 12.3 and 12.6).
- Then, we observe that our “**Applicative-like**” is actually a natural generalization of the conventional **Applicative** into dependent types (section 12.6.1).
- We notice that **Monadic** parser combinators can be generalized into indexed **Monads** thus allowing one to “parse” (transform between) arbitrary data types/trees (part IV).

## Chapter 4

# Introduction to the Basic Structures of Haskell

While algebraic structures used in this work are simple, there are a lot of them. This chapter is intended as a reference point for all algebraic structures relevant in the context of this document (for reader's convenience and for high self-sufficiency of the Literate Haskell version). Most of those are usually assumed to be common knowledge among Haskell programmers. Note however, that this section is not a tutorial introduction to the structures in question: when discussing a structure we shall only show the most primitive examples of its usage, if any at all. To get a deeper understanding of those structures and their use the reader will have to look into the examples given in the original papers and sources (which we shall cite) and, most likely, do some programming practice.

All structures of this section are ordered from semantically simple to more complex (that is, we do not topologically sort them by their dependencies in GHC sources). For the reasons of simplicity, uniformity, self-containment, and novel perspective some of the given definitions differ slightly from (but are isomorphic/equivalent to) the versions provided by their original authors. The most notable difference is the use of a common `Pointed` type class (see section 4.1.2) instead of conventional duplication of `Monadic return` and `Applicative pure`. All structures are listed alongside references to the corresponding papers, documentation and original source code.

This section can be boring (although, the author feels like most remarks and footnotes are not). On the first reading we advise to skip straight to chapter 5 and refer back to this section on demand.

### 4.1 Before-Monadic

This section describes the simplest type classes used in this work.

#### 4.1.1 Monoid

`GHC.Base` from `base` [26] package defines `Monoid` type class as follows<sup>1</sup>

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a

  -- defined for performance reasons
  mconcat :: [a] -> a
  mconcat = foldr mappend mempty
```

and wants its instances to satisfy the following conventional equations (“`Monoid` laws”)

---

<sup>1</sup> Note that by following `Pointed` logic used below we should have split `Monoid` into two type classes, but since we will not use `Monoids` that much in the rest of the work we shall use the original definition as is.

```

-- `mempty` is left identity for `mappend`,
mempty `mappend` x == x

-- `mempty` is right identity for `mappend`,
x `mappend` mempty == x

-- `mappend` is associative,
x `mappend` (y `mappend` z)
  == (x `mappend` y) `mappend` z

```

and an additional constraint

```

-- and `mconcat` is extensionally
-- equal to its default implementation
mconcat == foldr mappend mempty

```

Signature and default implementation for `mconcat` is defined in the type class because `mconcat` is a commonly used function that has different extensionally equal intensionally non-equal definitions with varied performance trade-offs. For instance,

```

mconcat' :: Monoid a => [a] -> a
mconcat' = foldl' mappend mempty

```

(where `foldl'` is a strict left fold) is another definition that satisfies the law given above (since `mappend` is associative), but this implementation will not produce any superfluous thunks for strict `mappend`.

Arguably, `Monoids` provide the simplest (after “just abort the program”) “error handling” mechanism: programmers can use their neutral elements to represent an error and associative composition to ignore them. Whenever “ignoring” is “handling” is a matter of personal taste.

One of the simpler instances is, of course, a list

```

instance Monoid [a] where
  mempty = []
  mappend = (++)

```

and hence, for instance, functions generating errors can produce empty lists on errors and singleton lists on successes.

#### 4.1.2 Functor, Pointed, Applicative

Most of the structures that follow are `Applicative Functors` [74]. `GHC.Base` from `base` [26] package defines those two algebraic structures as follows

```

class Functor f where
  fmap :: (a -> b) -> f a -> f b

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

```

and wants their instances to satisfy

```

-- `fmap` preserves identity
fmap id == id

-- `(<*>)` is `fmap` for pure functions
pure f <*> x == fmap f x

```

and some more somewhat more complicated equations [40]. We shall ignore those for the purposes of this work (we will never use them explicitly). Meanwhile, for the purposes of this work we shall split the `pure` function out of `Applicative` into its own `Pointed` type class and redefine `Applicative` using it as follows (this will simplify some later definitions).

```
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
```

We shall give all definitions and laws using this hierarchy unless explicitly stated otherwise.

The most trivial example of `Applicative` is the `Identity Functor` defined in `Data.Functor.Identity` of base

```
newtype Identity a = Identity
  { runIdentity :: a }

instance Pointed Identity where
  pure = Identity

instance Functor Identity where
  fmap f (Identity a) = Identity (f a)

instance Applicative Identity where
  (Identity f) <*> (Identity x) = Identity (f x)
```

The most trivial example of a `Functor` that is not `Applicative` is `Constant Functor` defined in `Data.Functor.Const` of base as

```
newtype Const a b = Const
  { getConst :: a }

instance Functor (Const a) where
  -- note that it changes type here
  fmap f (Const a) = Const a
  -- so the following would not work
  -- fmap f x = x
```

It is missing a `Pointed` instance. However, if the argument of `Const` is a `Monoid` we can define it as

```
instance Monoid a => Pointed (Const a) where
  pure a = Const mempty

instance Monoid a => Applicative (Const a) where
  Const x <*> Const a = Const (mappend x a)
```

**Remark 1.** One can think of `Applicative f` as representing generalized function application on structure `f`: `pure` lifts pure values into `f` while `(<*>)` provides a way to apply functions to arguments over `f`. Note however, that `Applicative` is not a structure for representing generalized functions (e.g. `Applicative` gives no way to compose functions or to introduce lambdas, unlike the `Monad`, see remark 2).

### 4.1.3 Alternative

`Control.Applicative` module of base [26] defines `Alternative` class as a monoid on `Applicative Functors`.<sup>1</sup>

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a

  -- defined for performance reasons
  some :: f a -> f [a]
  some p = fmap (:) p <*> many p

  many :: f a -> f [a]
  many p = some p <|> pure []
```

requiring monoid laws to hold for `empty` and `(<|>)`

```
-- `empty` is left identity for `(<|>)`,
empty <|> x == x

-- `empty` is right identity for `(<|>)`,
x <|> empty == x

-- `(<|>)` is associative,
x <|> (y <|> z)
== (x <|> y) <|> z

-- and both `some` and `many` are
-- extensionally equal to their
-- default implementations
some p == fmap (:) p <*> many p
many p == some p <|> pure []
```

Combinators `some` and `many`, similarly to `mconcat`, commonly occur in functions handling `Alternatives` and can have different definitions varying in performance for different types. The most common use of `Alternative` type class is parser combinators (section 5.2) where `some` and `many` coincide with `+` (“one or more”) and `*` (“zero or more”, Kleene star) operators from regular expressions/EBNF. Before the introduction of `Alternative` that role was played by now deprecated `MonadPlus` class, currently defined in `Control.Monad` of base as follows

```
class (Alternative m, Monad m) => MonadPlus m where
  mzero :: m a
  mzero = empty

  mplus :: m a -> m a -> m a
  mplus = (<|>)
```

We shall give example instance and usage of `Alternative` in section 5.2.

## 4.2 Purely Monadic

This section describes algebraic structures that involve `Monad` type class and its instances.



### 4.2.1 Monad

`GHC.Base` from `base` [26] defines `Monad` in the following way using the original (i.e. not `Pointed`) hierarchy (also, at the time of writing `base` uses a bit uglier definition which is discussed in section 4.2.3)

```
infixl 1 >>=
class Applicative m => Monad m where
  return  :: a -> m a
  (>>=)   :: m a -> (a -> m b) -> m b
```

and wants its instances to satisfy the following equations known as “`Monad` laws”

```
-- `return` is left identity for `(>>>=)`
return a >>= f == f a

-- `return` is right identity for `(>>>=)`
f >>= return == f

-- `(>>>=)` is associative
(f >>= g) >>= h == f >>= (\x -> g x >>= h)
```

Note that this definition also expects the following additional “unspoken laws” from its parent structures (see section 4.3 for definitions of `liftM` and `ap`).

```
fmap == liftM
pure == return
(<*>) == ap
```

Moreover, the author feels that the name “return” itself is an unfortunate accident since `return` only injects pure values into `m` and does not “return” anywhere. We shall avoid that problem and simplify the above equations by redefining `Monad` using `Pointed` hierarchy instead

```
infixl 1 >>=
class Applicative m => Monad m where
  (>>=)   :: m a -> (a -> m b) -> m b

-- for backward-compatibility
return :: Monad m => a -> m a
return = pure
```

If one is to swap the order of arguments of `(>>=)` then the result is very similar to the type of `( $\$$ )` (which is just a low-infix-priority version of the normal function application operator) and `<*>` operators. Essentially, `(>>=)` is `<*>` that allows to introduce lambdas “outside of context” of `m` but nevertheless keeps results confined to `m`. Compare the following, also see remark 2.

```
( $\$$ )   :: (a -> b) -> a -> b
(<*>)  :: m (a -> b) -> m a -> m b
bind   :: (a -> m b) -> m a -> m b
```

A very common combinator used with `Monads` bears a name of `(>>)` and can be defined as

```
(>>) :: Monad m => m a -> m b -> m b
a >> b = a >>= const b
      -- a >>= \_ -> b
```

The following sections will provide many example instances.

## 4.2.2 MonadFish

A somewhat lesser known but equivalent way to define `Monad` is to define `(>=>)` in “fish” form as follows

```
infixl 1 >=>
class Applicative m => MonadFish m where
  (>=>)  :: (a -> m b) -> (b -> m c) -> (a -> m c)
```

This way `Monad` laws become `Monoid` laws

```
-- `pure` is left identity for `(>=>)`
pure >=> f == f

-- `pure` is right identity for `(>=>)`
f >=> pure == f

-- `(>=>)` is associative
(f >=> g) >=> h == f >=> (g >=> h)
```

Both definitions of `Monad` are known to be equivalent in the folklore, but the author could not find a reference with a simple proof of that fact, hence this section shall give one.

**Lemma 1.**  $(f \gg g) . h == (f . h) \gg g$

*Proof.* For pure values `(>=>)` is a composition with flipped order of arguments (`.`)

```
instance MonadFish Identity where
  f >=> g = g . runIdentity . f
```

In other words,  $f \gg g == g . f$ , which gives the following

```
(f >=> g) . h == h >=> (f >=> g)
              == (h >=> pure) >=> (f >=> g)
              == ((h >=> pure) >=> f) >=> g
              == (h >=> f) >=> g
              == (f . h) >=> g
```

which, with some abuse of notation (`(>=>)` is not heterogeneous, the above lifts pure values into `m` with `pure`), can be written simply as

```
(f >=> g) . h == h >=> (f >=> g)
              == (h >=> f) >=> g
              == (f . h) >=> g
```

□

**Lemma 2.** *Monad and MonadFish define the same structure.*

*Proof.* The cross-definitions:

```
instance (Applicative m, Monad m) => MonadFish m where
  f >=> g = \a -> (f a) >>= g -- (1)

instance {-# OVERLAPPABLE #-}
  (Applicative m, MonadFish m) => Monad m where
  ma >>= f = (id >=> f) ma -- (2)
```

- (1) implies (2):

```
ma >>= f == (id >=> f) ma
          == (\a -> id a >>= f) ma
          == ma >>= f
```

- (2) implies (1):

```
f >=> g == \a -> (f a) >>= g
        == \a -> (id >=> g) (f a)
        == (id >=> g) . f
        == (id . f) >=> g
        == f >=> g
```

□

**Remark 2.** Note that while *Applicative* is too weak to express generalized functions (remark 1), *Monad*, in some sense, is too strong since ( $>=>$ ) and, thus, ( $>>=$ ) combine function composition/application (the whole type) with lambda introduction “outside of context”.

What is the “just right” structure for representing a generalized function is a matter of debate: some would state “an *Arrow!*” [43], others “a (Cartesian Closed) *Category!*” [22], yet others might disagree with both.

### 4.2.3 Monad’s fail and MonadFail

Section 4.2.1 did not give the complete definition of *Monad* as is defined in the current version of `base` [26]. Current `GHC.Base` module defines *Monad* in the following way using the original (not *Pointed*) hierarchy

```
infixl 1 >>=
class Applicative m => Monad m where
  return  :: a -> m a
  (>>=)   :: m a -> (a -> m b) -> m b

  fail    :: String -> m a
  fail s  = error s
```

Note the definition of the `fail` operation. That function is invoked by the compiler on pattern match failures in `do`-expressions (see section 4.7 for examples, see section 4.5.6 for the definition of `error`), but it can also be called explicitly by the programmer in any context where the type permits to do so.

The presence of `fail` in *Monad* class is, clearly<sup>2</sup>, a hack. There is an ongoing effort (aka “*MonadFail* proposal”, “MFP”) to move this function from *Monad* to its own type class defined as follows (in both hierarchies)

```
class Monad m => MonadFail m where
  fail :: String -> m a
  fail s = error s
```

As of writing of this work the new class is available from `Control.Monad.Fail`, but `fail` from the original *Monad* is not even deprecated yet. We shall use *MonadFail* instead of the original `fail` in our hierarchy for simplicity.

---

<sup>2</sup> It involves an error handling mechanism that is more complicated than the thing itself. It creates semantic discrepancies (e.g. `Maybe` is not equivalent to `Either ()`), see section 4.2.6).

#### 4.2.4 Identity Monad

We can define the following `Monad` and `MonadFail` instances for the `Identity Functor`

```
instance Monad Identity where
  (Identity x) >>= f = f x

instance MonadFail Identity where
  -- default implementation
```

despite this instance it is still usually referenced as “`Identity Functor`” even though it is also an `Applicative` and a `Monad`.

#### 4.2.5 Maybe Monad

The simplest form of `Monadic` error handling (that is, not just “error ignoring”) can be done with `Maybe` data type and its `Monad` instance defined in `Data.Maybe` of base [26] equivalently to

```
data Maybe a = Nothing | Just a

instance Pointed Maybe where
  pure = Just

instance Monad Maybe where
  (Just x) >>= k = k x
  Nothing >>= _ = Nothing

instance MonadFail Maybe where
  -- custom `fail`
  fail _ = Nothing
```

The `pure` operator simply injects a given value under `Just` constructor, while the definition of (`>>=`) ensures that

- injected values are transparently propagated further down the computation path,
- computation stops as soon as the first `Nothing` gets emitted.

In other words, `Maybe Monad` is `Identity Monad` that can stop its computation on request. A couple of examples follow

```
maybeTest1 :: Maybe Int
maybeTest1 = do
  x <- Just 1
  pure x

maybeTest2 :: Maybe Int
maybeTest2 = do
  x <- Just 1
  pure x
  Nothing
  Just 2

maybeTest = maybeTest1 == Just 1
            && maybeTest2 == Nothing
```

## 4.2.6 Either Monad

`Either` data type is defined in `Data.Either` of base [26] equivalently to

```
data Either a b = Left a | Right b

instance Pointed (Either e) where
  pure = Right

instance Monad (Either e) where
  Left l >>= _ = Left l
  Right r >>= k = k r

instance MonadFail (Either e)
  -- default `fail`
```

`Either` is a computation that can stop and report a given value (the argument of `Left`) when falling out of `Identity` execution. The intended use is similar to `Maybe`

```
eitherTest1 :: Either String Int
eitherTest1 = do
  x <- Right 1
  pure x

eitherTest2 :: Either String Int
eitherTest2 = do
  x <- Right 1
  pure x
  Left "oops"
  Right 2

eitherTest = eitherTest1 == Right 1
            && eitherTest2 == Left "oops"
```

Purely by its data type definition `Maybe a` is isomorphic to `Either () a` (where `()` is Haskell's name for the ML's `unit` type and type-theoretic “top” type), but their `Monad` instances (in the original hierarchy, `MonadFail` in our hierarchy) differ: `Maybe` has non-default `fail`, while `Either` does not. This produces some observable differences discussed in section 4.7.

## 4.3 An Intermission on Monadic Boilerplate

Haskell does not support default definitions for functions in superclasses that use definitions given in subclasses. That is, Haskell has no syntax to define `Functor` and `Applicative` defaults from `Monad` instance of the same type.

Which is why to compile the code above we have to borrow a couple of functions from `Control.Monad` of base

```
liftM :: (Monad m)
      => (a -> b) -> m a -> m b
liftM f ma = ma >>= pure . f

ap :: (Monad m)
   => m (a -> b) -> m a -> m b
ap mf ma = mf >>= \f -> liftM f ma
```

and use them to define

```
instance Functor Maybe where
  fmap = liftM

instance Applicative Maybe where
  (<*>) = ap
```

and analogously for `Either`. For all the listings that follow we shall silently hide this type of boilerplate code from the document version where appropriate (it can still be observed in the Literate Haskell version).

## 4.4 MonadTransformers

The problem with `Monads` is that they, in general, do not compose. `Monad` transformers [68] provide a systematic way to define structures that represent “a `Monad` with a hole” that allow computations from an inner `Monad` `m` to be `lifted` through a hole in an outer `Monad` `(t m)` (`t` transforms `monad m`, hence “`monad transformer`”). The main type class is defined in `Control.Monad.Trans.Class` module of `transformers` [31] package as follows

```
class MonadTrans t where
  lift :: (Monad m) => m a -> t m a
```

Haskell type class system is not flexible enough to encode the requirement that `t m` needs to be a `Monad` in a single definition, so it has to be encoded in every instance by using the following instance schema

```
instance Monad m => Monad (t m) where
  -- ...
```

Different `MonadTransformers` (`t1`, `t2` ... `tn`) can then be composed with an arbitrary `Monad` `m` (usually called “*the inner Monad*”) using the following scheme

```
newtype comp m a = t1 (t2 (. (tn (m a))))
```

and the whole composed stack would get a `Monad` instance inferred for it. Popular choices for the inner `Monad` `m` include `Identity Functor` and `IO Monad` (see section 4.5).

In short, `MonadTransformers` are, pretty much, composable `Monadic` structures. The following sections will provide many example instances. For an in-depth tutorial readers are referred to [49] and [68].

### 4.4.1 Identity

The simplest `MonadTransformer` is `IdentityT` defined in `Control.Monad.Trans.Identity` of `transformers` [31] package equivalently to

```
newtype IdentityT m a = IdentityT
  { runIdentityT :: m a }

instance MonadTrans IdentityT where
  lift = IdentityT

instance Monad m
  => Pointed (IdentityT m) where
  pure = lift . pure
```

```
instance Monad m
  => Monad (IdentityT m) where
  x >>= f = IdentityT $ do
    v <- runIdentityT x
    runIdentityT (f v)
```

**Remark 3.** Note that *IdentityT MonadTransformer* is different from *Identity Monad* and cannot be redefined as simply

```
type IdentityT' m a = Identity (m a)
```

(even though the data type definition matches exactly) because *IdentityT* “inherits” *Monad* implementation from its argument *m* while *Identity* provides its own. I.e. *IdentityT* is an identity on *MonadTransformers* while *Identity* is an identity on types.

In particular, for *Identity (Maybe a)*

```
pure == Identity
```

```
while for IdentityT Maybe a
```

```
pure == IdentityT . pure == IdentityT . Just
```

#### 4.4.2 Maybe

Transformer version of *Maybe* called *MaybeT* is defined in `Control.Monad.Trans.Maybe` from `transformers` [31] package equivalently to

```
newtype MaybeT m a = MaybeT
  { runMaybeT :: m (Maybe a) }
```

```
instance MonadTrans MaybeT where
  lift = MaybeT . liftM Just
```

```
instance Monad m
  => Pointed (MaybeT m) where
  pure = lift . pure
```

```
instance Monad m
  => Monad (MaybeT m) where
  x >>= f = MaybeT $ do
    v <- runMaybeT x
    case v of
      Nothing -> pure Nothing
      Just y   -> runMaybeT (f y)
```

```
instance MonadFail m
  => MonadFail (MaybeT m) where
  fail _ = MaybeT (pure Nothing)
```

#### 4.4.3 Except

Transformer version of *Either* for historical reasons bears a name of *ExceptT* and is defined in `Control.Monad.Trans.Except` from `transformers` [31] package equivalently to

```

newtype ExceptT e m a
  = ExceptT { runExceptT
              :: m (Either e a) }

instance MonadTrans (ExceptT e) where
  lift = ExceptT . liftM Right

instance Pointed m
  => Pointed (ExceptT e m) where
  pure a = ExceptT $ pure (Right a)

instance Monad m
  => Monad (ExceptT e m) where
  m >>= k = ExceptT $ do
  a <- runExceptT m
  case a of
    Left e -> pure (Left e)
    Right x -> runExceptT (k x)

instance MonadFail m
  => MonadFail (ExceptT e m) where
  fail = ExceptT . fail

```

The main attraction of `ExceptT` for the purposes of this work is the fact that it provides its own non-imprecise non-dynamic-dispatching `throw` and `catch` operators defined as

```

throwE :: (Monad m) => e -> ExceptT e m a
throwE = ExceptT . pure . Left

catchE :: (Monad m) =>
  ExceptT e m a
  -> (e -> ExceptT f m a)
  -> ExceptT f m a
m `catchE` h = ExceptT $ do
  a <- runExceptT m
  case a of
    Left l -> runExceptT (h l)
    Right r -> pure (Right r)

```

There also exists deprecated `ErrorT` (defined in `Control.Monad.Trans.Error` from `transformers` package) which at the time of writing has exactly the same definition as `ExceptT`

```

newtype ErrorT e m a
  = ErrorT { runErrorT
             :: m (Either e a) }

```

but its instances require type class `Exception` (see section 4.5.4) from its argument `e`. Older versions of `transformers` package made this requirement in the definition of `ErrorT`

```

newtype ErrorT e m a
  = Exception e =>
  ErrorT { runErrorT
          :: m (Either e a) }

```

but that mechanism itself was deprecated awhile ago.



#### 4.4.4 Reader

`Reader Monad` is defined in `Control.Monad.Trans.Reader` module of `transformers` [31] package equivalently to

```
type Reader s = ReaderT s Identity

newtype ReaderT s m a = ReaderT { runReaderT :: s -> m a }

instance MonadTrans (ReaderT s) where
  lift m = ReaderT $ \_ -> m

instance Pointed m => Pointed (ReaderT s m) where
  pure a = ReaderT $ \_ -> pure a

instance Monad m => Monad (ReaderT s m) where
  m >>= k = ReaderT $ \s -> do
    a <- runReaderT m s
    runReaderT (k a) s

instance MonadFail m => MonadFail (ReaderT s m) where
  fail str = ReaderT $ \_ -> fail str
```

Essentially, `type Reader s a = s -> a`, thus `Reader r` is just a “function from `r`”. Its `Pointed` instance simply wraps a given value into a constant function while (`>>=`), essentially, does `Identity` computations while chaining `r` around. In other words, `Reader Monad` represents computations in immutable global context (e.g. think environment variables provided by the OS).

#### 4.4.5 State

`State Monad` is defined in `Control.Monad.Trans.State.Lazy` and `Control.Monad.Trans.State.Strict` modules (the difference between them does not matter for the purposes of this work, so we shall ignore it) from `transformers` [31] package equivalently to

```
newtype StateT s m a = StateT { runStateT :: s -> m (a, s) }

type State s = StateT s Identity
-- or, beta-equivalently
-- newtype State s a = State { runState :: s -> (a, s) }

instance MonadTrans (StateT s) where
  lift m = StateT $ \s -> do
    a <- m
    pure (a, s)

instance Pointed m => Pointed (StateT s m) where
  pure a = StateT $ \s -> pure (a, s)

instance Monad m => Monad (StateT s m) where
  m >>= k = StateT $ \s -> do
    (a, s') <- runStateT m s
    runStateT (k a) s'

instance MonadFail m => MonadFail (StateT s m) where
  fail str = StateT $ \_ -> fail str
```

Essentially, `type State s a = s -> (a, s)`. As we seen above, `ReaderT` simply applies variable `s` throughout its whole computation via its (`>>=`) operator thus supplying computations with a global immutable context, that is, immutable state. Meanwhile, `StateT` chains its `s` between computations, thus providing computations with a mutable state.

```
get :: State s s
get = StateT $ \s -> pure (s, s)

put :: s -> State s ()
put s = StateT $ \_ -> pure ((), s)
```

## 4.5 Imprecise Exceptions

GHC implements *imprecise exceptions* mechanism proposed in [90]. Such exceptions look superficially similar to those of C++/Java/Python/etc but differ in two important aspects.

Firstly, GHC imprecise exceptions in pure computations are completely imprecise. That is, evaluation of `(a `op` b)` with `a` raising `e` and `b` raising `f` (and assuming `op` can evaluate either argument first) can raise either or even both (on different evaluations) of `e` and `f`. Haskell is not the only language that does this, C++, for instance, defines *sequence points* that serve the same purpose [15]. However, in GHC the order in which exception are raised is limited only by data dependencies, while C++'s sequence points add some more ordering on top.

Secondly, the C++/Java/Python exceptions have dynamic dispatch builtin, while GHC's dynamically dispatched exceptions are implemented as a library on top of statically dispatched exceptions. To be more specific

- on the base level GHC runtime defines `raise#` and `catch#` operations for which `raise#` “simply”<sup>3</sup> unwinds the stack to the closest `catch#` (i.e. `raise#` is “just”<sup>3</sup> a GOTO; casting, re-raising, `finally`, etc are left for the libraries to implement and are not builtins),
- on top of that GHC libraries then provide dynamically dispatched exceptions by casting elements of `Typeable` types from/to `SomeException` existential type [72].

In the following sections we shall discuss the details of the actual implementation.

### 4.5.1 IO

GHC defines the mystical `IO Monad` in `GHC.Types` (the types) and `GHC.Base` (the instances), pretty much, as a `State Monad` (see section 4.4.5) on `State# RealWorld` (definitions of both of which are beyond the scope of this work)

```
type IO# a = State# RealWorld
    -> (# State# RealWorld, a #)

newtype IO a = IO { runIO :: IO# a }

instance Pointed IO where
    pure a = IO $ \s -> (# s, a #)

instance Monad IO where
    m >>= f = IO $ \s -> case runIO m s of
        (# s', a #) -> runIO (f a) s'
```

---

<sup>3</sup> We put “simply” and “just” into quotes since unwinding of the stack must unwind into the lexically correct handler which is nontrivial in a lazy language like Haskell where thunks can be evaluated in an environment different from the one they were created in. In short, thunks must capture exception handlers as well as variables.

The `IO#` definition given above is not actually in GHC but without it all of the definitions below become unreadable. We also renamed `unIO` to `runIO` for uniformity with `State`. Note however, that we did not swap the elements of the result tuple of `IO#` to match those of `State` since that would make it incompatible with GHC runtime we reuse in Literate Haskell version.

**Remark 4.** *Note that `IO` is not a proper `Monad` since it cannot satisfy the laws simply for the fact that `RealWorld` cannot have an equality.*<sup>4</sup>

*In this work, however, for the purposes of formal arguments involving `IO` we shall treat `IO` as if it was just a `State` over some state type with some simple denotational semantics (although, possibly unknown value). This, of course, immediately disqualifies our proofs for `IO` from using non-determinism, hence, for instance, we will not be able to prove things about imprecise exceptions or threads.*

*The alternative would be to split every lemma and theorem mentioning `IO` into two: one for a `RawMonad` (`Monad` without laws) for cases mentioning `IO`, and one for `Monad` for all other cases. This would make a very little practical sense for this work since we will not attempt proofs involving non-determinism anyway.*

## 4.5.2 `raise#` and `catch#`

Primitive `raise#` and `catch#` operations are “defined” (those, of course, are just stubs to be replaced by references to the actual implementations in GHC runtime) in `GHC.Prim` module like follows

```
raise# :: a -> b
raise# = raise#

catch# :: IO# a -> (b -> IO# a)
        -> IO# a
catch# = catch#
```

Evaluating `raise#` “simply”<sup>3</sup> unwinds computation stack to the point of the closet `catch#` with the appropriate type and applies raised value to the second argument of the latter. Note, however, that while the type of `raise#` permits its use anywhere in the program, `catch#` is sandboxed to `IO#` on the lowest observable level and GHC provides no “`unsafeCatch`”. This allows GHC to perform many useful optimizations that influence evaluation order without exposing pure computations to non-determinism.

## 4.5.3 `Typeable`

GHC implements dynamic casting with `Typeable` type class. The details of its actual implementation are beyond the scope of this work. For our purposes it suffices to say that it is a type class of types that have type representations that can be compared at runtime

```
class Typeable a where
  -- magic beyond the scope of this work
```

and it provides a `cast` operation with the following type signature that shows that it compares said representations of types of its argument and result and either returns its argument value wrapped in `Just` constructor when the types match or `Nothing` else

```
cast :: forall a b
      . (Typeable a, Typeable b)
      => a -> Maybe b
```

Interested readers should inspect the source code of `Data.Typeable` module of `base` [26].

---

<sup>4</sup> Although `IO` can be reformulated as a free `Monad` made of “requests to the interpreter” and continuations if one is willing to forget about the internal structure of the `RealWorld` [57].

#### 4.5.4 Exception

On top of `Typeable` in `GHC.Exception` module of base [26] GHC provides the `Exception` type class that casts values to and from `SomeException` existential type (the following syntactic `forall` is type-theoretic `exists`, historic reasons)

```
data SomeException = forall e . Exception e
                    => SomeException e

class (Typeable e, Show e) => Exception e where
  toException    :: e -> SomeException
  fromException  :: SomeException -> Maybe e

  toException = SomeException
  fromException (SomeException e) = cast e

instance Show SomeException where
  show (SomeException e) = show e

instance Exception SomeException where
  toException = id
  fromException x = Just x
```

#### 4.5.5 throw and catch

Finally, `throw` and `catch` operators defined in `GHC.Exception` module of base [26] use all of the above to implement dynamic dispatch of exceptions.

The `throw` operator simply wraps given exception into `SomeException` and `raise#s`

```
throw :: Exception e => e -> a
throw e = raise# (toException e)
```

The `catchException` operator defined in `GHC.IO` does the actual dynamic dispatch

- it `catch#s` an exception produced by its first argument (“computation”),
- tries to `cast` it to a type expected by its second argument (“handler”) and either calls the latter on success, or `raise#s` (actually `raiseIO#s`, since its a precise exception, this will be discussed in section 4.6) again on failure.

```
catchException :: Exception e
                => IO a -> (e -> IO a)
                -> IO a

catchException (IO io) handler
= IO $ catch# io handler'
  where
    handler' e = case fromException e of
      Just f -> runIO (handler f)
      Nothing -> raiseIO# e
```

The `catch` operator simply calls `catchException` after forcing its first argument into a thunk with `lazy` operator (this wrapping is necessary to prevent GHC from performing strictness analysis on the “computation”; this fact can be ignored for the purposes of this work) which is yet another special GHC runtime function (this time, extensionally equal to its definition, i.e. identity).

```

lazy :: a -> a
lazy x = x

catch  :: Exception e
      => IO a -> (e -> IO a)
      -> IO a
catch act = catchException (lazy act)

```

That is, `catch` is extensionally equal to `catchException`. `Control.Exception` module of `base` simply reexports `throw`, `catch`, and `Exception` type class and implements a bunch of practically convenient combinators using them.

We should also mention that older versions of `base` package had another special `catch` that handled only `IOErrors` defined in `Prelude` and `System.IO.Error` respectively. Those were deprecated in 2011 and as of writing of this work are completely gone from current version of `base`. But they are occasionally mentioned in tutorials, usually in the context of “don’t use `catch` from `Prelude`, use the one from `Control.Exception`”, nowadays the `catch` from `Prelude` *is* the `catch` from `Control.Exception`.

#### 4.5.6 error and undefined

`error` and `undefined` primitives are defined in `GHC.Err` of `base` as follows

```

newtype ErrorCall = ErrorCall String

instance Exception ErrorCall where

error :: String -> a
error s = throw (ErrorCall s)

undefined :: forall a . a
undefined = error "Prelude.undefined"

```

Actually, this implementation is taken from the older version of `base`, modern version also implements call stack capture, which is beyond the scope of this work. Interested readers are referred to the source code of `GHC.Err`.

## 4.6 Precise raiseIO# and throwIO

Besides imprecise exceptions GHC’s `IO` also has operators for precise exceptions a-la `ExceptT` defined in `GHC.Prim` and `GHC.Exception` as follows

```

raiseIO# :: a -> IO# b
raiseIO# = raiseIO#

throwIO :: Exception e => e -> IO a
throwIO e = IO $ raiseIO# (toException e)

```

While `throwIO` has a type that is an instance of `throw`, their semantics differ: `throwIO` produces `Monadic` actions while `throw` produces values. For example, both functions in the following example will raise `SomethingElse`, not `ErrorCall`.

```

data SomethingElse = SomethingElse

instance Exception SomethingElse where

```

```

throwTest :: IO ()
throwTest = do
  let x = throw (ErrorCall "lazy")
  pure (Right x)
  throwIO SomethingElse

```

```

throwTest' :: IO ()
throwTest' = do
  let x = throw (ErrorCall "lazy")
  pure x
  throwIO SomethingElse

```

The `catch` operator, however, can be reused for handling both imprecise and precise exceptions.

**Remark 5.** *In other words, we can say that `IO` has two different exception mechanisms (precise and imprecise exceptions) with a single exception handling mechanism (`catch`). (And this is pretty weird.)*

## 4.7 Non-exhaustive Patterns

As a side note, non-exhaustive pattern matches (and `cases`) throw `PatternMatchFail` exception, while the default `fail` implementation calls `error` which throws `ErrorCall`.

```

{-# LANGUAGE ScopedTypeVariables #-}

import Control.Exception

check t =
  (evaluate t >> print "ok")
  `catch`
  (\(e :: PatternMatchFail)
   -> print "throws PatternMatchFail")
  `catch`
  (\(e :: ErrorCall)
   -> print "throws ErrorCall")

patFail 1 x = case x of 0 -> 1
fail1 = patFail 1 1
fail2 = patFail 2 2
maybeDont = do { 1 <- Just 1 ; return 2 }
maybeFail = do { 0 <- Just 1 ; return 2 }

-- These are GHC < 8.6 only, GHC 8.6 uses MonadFail
eitherDont = do { 1 <- Right 1 ; return 2 }
eitherFail = do { 0 <- Right 1 ; return 2 }

testPatterns = do
  check fail1      -- throws PatternMatchFail
  check fail2      -- throws PatternMatchFail
  check maybeDont  -- ok
  check maybeFail  -- ok (`Nothing`)
  check eitherDont -- ok
  check eitherFail -- throws ErrorCall

```

## 4.8 Monad Generalizations

In previous sections we have seen a plethora of slightly different error handling structures with different `throw` and `catch` operators. In this section we shall describe several Hackage packages that provide structures that try to unify this algebraic zoo.

### 4.8.1 MonadError

`MonadError` class (`Control.Monad.Error.Class` from `mtl` [30] package) is defined as

```
class (Monad m) => MonadError e m
  | m -> e where
  throwError :: e -> m a
  catchError :: m a
             -> (e -> m a) -> m a
```

This structure simply generalizes `ExceptT`

```
instance Monad m => MonadError e (ExceptT e m) where
  throwError = throwE
  catchError = catchE
```

in a way that is transitive over many other `MonadTransformers`, for instance

```
-- (these require UndecidableInstances GHC extension, however)
```

```
instance MonadError e m => MonadError e (IdentityT m) where
  throwError = lift . throwError
  catchError a h = IdentityT $ catchError (runIdentityT a) (runIdentityT . h)
```

```
instance MonadError e m => MonadError e (MaybeT m) where
  throwError = lift . throwError
  catchError a h = MaybeT $ catchError (runMaybeT a) (runMaybeT . h)
```

### 4.8.2 MonadThrow and MonadCatch

`MonadThrow` and `MonadCatch` classes (`Control.Monad.Catch` from `exceptions` [59]) are defined as<sup>5</sup>

```
class Monad m => MonadThrow m where
  throwM :: Exception e => e -> m a
```

```
class MonadThrow m => MonadCatch m where
  catchM :: Exception e
         => m a -> (e -> m a) -> m a
```

These two structures, too, generalizes `ExceptT`

```
instance MonadThrow m => MonadThrow (ExceptT e m) where
  throwM = lift . throwM
```

```
instance MonadCatch m => MonadCatch (ExceptT e m) where
  catchM x f = ExceptT $ catchM (runExceptT x) (runExceptT . f)
```

---

<sup>5</sup> Except for the fact that `MonadCatch` from `exceptions` names its operator `catch`, not `catchM`, we renamed it for uniformity and so that it would not be confused with the operator from `Control.Exception`.

and they, too, are transitive over common `MonadTransformers`

```
-- (this time without UndecidableInstances)

instance MonadThrow m => MonadThrow (IdentityT m) where
  throwM = lift . throwM

instance MonadCatch m => MonadCatch (IdentityT m) where
  catchM x f = IdentityT $ catchM (runIdentityT x) (runIdentityT . f)

instance MonadThrow m => MonadThrow (MaybeT m) where
  throwM = lift . throwM

instance MonadCatch m => MonadCatch (MaybeT m) where
  catchM x f = MaybeT $ catchM (runMaybeT x) (runMaybeT . f)
```

but they constrain their argument `e` to the `Exception` type class, and they also generalize the imprecise exceptions

```
instance MonadThrow IO where
  throwM = throw

instance MonadCatch IO where
  catchM = catch
```

The latter fact complicates their use somewhat since one can not be sure about the dynamic-dispatch part of the semantics without actually looking at the definitions for a particular instance.



## Chapter 5

# Introduction to Some Non-basic Structures of Haskell

This section, logically, is a continuation of chapter 4. However, in contrast to that section this section discusses non-basic structures that are of particular importance to the rest of the work. While this section does not introduce any non-trivial novel ideas, some perspectives on well-known ideas seem to be novel.

### 5.1 Continuations

When speaking of “continuations” people usually mean one or more of the three related aspects explained in this section.

#### 5.1.1 Continuation-Passing Style

Any (sub)program can be rewritten into Continuation-Passing Style (CPS) [2, 94] by adding a number of additional *continuation* arguments to every function and tail-calling into those arguments with the results-to-be at every return point instead of just returning said results.

For instance, the following pseudo-Haskell program

```
foo =
  if something
    then Result1 result1
    else Result2 result2

bar = case foo of
  Result1 a -> bar1 a
  Result2 b -> bar2 b
```

can be transformed into (here we CPS-ignore `something` and the `if` for illustrative purposes)

```
fooCPS cont1 cont2 =
  if something
    then cont1 result1
    else cont2 result2

barCPS = fooCPS bar1 bar2
```

In conventional modern low-level imperative terms this transformation requires all functions to receive their return addresses as explicit parameters instead of popping them from the bottom of their stack frame.

The latter, of course, means that we can treat “*normal*” programs (in which all functions have a single return address) as a degenerate case of programs written in “*implicit-CPS*” (in fact, `Cont Monad` of section 5.1.3 is exactly such an “*implicit-CPS*”) — a syntactic variant of CPS in which

- every function has an implicit argument that specifies a default return address (which is set to the next instruction following a corresponding function call by default)
- that can be reached from the body of the function by tail-calling a special symbol that `jumps` to the implicitly given address.

Finally, one can even imagine a computer with a “*CPS-ISA*” (i.e. an ISA where each instruction explicitly specifies its own return address) in which case all programs for such a computer would have to be translated into an explicit CPS form to be executed. In fact, drum memory-based computers like IBM 650 had exactly such an ISA. From the point of view of an IBM 650 programmer modern conventional CPUs simply convert their non-CPS Opcodes into their CPS forms on the fly, thus applying machine instruction level CPS-transform to any given program on the fly.

Returning to the pseudo-Haskell listing above, note that programs written in CPS

- introduce a linear order on their computations, hence they are not particularly good for parallel execution,
- consume somewhat more memory in comparison to their “*normal*” representations (as they have to handle more explicit addresses),
- can have poorer performance on modern conventional CPUs (since said CPUs split their branch predictors into “`jump`” and “`call`” units and the latter unit rests completely unused by CPS programs),
- are harder to understand.

However, the advantage of the CPS form is that it allows elimination of duplicate computations. For instance, in the example above `foo` produces different results depending on the value of `something` and `bar` has to duplicate that choice (but not the computation of `something`) again by switching `cases` on the result of `foo`. Meanwhile, `barCPS` is free from such an inefficiency. Applying this transformation recursively to a whole (sub)program allows one to transform the (sub)program into a series of tail calls whilst replacing all constructors and eliminators in the (sub)program with tail calls to newly introduced continuation arguments and `case` bodies respectively.

The logical mechanic behind this transformation is a technique we call *generalized Kolmogorov’s translation* (since it is a trivial extension of Kolmogorov’s translation [63]) of types of functions’ results. That is, double negation followed by rewriting by well-known isomorphisms until formula contains only arrows, bottoms and variables followed by generalizing bottoms by a bound variable.

For instance, the result of a function of type

$$i \rightarrow j \rightarrow b$$

is `b`, which can be doubly negated as

$$\begin{aligned} & \neg\neg b \\ & (b \rightarrow \perp) \rightarrow \perp \end{aligned}$$

and generalized to either of

$$\begin{aligned} & \forall c.(b \rightarrow c) \rightarrow c \\ & \lambda c.(b \rightarrow c) \rightarrow c \end{aligned}$$

which allows us to generalize the whole function to either of

$$\begin{aligned} \text{former} &= \forall c.i \rightarrow j \rightarrow (b \rightarrow c) \rightarrow c \\ \text{latter} &= \lambda c.i \rightarrow j \rightarrow (b \rightarrow c) \rightarrow c \end{aligned}$$

depending on the desired properties:

- the former term requires a rank-2 type system but it does not add any new type lambdas or free type variables, thus keeping the transformation closed,
- the latter term does not need rank-2 types, but it requires tracking of these new type variables,
- the latter term also retains full control over  $c$  variable, (for instance, it can produce the former term in rank-2 type system on demand with  $\forall c.latter\ c$ ).

Similarly, `Either a b` may be seen as logical  $a \vee b$  which can be rewritten as

$$\begin{aligned} & \neg\neg(a \vee b) \\ & \neg(\neg a \wedge \neg b) \\ & (a \rightarrow \perp \wedge b \rightarrow \perp) \rightarrow \perp \\ & (a \rightarrow \perp) \rightarrow (b \rightarrow \perp) \rightarrow \perp \end{aligned}$$

and a pair of  $(a, b)$  is logical  $a \wedge b$  and can be rewritten as

$$\begin{aligned} & \neg\neg(a \wedge b) \\ & \neg(a \wedge b) \rightarrow \perp \\ & (a \wedge b \rightarrow \perp) \rightarrow \perp \\ & (a \rightarrow b \rightarrow \perp) \rightarrow \perp \end{aligned}$$

Hence,  $i \rightarrow j \rightarrow (a \vee b)$  can be rewritten into either of

$$\begin{aligned} & \forall c.i \rightarrow j \rightarrow (a \rightarrow c) \rightarrow (b \rightarrow c) \rightarrow c \\ & \lambda c.i \rightarrow j \rightarrow (a \rightarrow c) \rightarrow (b \rightarrow c) \rightarrow c \end{aligned}$$

and  $i \rightarrow j \rightarrow (a \wedge b)$  into either of

$$\begin{aligned} & \forall c.i \rightarrow j \rightarrow (a \rightarrow b \rightarrow c) \rightarrow c \\ & \lambda c.i \rightarrow j \rightarrow (a \rightarrow b \rightarrow c) \rightarrow c \end{aligned}$$

## 5.1.2 Scott-encoding

A technique of applying generalized Kolmogorov's translation to data types and their constructors and eliminators instead of normal functions in a (sub)program is called Scott-encoding (apparently, Dana Scott did not publish, to our best knowledge the first mention in print is [18, p. 219] and first generic description of the technique for arbitrary data types is [103]).

As before, `Either` can be replaced with either of

$$\begin{aligned} & \forall c.(a \rightarrow c) \rightarrow (b \rightarrow c) \rightarrow c \\ & \lambda c.(a \rightarrow c) \rightarrow (b \rightarrow c) \rightarrow c \end{aligned}$$

which can be encoded in Haskell as either of

```
newtype EitherS a b = EitherS
  { runEitherS
    :: forall c
    . (a -> c) -> (b -> c) -> c }

left :: a -> EitherS a b
left a = EitherS (\ac bc -> ac a)
```

```

right :: b -> EitherS a b
right b = EitherS (\ac bc -> bc b)

newtype EitherS' c a b = EitherS'
  { runEitherS'
    :: (a -> c) -> (b -> c) -> c }

left' :: a -> EitherS' c a b
left' a = EitherS' (\ac bc -> ac a)

right' :: b -> EitherS' c a b
right' b = EitherS' (\ac bc -> bc b)

```

with `runEitherS` (`runEitherS'`) taking the role of an eliminator (`case` operator) and `left` and `right` (`left'` and `right'`) taking the roles of `Left` and `Right` constructors respectively.

Similarly, `(a, b)` can then be generalized to either of

$$\forall c.(a \rightarrow b \rightarrow c) \rightarrow c$$

$$\lambda c.(a \rightarrow b \rightarrow c) \rightarrow c$$

and encoded in Haskell as either of

```

newtype PairS a b = PairS
  { runPairS
    :: forall c
    . (a -> b -> c) -> c }

pair :: a -> b -> PairS a b
pair a b = PairS (\f -> f a b)

newtype PairS' c a b = PairS'
  { runPairS'
    :: (a -> b -> c) -> c }

pair' :: a -> b -> PairS' c a b
pair' a b = PairS' (\f -> f a b)

```

Substituting all `Lefts` with `left`, `Rights` with `right`, `cases` on `EITHERS` with `runEitherS`, `pair` constructions with `pair`, and `cases` on pairs with `runPairS` (and similarly for primed versions) does not change computational properties of the transformed program in the sense that Scott-transformation of the original program's normal form coincides with the normal form of the Scott-transformed program.

Replacing a single data type in a program with its Scott-encoding can be viewed as a kind of selective CPS-transform on those subterms of the program that use the data type. The type of transformed functions changes the same way in both transformations, but Scott-encoding groups all continuation arguments, hides them behind a type alias and introduces a bunch of redundant beta reductions in constructors and eliminators.

The upside of CPS-transforming with Scott-encoding is that it supports partial applications, requires absolutely no thought to perform and no substantial changes to the bodies of the functions that are being transformed. It is also very useful for designing new languages and emulating data types in languages that do not support them<sup>1</sup> as it allows to use data types when none are supported by the core language.

---

<sup>1</sup> For example, most instances of the *visitor* object-oriented design pattern that are not simply emulating `Functor` instances usually emulate pattern matching with Scott-encoding.

The most immediate downside of this transformation is very poor performance on modern conventional CPUs. For instance, pattern matching on `Either` produces a simple short conditional `jmp` while for `runEitherS` the compiler, in general, cannot be sure about value of the arguments (it can be anything of the required type, not only `left` or `right`) and has to produce an indirect `jmp` (or `call` if it is not a tail call) and both `left` and `right` require another indirect `jmp`. This wastes address cache of CPU's branch predictor and confuses it<sup>2</sup> when instruction pointer jumps out of the stack frame.

For some classes of programs, however, it can increase performance significantly. For instance, in a “`case-tower`” like

```
doSomethingOn s = case internally s of
  Right a -> returnResult a
  Left b  -> handleError b

internally s =
  case evenMoreInternally s of
    Right (a,s) -> doSomethingElse a s
    Left b      -> Left b

doSomethingElse a s =
  case evenMoreInternally s of
    Right (a,s) -> Right a
    Left b      -> Left b
```

(which is commonly produced by parser combinators) performing this selective CPS-transform followed by inlining and partial evaluation of the affected functions will replace all construction sites of `Lefts` with direct calls to `handleError`, and `Rights` in `doSomethingElse` (and, possibly, the ones residing in `evenMoreInternally`) with `returnResult`.

In other words, rewriting this type of code using Scott-encoded data types is a way to apply deforestation [109] to it, but semi-manually as opposed to automatically, and with high degree of control. This fact gets used a lot in Hackage libraries, where, for example, most parser combinators (section 5.2) use Scott-encoded forms internally.

### 5.1.3 Cont

One of the roundabout ways to express pure values in Haskell is to wrap them with the `Identity Functor` (section 4.1.2) for which `Identity a`, logically, is just a pure type variable  $a$ . Applying generalized Kolmogorov's translation to this variable gives either of

$$\forall c.(a \rightarrow c) \rightarrow c$$

$$\lambda c.(a \rightarrow c) \rightarrow c$$

In Haskell the latter type is called `Cont`. It is defined in `Control.Monad.Cont` of `mtl` [30] as

```
newtype Cont r a = Cont
  { runCont :: (a -> r) -> r }
```

with the following `Monad` instance

```
instance Pointed (Cont r) where
  pure a = Cont $ \c -> c a

instance Monad (Cont r) where
  m >>= f = Cont $ \c -> runCont m
           $ \a -> runCont (f a) c
```

<sup>2</sup>Note that this does not happen for the full CPS-transform of the previous section since that translation does no calls.

`Cont` has a transformer version defined in `Control.Monad.Trans.Cont` module of `transformers` [31] package as follows

```
newtype ContT r m a = ContT { runContT :: (a -> m r) -> m r }

instance MonadTrans (ContT' r) where
  lift m = ContT (m >>=)
```

Interestingly, however, unlike `Identity` and `IdentityT` which have different `Monad` instances (see section 4.4.1), `Cont` and `ContT` have identical ones (equivalent to the one given above). Of particular note is the fact that the definition of `(>>=)` for `ContT` does not refer to the `Monad` operators of its argument `m`. This means that in cases when we do not need the `MonadTrans` instance (for which we have to have a `newtype` wrapper) we can redefine `ContT` as simply

```
type ContT r m a = Cont (m r) a
```

The latter fact means that `ContT`, unlike other `MonadTransformers` we saw before, is not a “`Monad` transformer” as it is not a functor on category of monads (it is always a `Monad` irrespective of the argument `m`). This property can be explained by the fact that, as we noted at the top of this section, `Cont Monad` is a kind of “*implicit-CPS*” form of computations. Since all it does is chain return addresses it does not care about types of computations those addresses point to.

#### 5.1.4 Delimited callCC

Peirce’s law states that

$$((a \rightarrow b) \rightarrow a) \rightarrow a$$

by applying generalized Kolmogorov’s translation we get

$$\begin{aligned} & \neg\neg(((a \rightarrow b) \rightarrow a) \rightarrow a) \\ & \neg(\neg a \rightarrow \neg((a \rightarrow b) \rightarrow a)) \\ & \neg\neg((a \rightarrow b) \rightarrow a) \rightarrow \neg\neg a \\ & (\neg\neg(a \rightarrow b) \rightarrow \neg\neg a) \rightarrow \neg\neg a \\ & ((\neg\neg a \rightarrow \neg\neg b) \rightarrow \neg\neg a) \rightarrow \neg\neg a \end{aligned}$$

which can be encoded in Haskell as (note that this time we use  $\forall$  variant of the translation)

```
peirceCC :: ((Cont r a -> Cont r b) -> Cont r a)
          -> Cont r a
peirceCC f = Cont $ \c ->
  runCont (f (\ac -> Cont $ \_ -> runCont ac c)) c
```

This operator takes a function `f`, applies some magical subterm to it and then gives it its own return address. That is, for a function `f` that ignores its argument `peirceCC` is completely transparent. The magical argument `peirceCC` applies to `f` is itself a function that takes a computation producing value of the same type `f` returns as a result. The subterm then computes the value of the argument but ignores its own return address and continues to the return address given to `peirceCC` instead. In other words, `peirceCC` applies `f` with an *escape continuation* which works exactly like a `return` statement of conventional imperative languages (as opposed to `Monad`’s `pure` which should not be called “`return`”, see section 4.2.1).

Note that `ac` argument to the magical subterm is pretty boring: it is a computation that gets computed immediately. Hence, unless we require every subterm of our program to be written in *implicit-CPS* form we can simplify `peirceCC` a bit as follows

```
callCC :: ((a -> Cont r b) -> Cont r a) -> Cont r a
callCC f = Cont $ \c ->
  runCont (f (\a -> Cont $ \_ -> c a)) c
```

This operator bears a name of “delimited `call/cc` (`callCC`)” [3] and the escape continuation it supplies to `f` not only works but also looks exactly like an imperative `return` (in that it takes a pure value instead of a computation producing it).

### 5.1.5 Scheme’s `call/cc` and ML’s `callcc`

Note that delimited `callCC` is semantically different from similarly named operators of SML [102] and Scheme [101]. SML defines its operator as

```
type 'a cont
val callcc : ('a cont -> 'a) -> 'a
```

where `'a cont` type is the type of the *current global continuation* which is the computation till the end of the whole program, this type is a kind of technical alias for what, logically, should be  $a \rightarrow b$ , i.e. `callcc`’s type, logically, is non-Kolmogorov-translated Peirce’s law.

The difference is that by applying Kolmogorov’s translation to Peirce’s law `callCC` gains intuitionistic witnesses (and, hence, purely functional implementations) and becomes *delimited* by the current `Cont` context instead of the whole program. Meanwhile, implementations of non-delimited `callcc` and `call/cc` require special support from the compiler/interpreter and Kiselyov [54] eloquently advocates that they simply should not exist as they are *less* useful than their delimited versions and their implementations introduce nontrivial trade-offs to the languages in question.

## 5.2 Monadic Parser Combinators

Parser combinators, as their name suggests, are combinators (closed terms) that are designed with the purpose of building parsers (functions from `Streams` to trees/structured data) by combining simpler parsers into progressively more and more complex ones. The resulting parsers usually have worse performance than those produced by parser generators (e.g. Yacc [48] and Bison [17]), but parsers expressed using parsers combinators can be much simpler to understand as they are commonly built using high-level declarative descriptions (since such parsers are first-class objects of the language, all the usual compositional powers of the target language apply), unlike for the usual way of describing grammars with EBNF and similar and then generating an opaque parser for the target language in a single step.

In the context of error handling, while parser combinators are not by themselves a general error handling mechanism, in practice, parsers frequently need to handle failed parsing attempts using rather complex strategies, thus parser combinator libraries usually provide rather elaborate error handling mechanisms.

**Monadic** parser combinators, as their name suggests, are parser combinators that also form a **Monad**. The most popular **Monadic** parser combinator libraries for Haskell are Parsec [66, 67], Attoparsec [88], and Megaparsec [51].

In general, such structures can possess a wide variety of semantics and implementations, to mention just a few possible dimensions of the space:

- they can parse various classes of parsing grammars (PEG [25],  $LL\infty$ , etc; admitting left recursion, or not),
- they can either automatically backtrack on errors or keep the state as is,
- they can distinguish not only successful and failed parsing attempts but also attempts that consumed none of the input and those that consumed at least one element of the input [67],

- they can support simple **Strings**, lists of arbitrary elements, or impure **Streams** (in general, a **Stream** is any structure that can produce elements of input on demand, e.g.

```
class Stream s m a where
  getNextElement :: s -> m (Maybe (a, s))
  -- where Nothing signifies end-of-stream / end-of-file
```

or similar; of course, lists and thus **Strings**, which are lists of **Chars**, also can be trivially represented this way),

- track position in the input **Stream**,
- allow for programmer-provided types in errors,
- provide **MonadTransformer** versions,
- encode their internals with Scott-encoding (section 5.1.2) for efficiency.

Discussing most of those features and their combinations is beyond the scope of this work. Internal structures used to implement parsing combinators presented throughout this document will be very similar to those used in Parsec and Megaparsec. They, too, parse PEG languages and are built by combining the **State Monad** over input **Stream** with a structure for handling failed parsing attempts. Meanwhile, Attoparsec is built using a somewhat different internal structure (though, it can be argued that it, too, is a variant of **State**) which we shall not discuss in this work (thus reading its sources is highly recommended for educational reasons). All aforementioned libraries also Scott-encode their internals for efficiency, we shall skip that step for clarity reasons, except for section 8.4 where we shall demonstrate Scott-encoding applied to, essentially, the same algebraic structure. From the rest of the possible dimensions listed above in this document we shall only discuss “backtrack vs. not” and “programmer-provided error types”. Detailed implementations of other features can be studied by following respective references.

### 5.2.1 Simple Monadic Parser Combinator

The simplest **Monadic** parser combinator is just a composition of **StateT** (section 4.4.5) and **ExceptT** (section 4.4.3) **MonadTransformers** with inner **Identity** (section 4.1.2)

```
type SParser s e = StateT s (ExceptT e Identity)
```

which can be  $\beta$ -reduced into

```
newtype SParser s e a = SParser
  { runSParser :: s -> Either e (a, s) }
```

with the following **Monad** instance

```
instance Pointed (SParser s e) where
  pure a = SParser $ \s -> Right (a, s)

instance Monad (SParser s e) where
  p >>= f = SParser $ \s ->
    case runSParser p s of
      Left x -> Left x
      Right (a, s') -> runSParser (f a) s'
```

**Theorem 1.** *SParser satisfies Monad laws.*

*Proof.* It is a composition of **StateT** and **ExceptT** **MonadTransformers**.

□



In fact, the above definition is (almost) exactly the definition used in Ponder [73] parser combinator library (it exports the general `m` instead of substituting it with `Identity`). We just need to define an `Alternative` instance for handling failed parsing attempts

```
instance Monoid e => Alternative (SParser s e) where
  empty = SParser $ \s -> Left mempty
  f <|> g = SParser $ \s -> case runSParser f s of
    Right x -> Right x
    Left e -> case runSParser g s of
      Right x -> Right x
      Left e' -> Left (e `mappend` e')
```

and it already gives us enough headroom to define some primitive parsers and a couple of examples

```
type Parser = SParser String [String]

eof :: Parser ()
eof = SParser $ \s -> case s of
  [] -> Right ((), s)
  _ -> Left ["expected eof"]

char :: Char -> Parser ()
char x = SParser $ \s -> case s of
  [] -> Left ["unexpected eof"]
  (c:cs) -> if (c == x)
    then Right ((), cs)
    else Left ["expected `" ++ [x] ++ "' got `" ++ [c] ++ "'"]

string :: String -> Parser ()
string [] = pure ()
string (c:cs) = char c >> string cs

testSParser = runSParser (string "foo") "foo bar"
              == Right((), " bar")
              && runSParser (string "abb" <|> string "abc") "aba"
              == Left ["expected `b' got `a'", "expected `c' got `a'"]
```

### 5.2.2 ... with Full Access to the State

While the definitions above are very simple and get the parsing job done, `SParser` provides no way to access the state of the parser on error, which can make it somewhat inconvenient. However, a simple modification of the type that moves `Either` into the tuple

```
newtype EParser s e a = EParser
  { runEParser :: s -> (Either e a, s) }
```

which, of course, is isomorphic to

```
newtype EParser s e a = EParser
  { runEParser :: s -> Either (e, s) (a, s) }
```

solves this problem of access to state while keeping the definition of `Monad` identical and definitions of the primitive combinators essentially identical to those described in the previous section.

**Theorem 2.** *EParser satisfies Monad laws.*

*Proof.* By case analysis. Also see the proof of theorem 3. □

`MonadTransformer` version of `EParser` can be trivially obtained by adding `Monad` index `m` after the arrow (i.e. by exposing the internal `Monad` of the original `MonadTrans` stack) and correspondingly tweaking all primitive combinators and type signatures.

### 5.2.3 Rollback vs. Not, Programmer-provided Error Types

Note that since `Left` results of the `runSParser` produce no state, the above definition of `<|>` operator has no choice but to supply the same state to both alternatives. In other words, `<|>` of `SParser`, from the point of view of its second argument, always rolls-back the state on failure. Meanwhile, there are six variants of that term for `EParser`

```
f `altEPVariant` g = EParser $ \s -> case runEParser f s of
  Right x -> Right x
  Left (e, s') -> case runEParser g X of
    Right x -> Right x
    Left (f, s'') -> Left (f, Y)
```

with `X` being one of `s`, `s'` and `Y` being one of `s`, `s'`, `s''`. Let us particularly note the following two of those six

```
f `altEPR` g = EParser $ \s -> case runEParser f s of
  Right x -> Right x
  Left (e, _) -> case runEParser g s of
    Right x -> Right x
    Left (e', _) -> Left (e `mappend` e', s)
```

```
f `altEPC` g = EParser $ \s -> case runEParser f s of
  Right x -> Right x
  Left (e, s') -> case runEParser g s' of
    Right x -> Right x
    Left (e', s'') -> Left (e `mappend` e', s'')
```

The `altEPR` is a direct analogue of the `<|>` operator of `SParser`, it runs both alternatives over the same state and returns that same state as the result on failure. Conversely, `altEPC` tries to run the next alternative using the state produced by the previous one, and returns the rightmost state as the result on failure.

**Theorem 3.** *Both `altEPR` and `altEPC` are associative, but the other four variants of `altEPVariant` are not.*

*Proof.* By case analysis.

Note that to convince yourself of the fact that `altEPR` and `altEPC` are associative it is enough to observe that in `a <|> b <|> c` for these two operators

- `Right` is a zero,
- in the absence of zeros, the resulting value of `e` is always the same `mconcat` of all the intermediate values of `e`,
- the value of `s`
  - always stays the same in `altEPR`,
  - always propagates one `runEParser`-step to the right in `altEPC`.

Which means that parentheses can't influence anything for either function.

The same observation that the state either always stays the same or propagates one step at a time can be used to prove associativity of other similar operators for structures derived from `State Monad`.  $\square$

Thus, `Alternative` instance for `EParser` can use either of the two operators and still satisfy the laws of `Alternative`.

```
instance Monoid e => Alternative (EParser s e) where
  empty = EParser $ \s -> Left (mempty, s)
  (<|>) = altEPR
  -- or, alternatively
  -- (<|>) = altEPC
```

**Theorem 4.** *The `Alternative` instance for `SParser` from section 5.2.1 and both of the above instances for `EParser` satisfy `Alternative` laws.*

*Proof.* By the argument of theorem 3.  $\square$

From the popular Haskell parser combinator libraries mentioned above `Attoparsec` rolls-back while `Parsec` and `Megaparsec` do not, instead they implement backtracking with a separate combinator for which we could give the following type signature

```
try :: EParser s e a -> EParser s e a
```

Thus, given a parser for the rolling-back (`<|>`) one can produce an equivalent parser for non-rolling-back (`<|>`) by wrapping all the first arguments of all calls to (`<|>`) with `trys`. This may sound like a pointless complication but it has some performance advantages since some of that wrapping can be skipped when `a <|> b` start by parsing the same prefix, which allows one to write parsers that can parse arbitrary long inputs in constant memory [67].<sup>3</sup>

Semantics-wise our `EParser` combines features of `Attoparsec` (backtracking) and `Megaparsec` (custom error types). Of course, it fits on a single page only because it has a minuscule number of features in comparison to either of the two. To make it practical we would need, at the very least, to implement tracking of the position in the input `Stream` and a bunch of primitive parsers, which we leave as an exercise to the interested reader.

Interestingly, this exact explicit implementation of handling of errors by accumulation via `Alternative` over a `Monoid` seems to be novel (although, pretty trivial). `Megaparsec`, however, does something very similar by accumulating errors in `Sets` instead of `Monoids`. `Parsec` and `Attoparsec` use fixed error types that form `Monoids` instead. `Ponder` gets pretty much the same `Alternative` instance as the one used by `SParser` above for free by being constructed from standardized `MonadTransformer` parts.

## 5.3 Indexed Monads

`Control.Monad.Indexed` module of `indexed` [60] defines an indexed variant of the `Pointed`, `Functor`, `Applicative`, `Monad` type class hierarchy. In this work we shall use the following equivalent set of definitions.

```
class IxPointed m where
  ipure :: a -> m i i a
```

---

<sup>3</sup> Though, the author feels that introducing operators that explicitly drops pieces of the old state is a better approach since it is hard to reason about parsers using non-rolling-back (`<|>`) without knowing their internals (“How much can it consume before failing, again?”). But that discussion is out of scope of this work.

```

class IxFunctor f where
  ifmap :: (a -> b) -> f i j a -> f i j b

infixl 4 <*>
class (IxPointed m, IxFunctor m) => IxApplicative m where
  (<*>) :: m i j (a -> b) -> m j k a -> m i k b

infixl 1 >>=+
class IxApplicative m => IxMonad m where
  (>>=+) :: m i j a -> (a -> m j k b) -> m i k b

-- (>>) equivalent for (>>=+)
(>>+) :: IxMonad m => m i j a -> m j k b -> m i k b
a >>+ b = a >>=+ const b

-- IxApplicative from IxMonad
iap :: IxMonad m => m i j (a -> b) -> m j k a -> m i k b
iap a b = a >>=+ \f -> b >>=+ \a -> ipure (f a)

-- IxFunctor from IxApplicative
iliftM :: IxApplicative m => (a -> b) -> m i j a -> m i j b
iliftM f a = (ipure f) <*> a

```

### 5.3.1 Indexed State Monad

`IxMonad` is not a particularly popular algebraic structure, but it well-known enough to have its own Hackage library. Its usefulness in the context of this work comes from the fact that if one is to generalize the `State` type

```

newtype State s a = State { runState :: s -> (a, s) }

```

a little producing the following type we shall call `IxState`

```

newtype IxState i j a = IxState { runIxState :: i -> (a, j) }

```

then the terms of operators of `Monad State` can also be used as terms for `IxMonad IxState` without modifications

```

instance IxPointed IxState where
  ipure a = IxState $ \i -> (a, i)

```

```

instance IxMonad IxState where
  m >>=+ f = IxState $ \i ->
    let (a, j) = runIxState m i
    in runIxState (f a) j

```

```

instance IxFunctor IxState where
  ifmap = iliftM

```

```

instance IxApplicative IxState where
  (<*>) = iap

```

## 5.4 Other Variants of MonadCatch

Finally, returning back to the topic `throw` and `catch`, worth mentioning are two lesser-known variants of structures similar to structures of section 4.8. The first one is defined in `Control.Monad.Exception.Catch` module of `control-monad-exception` [45] package as

```
class (Monad m, Monad n) => MonadCatch e m n | e m -> n, e n -> m where
  catch :: m a -> (e -> n a) -> n a
```

and the second one in `Control.Monad.Catch.Class` module of `catch-fd` [100] package

```
class Monad m => MonadThrow e m | m -> e where
  throw :: e -> m a
```

```
class (MonadThrow e m, Monad n) => MonadCatch e m n | n e -> m where
  catch :: m a -> (e -> n a) -> n a
```

Note that `control-monad-exception` does not define a type class with a `throw` operator, that library provides a universal computation type `EM` (similar to `EIO` of section 8.4) with such an operator instead. Also note that the common point of those two definitions is that both `catch` operators change the type of computations from `m` to `n`.

## Part II

# Exceptionally Monadic Error Handling

# Chapter 6

## Motivation

**Definition 1.** *Generally, when program encounters an “error” all it can do is to switch to an “exceptional” execution path [9]. The latter can then either encounter an “error” itself or*

1. *gracefully “terminate” some part of the previous computation (including the whole program as a degenerate case) and continue (when there is something left to continue),*
2. *“fix” the “problem” and resume the computation as if nothing has happened.*

*Error handling*<sup>1</sup> is an algebraic subfield of the programming languages theory that studies this sort of seemingly simple control structures.

Different substitutions for “error”, “exceptional” and “terminate” into definition 1 variant 1 and substitutions for “error”, “exceptional”, “fix” and “problem” into definition 1 variant 2 produce different error handling mechanisms. Some examples:

- Identity substitution for variant 1 gives programming with error codes, programming with algebraic data types [4, 14] that encode errors, programming with algebraic data types with errors [32, 33] (not the same thing), exceptions in conventional programming languages [9, 34, 37, 38, 62] (with so called “termination semantics” [104, 16.6 Exception Handling: Resumption vs. Termination]), error handling with monads [44, 52, 79, 80, 105, 110], monad transformers [8, 31, 68], Scheme’s and ML’s `call/cc` [101], and delimited `callCC` [3, 31, 54].
- Substituting “unparsable string”, “alternative”, “backtrack” for variant 1 gives monadic parser combinators [67].
- Identity substitution for variant 2 gives error handling in languages with so called “resumption semantics” [104, 16.6 Exception Handling: Resumption vs. Termination] like, for instance, Common LISP [91] (*condition handling*) and Smalltalk [34].
- Substituting “effect”, “effect handler”, *handle*“, ”/it/” for variant 1 or 2 (depending on the details of the calculus) produces effect systems [8, 12, 50, 55, 56, 92] and effect systems based on modal logic with names [81, 82].

---

<sup>1</sup> Not a consensus term. Some people would disagree with this choice of a name as they would not consider some of our examples below to be about “errors”. However, for the purposes of this work we opted into generalizing the term “error” of “error handling” instead of inventing new terminology or appropriating terminology like “exceptions”, “interrupts”, “conditions” or “effects” that has other very specific uses. To see the problem with the conventional terminology consider how would you define “program encountered an error” formally and generally for **any** abstract interpreter (you can not). Now consider the case where an interpreter is a tower of interpreters interpreting one another. Clearly, what is an “error” for one interpreter can be considered normal execution for the one below. A simple example of such a structure is the **Maybe Monad** discussed in section 4.2.5 in which expressions using `do`-syntax never consider **Nothings** while handling of said **Nothings** by the **Monadic** (`>>=`) operator is a completely ordinary **case** for the underlying Haskell interpreter. Hence, in this work we consider anything that matches definition 1 to be about “error” *handling*. If the reader still feels like disagreeing with our argument we advise mentally substituting every our use of “error” with something like “an abnormal program state causing execution of an abnormal code path” (where definitions of both “abnormal”s are interpreter-specific).

- “*System call*”, “*system call handler*”, “*handle*”, “*it*” for variant 2 produces conventional *system calls* [46].<sup>23</sup>
- Substituting “*signal*”, “*signal handler*”, “*handle*”, “*it*”, “*it*” for variant 2 gives hardware interrupts and POSIX signals [46].<sup>4</sup>

The first complication of the above scheme is the question of whenever for a given error handling mechanism the “*error*” raising operator

1. passes control to a statically selected (lexically closest or explicitly specified) enclosing error handling construct (e.g. `throw` and `catch` in Emacs LISP [23], POSIX system calls and signals) or
2. the language does dynamic dispatch to select an appropriate error handler (like exceptions in most conventional languages like C++, Java, Python, etc do).

Another complication is ordering:

1. Most conventional programming languages derive their error handling from SmallTalk [34] and Common LISP [91] and the order in which the program handles “*errors*” corresponds to the order in which execution encounters them.
2. Meanwhile, some CPU ISAs<sup>5</sup> expose the internal non-determinism and allow different independent data-flows to produce hardware exceptions in non-deterministic manner (e.g. arithmetic instructions on DEC Alpha). So do Haskell [90] (see section 4.5) and, to some extent, C++ [15] programming languages.

Finally, another dimension of the problem is whenever the objects signifying “*errors*” (e.g. arguments of `throw`) are

1. first-class values (error codes, algebraic data types) as in most conventional languages,
2. labels or tags as in modal logic with names and, to some degree, with `call/cc` and `callCC`.

In short, despite its seemingly simple operational semantics, error handling is an algebraically rich field of programming languages theory.

Meanwhile, from the perspective of types there are several schools of thought about effects.

- The first one, started by Gifford and Lucassen [29, 69, 70] represents effects as type annotations. This works well in programming languages with eager evaluation, but becomes complicated in lazy languages (application in a lazy language delays effects until thunk’s evaluation, hence type system has to either put nontrivial restrictions on the use of effects in expressions or annotate both arrows and values with effects, the latter, among other things, breaks type preservation of  $\eta$ -conversion since  $\lambda x.f x$  moves effect annotation from the arrow to the result type).
- The second one, started by Moggi and Wadler [79, 110] confines effects to monadic computations. The latter can then be annotated with effect annotations themselves [111]. Monads work well for small programs with a small number of effects, but, it is commonly argued, they don’t play as nice in larger programs because they lack in modularity [12] (hence, the need for monad transformers, which are then critiqued as hard to tame [56]) and produce languages with non-uniform syntax (pure functions look very different from monadic ones and functions that are useful in both contexts have to be duplicated, think e.g. `map` and `mapM`).

---

<sup>2</sup> Except in most UNIX-like operating systems system calls cannot call other system calls directly and have to use an equivalent kernel API instead.

<sup>3</sup> Indeed, algebraic effects from the point of view of an OS-developer are just properly typed system calls with nesting and modular handling.

<sup>4</sup> Indeed, POSIX signals and hardware interrupts are “system calls in reverse” (with some complications outside of the scope of this work): kernel and/or hardware raises and applications handle them.

<sup>5</sup> Instruction Set Architecture (ISA) is a specification that describes a set of Operation Codes (OPcodes, which are a binary representation of an assembly language) with their operational semantics. “i386”, “i686”, “amd64” (“x86\_64”), “aarch64”, “riscv64”, etc are ISAs.



- The third one, started by Nanevski [81] represents effects using modal logic with names. Practical consequences of this way of doing things are unknown, as this construction didn't get much adoption yet.

In short, from type-theoretic point of view the progression of topics in the cited literature can be seen as pursuing calculi that are, at the same time, computationally efficient, algebraically simple (like monads), but modular (like effect systems).

Note, however, that all of those schools of thought consider exceptions to be effects, they only disagree about the way to represent the latter. Meanwhile, from a perspective of a programming language implementer, there are several problems with that world view:

- mechanisms that support resumption semantics are commonly disregarded as useless and computationally expensive error handling mechanisms (most notably [104, 16.6 Exception Handling: Resumption vs. Termination, pp. 390–393]),
- in particular, all popular programming languages implement builtin exceptions even though they have more general error handling mechanisms like *condition handling* in Common LISP and `call/cc` in Scheme and ML because those are just too computationally expensive for emulation of conventional exceptions [54],
- and even in languages with nothing but exceptions and termination semantics, high-performance libraries that do a lot of error handling frequently prefer not to use exceptions for performance reasons and to remove any non-local control-flow.

In short, from practical point of view *most* of those type-theoretic constructs are an overkill for *most* programs. Meanwhile, we are not aware of any non-ad-hoc language-agnostic algebraic structure that captures all of the exception handling (both `throwing`, and `catching`) without introducing any other superfluous structure on top. In this work we shall demonstrate a fairly straightforward but surprisingly useful solution to this problem.

Sections 7.1 and 7.2 derive a solution for the problem in question by purely pragmatic reasoning and then prove that the resulting structure is the only possible solution matching conventional operational semantics for `throw` and `catch` operators. Section 7.3 gives a proper formal definition for the resulting structure. Section 7.4 discusses the most trivial instance of the structure: the `Either` type. Section 7.5 discusses the issue of encoding of the resulting formal structure in Haskell and similar languages. Chapter 8 discusses many more instances of the structure. Chapter 9 shows that the resulting structure can be generalized even further by decomposing it into a Cartesian product with interaction laws and replacing parts with more general structures. Chapter 10 discusses the consequences of these observations and the general picture.

# Chapter 7

## Derivation

### 7.1 The Nature of an Error

Lets forget for a minute about every concrete algebraic error-handling structure mentioned before and try to invent our own algebra of computations by reasoning like a purely pragmatic programmer who likes to make everything typed as precisely as possible.

We start, of course, by pragmatically naming our type of computations to be `C`. Then, we reason, it should be indexed by both the type of the result, which we shall pragmatically call `a`, and the type of exceptions `e`. We are not sure about the body of that definition, so we just leave it undefined

```
data C e a
```

Now, we know that `Monads` usually work pretty well for the computation part (since we can as well just lift everything into `IO` which is a `Monad`), so we write

```
pure :: a -> C e a
```

```
(>>=) :: C e a -> (a -> C e b) -> C e b
```

and expect these operators to satisfy `Monad` laws (section 4.2.1).

Meanwhile, pragmatically, an “exceptional” execution path requires two conventional operators:

- a method of raising an exception; the type of this operator seems to be pretty straightforward

```
throw :: e -> C e a
```

as it simply injects the error into `C`,

- and a method to catch exceptions; the overly-general type for this operator is, again, pretty straightforward

```
catch :: C e a -> (e -> C f b) -> C g c
```

The only obvious requirement here is that the type the “handler” function (the second argument of `catch`) can handle should coincide with the type of errors the “computation” (the first argument) can `throw`.

Finally, we pragmatically expect the above to obey the conventional operational semantics of error handling operators, giving us the following definition.

**Definition 2. Pragmatic error handling structure.** Structure `m :: * => * => *` with `pure`, `(>>=)`, `throw`, and `catch` operators satisfying

1. `pure` and `(>>=)` obey `Monad` laws (section 4.2.1),

2. `throw e >>= f == throw e` (“throwing of an error stops the computation”),
3. `throw e `catch` f == f e` (“throwing of an error invokes the most recent error handler”),<sup>1</sup>
4. `pure a `catch` f == pure a` (“pure is not an error”).

## 7.2 The Type of Error Handling Operator

The first question to the structure of `C` is, of course, what is the precise type of `catch` operator.

```
catch :: C e a -> (e -> C f b) -> C g c
```

In other words, we would like to know which of the variables `f`, `g`, `b`, and `c` in this signature should have their own universal quantifier and which should be substituted with others. The answer comes by considering several cases.

- Firstly, let us consider the following expression.

```
pure a `catch` f
```

The expected semantics of `catch` requires (by item 4 of definition 2)

```
pure a `catch` f == pure a
```

Note that the most general type for `pure a` expression is `forall e . C e a` for `a : a2`. Moreover, we can assign the same type to any expression that does not `throw` since

- both `a` and `e` in the type signify the potential to `pure` and `throw` values of the corresponding types,
- and an expression that does not `throw` any errors can be said to not-`throw` an error of any particular type, similarly to how bottom elimination rule works. Or, equivalently, any such computation can be said to `throw` values of an empty type and an empty type can always be replaced with any other type by bottom elimination.<sup>3</sup>

- Now let us consider the following expression, assuming `e` and `f` are of different types (i.e. both the computation and the handler throw different exceptions).

```
throw e `catch` (\_ -> throw f)
```

The expected semantics of `catch` requires (by item 3 of definition 2)

```
throw e `catch` (\_ -> throw f) == throw f
```

These two cases show that `g` should be substituted with `f` and `e` should be kept separate from `f` because

- if computation `throws` then the type `f` in the handler “wins”,
- but if it does not `throw` then `e` is an empty type and it can be substituted for any other type, including `f` (similarly to the type of `pure` above)<sup>4</sup>

<sup>1</sup> Similarly to GHC’s imprecise exceptions of section 4.5 dynamic dispatch can be implemented on top of such a structure. We shall do this in section 8.1.2.

<sup>2</sup> The reader might have noticed already that we abuse notation somewhat by assuming type variables and term variables use distinct namespaces. This expression happens to be the first and the only one that uses both at the same time, hence it looks like an exiting “type-in-type” kind of thing, but it is not, it is ordinarily boring.

<sup>3</sup> Implicitly or with `f `catch` bot-elim` which is extensionally equal to `f`.

<sup>4</sup> The only nontrivial observation in this section.

- these two cases are mutually exclusive.

That is, the type for `catch` is at most as general as

```
catch :: forall e f . C e a -> (e -> C f b) -> C f c
```

- Continuing, item 4 of definition 2 shows that `c` has to coincide with `a`.
- Similarly, item 3 requires

```
throw e `catch` (\_ -> pure a) == pure a
```

which shows that `c` has to coincide with `b`.

All these observations combine into the following.<sup>5</sup>

**Theorem 5.** *For any type  $C :: * \Rightarrow * \Rightarrow *$  obeying definition 2 the most general type for the `catch` operator is*

```
catch :: forall a e f . C e a -> (e -> C f a) -> C f a
```

*Proof.* By the above reasoning. That is, by simple unification of types of `pure`, `throw`, (`>>=`) operators of definition 2 and the following equations that are consequences of equations of definition 2

```
pure a `catch` f == pure a
throw e `catch` (\_ -> pure a) == pure a
throw e `catch` (\_ -> throw f) == throw f
```

□

### 7.3 Formal Account: Conjoinedly Monadic Algebra

After theorem 5 it becomes hard to ignore the fact that `throw` has the type of `pure` and `catch` has the type of (`>>=`) in the “wrong” index for `C`. Moreover, item 3 of definition 2 looks exactly like a left identity law for `Monad` (section 4.2.1). While it is not as immediately clear that `catch` should be associative, it seems only natural to ask whenever the following conjoinedly `Monadic` restriction of definition 2 has any instances.

**Definition 3.** *Conjoinedly monadic error algebra.* A type  $m :: * \Rightarrow * \Rightarrow *$  for which

- $m$  is a *Monad* in its second index (that is,  $m\ e$  is a *Monad* for all  $e$ ),
- $m$  is a *Monad* in its first index (that is,  $\backslash e . m\ e\ a$  is a *Monad* for all  $a$ ),

and assuming

- the names of *Monad* operators in the second index of  $m$  are `pure` and (`>>=`),
- the names of *Monad* operators in the first index are `throw` and `catch`,

the following equations hold

---

<sup>5</sup> Spoilers! The reader is only supposed to notice the following after reading section 7.4.1.

Note that we could have written an equivalent up to names of operators sections 7.1 and 7.2 that explained why the type of (`>>=`) is the correct type for sequencing computations in `C` given that error handling should be done *Monadically*. In particular, the fact that the dual of definition 2 lists valid operational equations is a rather curious observation by itself. Which is another reason why we disagree with the conventional wisdom in footnote 1.

1. `pure x `catch` f == pure x,`
2. `throw e >>= f == throw e.`

If we replace `Monad` in definition 3 with `MonadFish` (section 4.2.2), as usual, the latter two equations become a bit clearer.

**Definition 4.** *Fishy conjoinedly monadic error algebra.* A type `m :: * => * => *` for which

- `m` is a `MonadFish` in its second index,
- `m` is a `MonadFish` in its first index,

and assuming

- the names of `MonadFish` operators in the second index are `pure` and `(>=>)`,
- the names of `MonadFish` operators in the first index are `throw` and `handle`,

the following equations hold

1. `pure `handle` f == pure,`
2. `throw >=> f == throw.`

On other words, definitions 3 and 4 define a structure that is a `Monad (MonadFish)` twice and for which `pure` is a left zero for `catch` (`handle`) and `throw` is a left zero for `(>=>)` (`(>=>)`).

## 7.4 Instance: Either

Pragmatic programmer finally loses last bits of concentration realizing that `Either` type seems to match requirements of definition 3 and goes into sources to check whenever Haskell's standard library already has such a `catch`. Unfortunately, `Data.Either` module does not define such an operator. However, `catchE` and `throwE` of `ExceptT` (section 4.4.3) match. Of course, if we substitute `Identity` for `m`, `ExceptT` turns into `Either` and those operators can be simplified to

```
throwE' :: e -> Either e a
throwE' = Left

catchE' :: Either e a
         -> (e -> Either f a)
         -> Either f a
catchE' (Left e) h = h e
catchE' (Right a) _ = Right a
```

**Lemma 3.** *For a given `Monad m` and a fixed argument `a`, `ExceptT` with `throwE` as `pure` and `catchE` as `(>=>)` is a `Monad` in argument `e`.*

*Proof.* Any of the following

- **By brute force:** by case analysis, using the fact that `m` satisfies `Monad` laws.
- **Another way:** trivial consequence of section 7.4.1.

□

**Lemma 4.** *For `ExceptT` with the above operators the following equations hold*

1. `pure x `catchE` f == pure x`,
2. `throwE e >>= f == throwE e`.

*Proof.* By trivial case analysis. □

**Theorem 6.** *ExceptT* and, by consequence, *Either* satisfy definition 3.

*Proof.* Consequence of lemma 3 and lemma 4.

Thus, using, the encoding of section 7.5, we can write:

```
instance ConjoinedMonads Either where
  cpure = pure
  cbind = (>>=)

  cthrow = throwE'
  ccatch = catchE'
```

□

### 7.4.1 Logical Perspective

Note, that from a logical perspective most of the above is simply trivial. *Either a b* is just  $a \vee b$  and so if  $\lambda b.a \vee b$  is a *Monad* then  $\lambda a.a \vee b$  must be a *Monad* too since  $\vee$  operator is symmetric.

In fact, in [35] Gabriel Gonzalez, the author of the *errors* [36] package, also explicitly mentions the fact that the *Monadic* operators for the other index of *Either* seem to match the semantics for the corresponding *throw* and *catch* operators (though, without proofs or claims of general applicability). He then mentions that the fact itself was first pointed out to him by Elliott Hird who named the other *Monad* the “success *Monad*”.

From this point of view, the contribution of sections 7.1 to 7.3 is that they generalize this observation into definition 3, prove theorem 5, and notice the general duality discussed in footnote 5.

Then, the main point of this whole part of the work is that **there are other instances** of this generalization and, more importantly, that **this generalization is itself interesting** — the facts that we shall demonstrate in the sections that follow.

## 7.5 Formal Account: Haskell Encoding

Despite the noted triviality, these facts do not seem to be appreciated by the wider Haskell community. In particular:

- *ExceptT* does not get much use in Hackage packages in general,
- the equivalent of *catchE* for *ErrorT* has an overly-restricted type

```
catchError :: (Monad m)
            => ErrorT e m a
            -> (e -> ErrorT e m a)
            -> ErrorT e m a
m `catchError` h = ErrorT $ do
  a <- runErrorT m
  case a of
    Left l -> runErrorT (h l)
    Right r -> pure (Right r)
```

- no *Monadic* parser combinator library from Hackage (most obvious beneficiaries of the observation) defines the would-be-*Monad* instance of *throwE* and *catchE*.

To our best knowledge, the only Hackage package that is explicitly aware of the fact that `Either` is a `Monad` twice is `errors` [36] and the only packages that seem to be aware that `throw` and `catch` in general need more general types than those given by `MonadCatch` of section 4.8 are those discussed in section 5.4 (but they miss the fact that their `catch` operators want to be `Monadic binds`). To our best knowledge, no Hackage package utilizes both facts.

As to the question why had not anybody notice and start exploiting these facts yet we hypothesize that the answer is because Haskell cannot express these properties conveniently (not to mention less expressive mainstream languages which cannot express them at all).

The simplest possible encoding of definition 3 in Haskell is just

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

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

but it does not play too well with the rest of the Haskell ecosystem. In the ideal world, definition 3 would get encoded with the following pseudo-Haskell definition

**Definition 5.** *Proper pseudo-Haskell definition.*

```
class (forall a . Monad (\e -> m e a) -- `Monad` in `e`
      , forall e . Monad (\a -> m e a)) -- `Monad` in `a`
  => ConjoinedMonads m where
  -- and that's it
```

however, Haskell allows neither rank 2 types in type classes, nor lambdas in types, which brings us to the following “theorem”.

**“Theorem” 7.** *Haskell cannot properly (equivalently to definition 5) define `ConjoinedMonads`.*

*Proof.* Proper definition of `ConjoinedMonads` requires rank 2 types in type class declaration, which is not possible in modern Haskell. There is no way to emulate rank 2 definition using only rank 1 constructions. □

We call it a “theorem” because we do not really know if its proof really works out for Haskell as Haskell has an awful lot of language extensions (including future ones) and there might be some nontrivial combination of those that gives the desired effect. In particular, GHC version 8.6 released just before this part of the work was finished introduced `QuantifiedConstraints` extension [10] allowing us to write

```
data Swap r a e = Swap { unSwap :: r e a }

instance (forall e . Monad (r e)
         , forall a . Monad (Swap r a))
  => ConjoinedMonads r where
  -- ...
```

(note that this is an `instance`, not a `class`) which, arguably, can be considered good enough, though, again, not very convenient in practice.

The purposes of this work, however, is not to demonstrate that there is a convenient form of definition 3 in Haskell but to show what could be achieved if there were such a convenient definition. Which means that we can and, hence, shall completely ignore the question of the most elegant Haskell representation for definition 3 and just use the following variation on very first definition of `ConjoinedMonads` from above for simplicity.

```
-- We have to add the `c` prefix here so that it won't conflict with other definitions.
class ConjoinedMonads m where
  cpure  :: a -> m e a
  cbind  :: m e a -> (a -> m e b) -> m e b

  cthrow :: e -> m e a
  ccatch :: m e a -> (e -> m f a) -> m f a
```

As to the naming, it is, indeed, tempting to call this structure **BiMonad**, but that name is already taken by another structure from category theory. Then, since the structure consists of two **Monads** that are “dual” to each other via interaction laws it is tempting to call it **DualMonad** as a double-pun, but that “duality” is different from the usual duality of category theory. Which is why we opted into using the name “**ConjoinedMonads**” (in the sense of “conjoined twins”, conjoined with left-zeroes).



# Chapter 8

## Non-trivial Instances

### 8.1 Constant Functors

In this section we discuss the relationship between `ConjoinedMonads` (and definition 3) and `MonadThrow`, `MonadCatch`, and `MonadError` from section 4.8.

#### 8.1.1 `MonadError`

`MonadError` (section 4.8.1) relationship to `ConjoinedMonads` turns out to be pretty simple. Remember that `MonadError` is defined using functional dependencies

```
class (Monad m) => MonadError e m
    | m -> e where
```

This means that Haskell type system guarantees that for each `m` there exist unique `e` if `MonadError e m` is inhabited. This, in turn, means that substituting a constant `Functor r = \x a -> m a` over `Monad m` into the definition of `ConjoinedMonads` produces

```
class ConjoinedMonads (\x a -> m a) where
  cpure :: a -> m a
  cbind :: m a -> (a -> m b) -> m b

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

The first two operators are just the definition of `Monad m`, the latter two match `MonadError`'s `throwError` and `catchError` exactly.

**Theorem 8.** *`MonadError` is a `ConjoinedMonads` that is constant in its first index.*

*Proof.* By the above argument. □

#### 8.1.2 `MonadThrow` and `MonadCatch`

For `MonadThrow` and `MonadCatch` (section 4.8.2) it is not the case that `e` is unique, since `Exception e` is a whole class of types. Moreover, operator `catchM` of `MonadCatch`, unlike `catchError` of `MonadError`, does dynamic dispatch by casting `Exceptions` to the type of its handler's argument and propagating errors when the `cast` fails. Note that, strictly speaking, purely from type perspective `MonadCatch` is not *required* but *allowed* to `cast`, but all the instances do actually `cast`. The latter fact means that we can distill that common computational pattern by redefining those structures using the technique used by imprecise exceptions of section 4.5 as follows

```

class Monad m => MonadThrowS m where
  throwS :: SomeException -> m a

class MonadThrow m => MonadCatchS m where
  catchS :: m a
           -> (SomeException -> m a) -> m a

throwM' :: (MonadThrowS m, Exception e)
         => e -> m a
throwM' = throwS . toException

handleOrThrowAgain h e = case fromException e of
  Just f -> h f
  Nothing -> throwM e

catchM' :: (MonadCatchS m, Exception e)
         => m a -> (e -> m a) -> m a
catchM' ma = catchS ma . handleOrThrowAgain

```

Note that `MonadCatchS` is, again, a constant `ConjoinedMonads` with error index fixed to `SomeException`. Also note that `throwM'` above is the only way to get an equivalent for `throwM` because `toException` is the only way to cast an arbitrary type to `SomeException`. On the other hand, `catchM` from `MonadCatch`, unlike `catchM'` above, allows for instances that can cheat. For example, `catchM` can give a constant `SomeException` to the handler every time instead of casting anything. The author feels that this implies that `MonadCatch` is not a proper formal structure for error handling.

**Definition 6. Proper `MonadCatch` instance.** We shall call an instance of `MonadCatch` proper when its `catchM` can be decomposed into `catchS` and `handleOrThrowAgain`.

**Theorem 9.** Every proper instance of `MonadCatch` is a composition of `ConjoinedMonads` that is constant in its error index with `toException` in `throwD` and `handleOrThrowAgain` in `catchD`. In particular, `MonadThrow` is a composition of `Pointed` in the error index with `toException`.

*Proof.* By the above reasoning. □

## 8.2 Parser Combinators

In this section we discuss the application of `ConjoinedMonads` and definition 3 to `Monadic` parser combinators discussed in section 5.2.

### 8.2.1 The Boring Part

To start off, let us continue using the definitions of `SParser` and `EParser` types from section 5.2. Similarly to `Alternative` (section 4.1.3) instances of section 5.2.3, the `Monad` instances in index `e` for those types can be discovered by going through all free functions of appropriate types satisfying `Monadic` laws. Thus, similarly, there is a single possible implementation for both `pures` in index `e`,

```

throwSP :: e -> SParser s e a
throwSP e = SParser $ \_ -> Left e

throwEP :: e -> EParser s e a
throwEP e = EParser $ \s -> Left (e, s)

```

one possible implementation for ( $\gg=$ ) of `SParser`, and two possible implementations for ( $\gg=$ ) of `EParser`,

```
catchSP :: SParser s e a -> (e -> SParser s f a) -> SParser s f a
catchSP p f = SParser $ \s ->
  case runSParser p s of
    Right x -> Right x
    Left e -> runSParser (f e) s

catchEPR :: EParser s e a -> (e -> EParser s f a) -> EParser s f a
catchEPR p f = EParser $ \s ->
  case runEParser p s of
    Right x -> Right x
    Left (e, _) -> runEParser (f e) s

catchEPC :: EParser s e a -> (e -> EParser s f a) -> EParser s f a
catchEPC p f = EParser $ \s ->
  case runEParser p s of
    Right x -> Right x
    Left (e, s') -> runEParser (f e) s'
```

all of which satisfy `Monad` laws. Similarly to section 5.2.3, `catchEPR` does backtracking on failures and `catchEPC` proceeds with the current state.

**Theorem 10.** *SParser is an instance ConjoinedMonads.*

*EParser is an instance of ConjoinedMonads for both versions of catchEP.*

*Proof.* `Monad` laws for `catchSP` and `catchEP` follow from the corresponding laws for ( $\gg=$ ) of section 5.2.

The rest can be proven by trivial case analysis and/or by using the observation from the proof of theorem 3.

```
instance ConjoinedMonads (SParser s) where
  cpure = pure
  cbind = (>>=)

  cthrow = throwSP
  ccatch = catchSP

instance ConjoinedMonads (EParser s) where
  cpure = pure
  cbind = (>>=)

  cthrow = throwEP
  ccatch = catchEPR
  -- or, alternatively
  -- ccatch = catchEPC

-- (>>) in index `e`
orElse :: ConjoinedMonads m => m e a -> m f a -> m f a
orElse f g = f `ccatch` const g
```

□

A curious consequence of the above theorem and symmetries noted in section 7.4.1 is that ( $\gg=$ ) of `EParser` of section 5.2.2 also has a roll-back version which satisfies `Monad` laws

```

bindEP p f = EParser $ \s ->
  case runEParser p s of
    Left x -> Left x
    Right (a, _) -> runEParser (f a) s

```

Though, of course, a `EParser` that would use `bindEP` in place of the usual `(>>=)` could not be called a “parser” anymore.

Finally, note that `(<|>)` operators of the `Alternative` of section 5.2 for both `SParser` and `EParser` can be expressed in terms of their respective `(>>=)` operators for their `Monads` in index `e`

```

f `altSP` g = f `catchSP` \e ->
  g `catchSP` \e' ->
  throwSP (e `mappend` e')
-- == (<|>)

```

```

f `altEPR` g = f `catchEPR` \e ->
  g `catchEPR` \e' ->
  throwEP (e `mappend` e')
-- == altEPR

```

```

f `altEPC` g = f `catchEPC` \e ->
  g `catchEPC` \e' ->
  throwEP (e `mappend` e')
-- == altEPC

```

```

instance Monoid e => Alternative (SParser s e) where
  empty = SParser $ \_ -> Left mempty
  (<|>) = altSP'

```

```

instance Monoid e => Alternative (EParser s e) where
  empty = EParser $ \s -> Left (mempty, s)
  (<|>) = altEPR'
  -- or, when `catch == catchEPC`
  -- (<|>) = altEPC'

```

## 8.2.2 The Interesting Part

The interesting part comes from the observation that **some** and **many** operators of `Alternative` (section 4.1.3) never use the `Monoidal mappend` the above definitions of `(<|>)` do, since both operators stop on the very first failure. In other words, `orElse`, which is just `(>>)` operator for the `Monad` in index `e` (see above), is enough to implement them. Interestingly, however, using `orElse` instead of `(<|>)` produces a curious effect of supplying these implementations with types that clearly show that `some` inherits errors produced by its argument while `many` ignores them

```

csome :: (ConjoinedMonads m, Applicative (m e))
  => m e a -> m e [a]
csome p = fmap (:) p <*> cmany p

cmany :: (ConjoinedMonads m, Applicative (m e))
  => m e a -> m f [a]
cmany p = csome p `orElse` cpure []

```

(In the above, `Applicative` constraint is superfluous, it is an artifact of our encoding of `ConjoinedMonads` into Haskell discussed in section 7.5. We could elide it if we were to derive `<*>` and `fmap` operators from `ConjoinedMonads`, an approach that we shall use in chapter 17.)

This method of substituting `<|>` with `orElse` extends to other similar combinators like `choice`, `optional`, `sepBy`, `notFollowedBy` of all three aforementioned parser combinator libraries (`Parser`, `Attoparsec`, `Megaparsec`) and similar structures. The overall effect of this substitution is very useful in practice: it produces generic parser combinators that can be used to express parsers that are precise about errors they raise and handle. We can not emphasize this fact enough.

All of the above results of this section trivially generalize to their `MonadTrans` versions as usual.

## 8.3 Conventional throw and catch via callCC

It is well-known fact that Emacs LISP-style `throw` and `catch` can be emulated with Scheme's `call/cc` and some mutable variables [65, 112]. As a Haskell instance, Neil Mitchel used the same technique translated to Haskell's `IORefs` and `callCC` in for Shake build system [76, 77] (however, at the time of writing Shake no longer uses that code). In this section we shall demonstrate that a structure with the same semantics can be implemented in pure Haskell without the use of mutable variables. In all the cases, as usual, C++/Java-style dynamic dispatch can be added on top using the same `casting` technique of sections 4.5 and 8.1.2. Hence without the loss of generality in this section we shall discuss only the most-recent-handler case.

### 8.3.1 Second-rank callCC

Remember the definition of `callCC` from section 5.1.4. The underappreciated fact about that function is that its type is not its most general type for its term. Note that variable `b` in Peirce's law

$$((a \rightarrow b) \rightarrow a) \rightarrow a$$

plays the same role as `r` plays in the definition of `Cont`: it is a generalization of the bottom  $\perp$  constant. This, of course, means that we can generalize Peirce's law to

$$((\forall b. a \rightarrow b) \rightarrow a) \rightarrow a$$

and, by repeating the derivation in section 5.1.4, give the following second-rank type for `callCC`

```
callCCR2 :: ((forall b . a -> Cont r b) -> Cont r a) -> Cont r a
```

while keeping exactly the same implementation.

### 8.3.2 ThrowT MonadTransformer

Note that, in essence, `catch` maintains a stack of handler addresses and `throw` simply jumps to the most recent one. Emulation of exceptions with `call/cc` works similarly [65, 112]. The main never explicitly stated observation in that translation is that the type of the handler in the type of

```
catch :: M -> (e -> M) -> M
```

matches the type of `throw :: e -> M` and the type of escape continuation when `M` is `ContT r m b`. In other words, we can simply assign

```
type Handler r e m = forall b . e -> ContT r m b
```

to be to type of our handler and since `callCC` provides an escape continuation directly to its argument `catch` can simply save it and `throw` can simply take the most recent one and escape into it

```
throwT :: e -> ThrowT r m e a
throwT e = ThrowT $ \currentThrow -> currentThrow e
```

Also note that since the stack `catch` maintains stays immutable between `catches` and each state of the stack is bound to the computation argument of `catch`, in principle, we should be able to use a simple context (pure function, `Reader`) instead of a mutable variable as follows

```
type ThrowT r m e a =
  ReaderT (Handler r e m) -- for saving last handler
    (ContT r m)          -- for callCC
  a
```

which, after inlining all the definitions except pure `Cont` becomes

```
newtype ThrowT r m e a = ThrowT
  { runThrowT :: (forall b . e -> Cont (m r) b)
    -> Cont (m r) a }
```

Finally, since the escape continuation of delimited `callCC` escapes to the same address where the body of `callCC` normally returns, to emulate a single `catch` we need to chain two `callCCs` as follows

```
catchT :: ThrowT r m e a
  -> (e -> ThrowT r m f a)
  -> ThrowT r m f a
catchT m h = ThrowT $ \outerThrow ->
  callCC $ \normalExit -> do
    e <- callCCR2 $ \newThrow -> runThrowT m newThrow >>= normalExit
    -- newThrow escapes here
    runThrowT (h e) outerThrow
    -- normalExit escapes here
```

Note that this expression requires our second-rank `callCCR2` since our `Handler` is universally quantified by the variable `b`. However, if we fix `e` to a constant type then the conventional `callCC` will suffice.

Similarly to other uses of generalized Kolmogorov's translation we, too, can hide `r` parameter behind `forall`

```
newtype ThrowT' m e a = ThrowT'
  { runThrowT' :: forall r
    . (forall b . e -> Cont (m r) b)
    -> Cont (m r) a }
```

```
throwT' :: e -> ThrowT' m e a
catchT' :: ThrowT' m e a
  -> (e -> ThrowT' m f a)
  -> ThrowT' m f a
```

without any changes to the bodies of `throw` and `catch`.

**Theorem 11.** *For Monad `m` and any `r`, `ThrowT r m` and `ThrowT' m` are ConjoinedMonads.*

*Proof.* For each index.

- In index `a`: `ThrowT` is a special case of `ReaderT` and `Cont` and `m` are `Monads`.
- In index `e`: by substitution of the above definitions into the `Monad` laws, since the definitions of `throwT` and `throwT'` are, essentially, identity functions.

```

instance Monad m => Pointed (ThrowT r m e) where
  pure a = ThrowT $ \_ -> pure a

instance Monad m => Monad (ThrowT r m e) where
  -- a lift of Cont's (>>=)
  ma >>= f = ThrowT $ \err
    -> Cont $ \c -> runCont (runThrowT ma err)
      $ \a -> runCont (runThrowT (f a) err) c

instance (Monad m) => ConjoinedMonads (ThrowT r m) where
  cpure = pure
  cbind = (>>=)

  cthrow = throwT
  ccatch = catchT

-- and similarly for `ThrowT`

```

□

## 8.4 Error-explicit IO

As we saw in section 4.5, `IO` is defined as a `State Monad` with some magical primitive operations.<sup>1</sup> Which means there is nothing preventing us from extending that `IO` signature with a type for errors.

```
newtype EIO e a
```

Similarly to parser combinators of section 8.2 there are several possible implementations of this `EIO` (including, in principle, the ones that do backtracking on errors, though, of course, that would be inconsistent with the semantics of the `RealWorld`). The simplest one matches a definition for non-backtracking parser combinator on `State# RealWorld` from section 5.2.2

```

newtype EIO e a = EIO
  { runEIO :: State# RealWorld
    -> (# Either e a, State# RealWorld #) }

instance Pointed (EIO e) where
  pure a = EIO $ \s -> (# Right a, s #)

instance Monad (EIO e) where
  m >>= f = EIO $ \s -> case runEIO m s of
    (# Left a, s' #) -> (# Left a, s' #)
    (# Right a, s' #) -> runEIO (f a) s'

-- Note how symmetric this is with Pointed and Monad instances.
throwEIO :: e -> EIO e a
throwEIO e = EIO $ \s -> (# Left e, s #)

catchEIO :: EIO e a -> (e -> EIO f a) -> EIO f a
catchEIO m f = EIO $ \s -> case runEIO m s of
  (# Left a, s' #) -> runEIO (f a) s'
  (# Right a, s' #) -> (# Right a, s' #)

```

<sup>1</sup> Some of which actually break `Monad` laws, but as mentioned in remark 4 that is out of scope of this discussion.

Note that very similar structures were proposed before in [44] and `Control.Monad.Exception.Catch` module of `control-monad-exception` [45] discussed in section 5.4. Also note that the definition of GHC’s `IO` before imprecise exceptions were introduced was similar to `EIO` above (but without the parameter `e`) and one of the primary motivations behind introduction of builtin exceptions into GHC mentioned in [90] was to make `IO` more efficient by allowing its `(>>=)` to be implemented without pattern-matching. But there are, of course, other ways to eliminate pattern matching. By moving `Either` in the definition of `EIO` out the parentheses using the technique from section 5.2.2 and then Scott-encoding the resulting type we can make the following definition

```

newtype SEIO e a = SEIO
  { runSEIO :: forall r
    . (e -> State# RealWorld -> r)
    -> (a -> State# RealWorld -> r)
    -> State# RealWorld
    -> r }

instance Pointed (SEIO e) where
  pure a = SEIO $ \err ok s -> ok a s

instance Monad (SEIO e) where
  m >>= f = SEIO $ \err ok s -> runSEIO m err (\a -> runSEIO (f a) err ok) s

-- Note the same here.
throwSEIO :: e -> SEIO e a
throwSEIO e = SEIO $ \err ok s -> err e s

catchSEIO :: SEIO e a -> (e -> SEIO f a) -> SEIO f a
catchSEIO m f = SEIO $ \err ok s -> runSEIO m (\e -> runSEIO (f e) err ok) ok s

```

**Theorem 12.** *Both `EIO` and `SEIO` with the above operations are `ConjoinedMonads`.*

*Proof.* Consequence of theorem 10 and the fact that Scott-encoding preserves computational properties.

```

instance ConjoinedMonads EIO where
  cpure = pure
  cbind = (>>=)

  cthrow = throwEIO
  ccatch = catchEIO

instance ConjoinedMonads SEIO where
  cpure = pure
  cbind = (>>=)

  cthrow = throwSEIO
  ccatch = catchSEIO

```

□

## 8.5 Conventional IO

**Theorem 13.** *`IO` is a composition of `ConjoinedMonads` that is constant in its error index with `toException` in `raiseIO#` and `handleOrThrowAgain` in `catch#`.*



*Proof.* A consequence of results of theorems 9 and 12 for `e == SomeException`. □

Note that, according to remark 4, the above works out only because `raiseIO#/throwIO`, unlike `raise#/throw`, are deterministic (see section 4.5).

Also note that in a dialect of Haskell with separate operators for imprecise exceptions (or without imprecise exceptions altogether) we can completely replace `IO` with `EIO` as defined above. We can not, however, apply that construction to GHC's Haskell dialect since it merges precise and imprecise `catch` (see remark 5).

## Chapter 9

# Cartesian Products

Now let us once more turn our attention to the bodies of definitions 3, 4, and 5 (all of which define the same structure).

```
class (forall a . Monad (\e -> m e a) -- `Monad` in `e`
      , forall e . Monad (\a -> m e a)) -- `Monad` in `a`
  => ConjoinedMonads m where
```

Since `ConjoinedMonads` is simply a `Monad × Monad` with interaction laws between `pure` and `bind` operators (definition 3) it is natural to ask what would happen if we replace one or both of those `Monads` with more general structures like `Applicative` and modify the interaction laws accordingly.

The two structures with `Applicative` in index `e` seem to be unusable for the purposes of this work since they lack conventional error handling operators. However, the structure with `Monad` in index `e` and `Applicative` in index `a` looks interesting.

```
class (forall a . Monad (\e -> m e a))
      , forall e . Applicative (\a -> m e a)
  => MonadXApplicative m where
```

In this structure the `Monad` index gives `throw` and `catch` operators, and the `Applicative` index can be treated as expressing generalized function application (see section 4.1.2) for structure `m`. In other words, such a structure, at the very least, can be used to express  $\lambda$ -calculus with exceptions by simply injecting all `pure` values and `lifting` all pure functions into it. Moreover, since `ConjoinedMonads` is a special case of `MonadXApplicative`, all `ConjoinedMonads` instances from the previous sections can also be used as a basis for such a formalism. However, there are some interesting instances of `MonadXApplicative` that are not `ConjoinedMonads`. Therefore, the question of interaction laws for this structure is also rather interesting.

Consider a folklore example of an `Applicative` that is not a `Monad`: “computations collecting failures in a `Monoid`” (we used the same idea in `Alternative` instances of sections 5.2 and 8.2), which can be defined as follows

```
newtype EA e a = EA { runEA :: Either e a }

instance Pointed (EA e) where
  pure = EA . Right

instance Monoid e => Applicative (EA e) where
  f <*> a = EA $ runEA f <***> runEA a where
    (Right f) <***> (Right a) = Right $ f a
    (Right f) <***> (Left e)  = Left e
    (Left e) <***> (Right a) = Left e
    (Left e1) <***> (Left e2) = Left $ e1 `mappend` e2
```

Note that this structure, like `Either`, can also be made a `Monad` in index `e`

```
throwEA :: e -> EA e a
throwEA = EA . Left

catchEA :: EA e a -> (e -> EA f a) -> EA f a
(EA a) `catchEA` f = case a of
  Right a -> pure a
  Left e -> f e
```

but, unlike for `Either`, from the two interaction laws of definition 3 only the

```
pure a `catch` f == pure a
```

survives. The law for `throw` for this structure, if we are hard-set to write it down, looks like this

```
f <*> throw e == throw e' where
  e' == e'' `mappend` e, for some e''
```

For the similar reasons it is not immediately obvious how to make imprecise exceptions into an instance of `MonadXApplicative` since

- they are non-deterministic, hence disobeying all naive laws for `throw`, and
- imprecise `throw` has a wrong type to be the identity element for `catch` (see remark 5).

However, if we now remember that

- graded monads [52] also require `e` to be a `Monoid` and
- imprecise exceptions, too, can be thought as producing a `Monoid` of possible errors with `catch` (including the implicit `catch` over `main`) “observing” one of its elements,

we shall come to a conclusion that in a calculus with `IO`-effects separated from non-determinism-effects, imprecise exceptions over non-deterministic `Applicative` computations, indeed, form a `Monad` (with equivalence defined up to raising the same set of exceptions, similarly to section 4 of [90]) over the `Monoid` of imprecise exceptions. That is, those, too, are examples of `MonadXApplicative` (with similarly complicated interaction laws for `throw`).

In part III we shall see that some interesting parser-combinator-related structures can be expressed by generalizing the `Applicative`. Similarly, in part IV we shall also see that even more interesting parser-combinator-related structures can be expressed by generalizing the `Monad`.

Thus, in general, the main observation is the following: **a lot of interesting structures that involve error handling are Cartesian products of `Applicative`, `Monad`, and/or their generalizations, with interaction laws that vary on case-by-case basis.** We shall return to this point numerous times.

# Chapter 10

## Discussion

The author hopes that with this part of the work we pointed and then at least partially plugged an algebraic hole in the programming languages theory by showing that

- exception handling without dynamic dispatch is dual to dual to conventional **Monadic** sequential computation, and
- conventional computational formalisms with **throw/try/catch**-exceptions are “conjoined” Cartesian products of pairs of **Monads** (or, less imperatively, **Monads** and **Applicatives**).

In author’s opinion, these facts make a lot of conventional programming “click into place” similarly to how plain **Monads** “click” imperative “semicolons”. Moreover, they provide an algebraic foundation for the argument against building new languages with builtin dynamic dispatch of exception handlers and/or an argument against extensively relying on that feature in the languages that have it, a point which is commonly discussed in the folklore (“exceptions are evil”) and was articulated by Hoare from programmer comprehension standpoint already in 1981 [42]. Not only dynamic dispatch of exceptions is, citing Hoare, “dangerous”, but it also prevents programs from directly accessing the inherent **Monadic** structures discussed in this work.

Also note that everything in this part (as well as in the rest of the work), including **EIO** of section 8.4, follows the “marriage” framework of [111] of confining effects to monads, but ignores the question of any additional rules for type indexes in question. In other words, ad-hoc exception encoding constructions like that of error-explicit IO [44] or graded monads [52] are mostly orthogonal to our “conjoined” structures and can be used simultaneously. This observation makes all the usual arguments against using **Monads** for error handling rather moot.

- The problem of syntactic non-uniformness between pure computations, **Applicatives** and **Monads** is almost trivial to solve: common primitives like **map/mapM** should be expressed in terms of **Applicatives** (of which pure functions are trivial instance) instead of **Monads**. For instance, **mapM** for lists<sup>1</sup> can be rewritten as

```
mapAp :: Applicative f => (a -> f b) -> [a] -> f [b]
mapAp f [] = pure []
mapAp f (a:as) = fmap (:) (f a) <*> mapAp f as
```

Meanwhile, the uniform syntax for pure functions and **Applicatives** can be made by adding some more missing instances of the LISP macros into the compiler in question.<sup>2</sup> For instance, quasiquotation [71] is one conventional way do such a translation, Conal Elliot’s “Compiling to Categories” [22] provides another categorically cute way to achieve similar results.

---

<sup>1</sup> And, similarly, for **Traversable** and so on.

<sup>2</sup> From a cynical LISP-evangelist point of view, all of “the progress” of the programming languages in the last 50 years can be summarized as “adopting more and more elements (lately, meta-programming) from LISP while trying very hard not to adopt the syntax of LISP”. From a less cynical perspective, the progress, at least in typed languages, consists of well-typing said elements.

- The problem of modularity as stated by Brady [12]

Unfortunately, useful as monads are, they do not compose very well. Monad transformers can quickly become unwieldy when there are lots of effects to manage, leading to a temptation in larger programs to combine everything into one coarse-grained state and exception monad.

can be solved by applying graded monads to the `Monad` part of `MonadXApplicative` conjoined product.

In other words, a programming language that

- provides a primitive `catch` operator that does no dynamic dispatch (or, alternatively, provides no builtin error handling at all),
- provides quasi-quoting/compiling to categories for generalized `Applicatives`,
- distinguishes between `IO`-effects and non-determinism,
- allows one to express Cartesian products of type classes, and, possibly,
- uses a graded `MonadXApplicative` for a base type of computations

could provide all the efficiency of imprecise exceptions, simplicity of `Monads` (doubled, in some sense, since error handling would stop being special), while having none of the usual arguments against said mechanisms applying to it.

## Part III

# Transforming Trees with Generalized Applicative Expressions

# Chapter 11

## Motivation

Let us recall the definition of `Applicative` type class [74] (section 4.1.2) as it is currently defined in the `base` [26] package of `Hackage` [39]

```
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 part of the work.) 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 chapters 4 and 5, and [40]), most (for the purposes of this work) notably, including `Applicative` parser 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` [66], `Attoparsec` [88] or `Megaparsec` [51] 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

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

While clearly limited to simple data types of a single<sup>1</sup> 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

---

<sup>1</sup> Two or more constructors can be handled with the help of `Alternative` type class and some tagging of choices.

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 part 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 (which, in particular, is rather useful for expressing isomorphisms and automorphisms of such types [99]) 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 11.1 provides some motivating examples that show why we want to use `Applicative-like` computations to express transformations between data types. Section 12.1 formalizes the notion of “`Applicative-like`” and discusses the properties we expect from such expressions. Section 12.2 derives one particular structure for one of the motivating examples using LISP-encoding for deconstructing data types. Section 12.3 proceeds to derive the rest of motivating examples by applying the same idea, thus showing that section 12.2 describes a technique, not an isolated example. Section 12.3 ends by demonstrating the total expressive power of the technique. Section 12.4 repeats the derivation and the implementations for Scott-encoded data types. Section 12.5 observes the general structure behind all of the terms used in the document. Section 12.6 gives a formal description of the technique and the underlying general algebraic structure. Chapter 13 shows how this technique can be combined with `Monad` error handling of part II. Chapter 14 discusses the consequences and the general picture.

## 11.1 Motivating Examples

Consider the following expressions produced with the help of first author’s favorite `safecopy` [41] data-type-to-binary serialization-deserialization library which can be used to deserialize-serialize `Device` with the following code snippet (simplified<sup>2</sup>)

```
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. [85],
- 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` [31], which is an instance of `Applicative`.<sup>3</sup>

Another example is the data-type-to-JSON-to-strings serialization-deserialization part of `aeson` [89] library which gives the following class signatures to its deserializer and serializer from/to JSON respectively.

---

<sup>2</sup> 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.

<sup>3</sup> We shall utilize this fact in the following sections.



```

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`.<sup>4</sup>

Other serialization-deserialization problems, e.g. conventional pretty-printing with the standard `Show` type class [26] 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

```

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

---

<sup>4</sup> As noted under footnote 3 and demonstrated in detail in section 12.2. 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 `Monadic` parser combinators discussed in part IV.

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 [64], Uniplate [78], Multiplate [87] or Lenses [58, 61] are not particularly useful in this case.<sup>5</sup> 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?

---

<sup>5</sup> Strictly speaking, both operations used in the “sum” part of `average` are `Monoid` operators, so generalized `zips` provided by some of the mentioned libraries can be used to implement that part, but the “divide” part is not so homogeneous.

# Chapter 12

## Derivation

### 12.1 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

- `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 `pieces` 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 a mechanism to handle (parse, pretty-print, etc) the `full` structure that exposes “ports” that subcomputations `plug` with computations that handle different `pieces`.

**Remark 6.** *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 **pieces**.

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

## 12.2 Deriving the Technique

We shall now demonstrate the derivation of the main technique of the document. 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 12.4).

```
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 `Strings` 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
-- == "False 19 1"
```

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` (remember footnote 3)

```
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 :: (r -> b) -> r -> ([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
-- == "False 19 1"
```

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 `Strings`. 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
```

## 12.3 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 -> (r -> b) -> r -> (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
  -- == Device True 119 201
```

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
  -- == Device False 138 202

```

The above transformations combined with

```

unDevice = unDeviceLISP
unBenchmark = unBenchmarkLISP

```

implement all the examples from section 11.1, thus solving the problem as it was originally described.

Note, however, that the above technique can be trivially extended to **chopping** 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
-- == Device True 1 1
```

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 use case.

## 12.4 Scott-encoded Representation

The LISP-encoding used above is not the only generic representation for data types, in this section we shall repeat the above results for Scott-encoded data types.

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 (see section 5.1.2), 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  $(a \rightarrow b \rightarrow c \rightarrow \dots \rightarrow z) \rightarrow z$  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.<sup>1</sup>

```
chopS :: ((s -> a -> b) -> c)
       -> (s -> a -> t)
```

---

<sup>1</sup> 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`.



```

    -> ((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
    -- == testShowDevice

```

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
    -- == testMapDevice

```

The most interesting part, however, is the reimplementing 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` (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
  -- == testZipDevice

```

thus, again, implementing all the examples from section 11.1, but now purely with Scott-encoded data types.

**Remark 7.** *Note that while the transformation from  $b$  to  $(a, b)$  for the LISP-encoding or the plain tuples is regular, the transformation from  $(a \rightarrow b \rightarrow c \rightarrow \dots \rightarrow z) \rightarrow z$  to  $(s \rightarrow a \rightarrow b \rightarrow c \rightarrow \dots \rightarrow z) \rightarrow z$  is not, the former is not a subexpression of the latter. Taking that into account, the author feels 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 similar transformations over Scott-encoded data types. The closest works we are aware of that do vaguely related things with Scott-encoded values are [95] and [19] (though, neither explicitly mentions the fact).*

## 12.5 General Case

Curiously, note that with the aforementioned order of `consS` `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 document and the generalizations of **Applicative**-like `zips` to larger numbers of structures can be uniformly produced from just `chop` and `chopS`.

## 12.6 Formal Account

The derivation of section 12.2, 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 12.2 and 12.3 or Scott-encoded representation in section 12.4),
- 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 12.3 showed that, in general, those expressions can express arbitrary 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 12.5 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 folding on, inductively speaking), the rest are simply baggage used internally by the corresponding operators.

### 12.6.1 Dependently-typed Applicative

Now, the obvious question is how a general structure unifying all those operators would look. Firstly, remember that the `pure` function of `Applicative` can be separated out into its own type class (see section 4.1.2)

```
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` [26] define `Category` [26] and `ArrowApply` [43] 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.

### 12.6.2 Haskell Encoding

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

```
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
```

Note however, that since our analogues of `pure` operator simply wrap results produced by the data type destructors into corresponding initial states, their 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

```
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. Thus, `ApplicativeLike`, unlike `Applicative`, should not be `Pointed`.

The rest of the section demonstrates the use of the above generic encoding.

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

```
instance ApplicativeLike (Mapper r (x -> y)) where
  type C (Mapper r (x -> y)) a b = (a, b)
  type G (Mapper r (x -> y)) a = a -> x
  type F (Mapper r (x -> y)) b = Mapper r 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
```

```
newtype Printer r a = Printer
  { runPrinter :: r -> ([String], a) }
```

```
instance ApplicativeLike (Printer e) where
  type C (Printer e) a b = (a, b)
  type G (Printer e) b = b -> String
  type F (Printer e) b = Printer e b
  f <*> g = Printer $ showa (runPrinter f) g
```

```

showDeviceG :: Printer Device ()
showDeviceG = Printer (depureShow unDeviceLISP)
  <*> show
  <*> show
  <*> show

testShowDeviceG :: String
testShowDeviceG = runShow $ runPrinter showDeviceG exampleDevice

newtype PrinterS c r a = PrinterS
  { runPrinterS :: r -> ([String] -> a) -> c }

instance ApplicativeLike (PrinterS c e) where
  type C (PrinterS c e) a b = a -> b
  type G (PrinterS c e) a = a -> String
  type F (PrinterS c e) b = PrinterS c e b
  f <*> g = PrinterS $ showaS (runPrinterS f) g

showDeviceGS :: PrinterS c Device c
showDeviceGS = PrinterS (depureShowS unDeviceScott)
  <*> show
  <*> show
  <*> show

testShowDeviceGS :: String
testShowDeviceGS = runShowS $ runPrinterS showDeviceGS exampleDevice

newtype Zipper ra rb f a = Zipper
  { runZipper :: ra -> rb -> (f, a) }

-- we need these, because we used tuples in `zipa`
toZipper f = Zipper $ \ra rb -> (\(a, b, c) -> ((a, b), c)) $ f ra rb
fromZipper f = \ra rb -> (\((a, b), c) -> (a, b, c)) $ runZipper f ra rb

instance ApplicativeLike (Zipper e e (x -> y, (a, b))) where
  type C (Zipper e e (x -> y, (a, b))) c d = (c, d)
  type G (Zipper e e (x -> y, (a, b))) c = a -> c -> x
  type F (Zipper e e (x -> y, (a, b))) d = Zipper e e (y, b) d
  f <*> g = toZipper $ zipa (fromZipper f) g

zipDeviceG :: Zipper Device Device (Device, ()) ()
zipDeviceG = toZipper (depureZip Device unDeviceLISP unDeviceLISP)
  <*> (&&)
  <*> (+)
  <*> (+)

testZipDeviceG :: Device
testZipDeviceG = runZip $ (\((a, b), c) -> (a, b, c))
  $ runZipper zipDeviceG exampleDevice testMapDeviceG

newtype ZipperS z z' ra rb f g a = ZipperS
  { runZipperS :: ra -> rb -> (((f -> g) -> z) -> a) -> z' }

```

```

instance ApplicativeLike (ZipperS z z' e e (x -> y) (a -> b)) where
  type C (ZipperS z z' e e (x -> y) (a -> b)) c d = c -> d
  type G (ZipperS z z' e e (x -> y) (a -> b)) c = a -> c -> x
  type F (ZipperS z z' e e (x -> y) (a -> b)) d = ZipperS z z' e e y b d
  f <*> g = ZipperS $ zipaS (runZipperS f) g

zipDeviceGS :: ZipperS z z' Device Device Device z z'
zipDeviceGS = ZipperS (depureZipS Device unDeviceScott unDeviceScott)
  <*> (&&)
  <*> (+)
  <*> (+)

testZipDeviceGS :: Device
testZipDeviceGS = runZipS $ runZipperS zipDeviceGS exampleDevice testMapDeviceGS

```

# Chapter 13

## Cartesian Products

If one is to take the `ApplicativeLike` of section 12.6.2 seriously, that is, if one is to use `<*>` as *the* function application operator then, as noted in chapter 9, it would make a lot of sense to take a Cartesian product of `Monad` with `ApplicativeLike` to supply such a formalism with error handling operators.

To archive such a Cartesian product we just need to add another index signifying errors to our state transformer and use that index in a composition with something that is a `Monad` in that index.

For instance, for LISP-encoded data types and `Either` this gives us the following definition for the corresponding analogue of the `chop` operator

```
chopE' :: Either e (s, (a, b)) -> (s -> a -> Either e t) -> Either e (t, b)
chopE' (Left e) _ = Left e
chopE' (Right (s, (a, b))) f = case f s a of
  Left e -> Left e
  Right fsa -> Right (fsa, b)
```

or, alternatively, explicitly using the fact that `Either` is a `Monad`

```
chopE :: Either e (s, (a, b)) -> (s -> a -> Either e t) -> Either e (t, b)
chopE i f = do
  (s, (a, b)) <- i
  fsa <- f s a
  pure (fsa, b)
-- == chopE'
```

(note that `chopE` is exactly the definition of `chop` of section 12.2 with all `let`-bindings transformed into `>>=`-bindings), which gives the following definition of the corresponding analogue for `mapa`

```
mapaE :: (r -> Either e (x -> y, (a, b)))
  -> (a -> Either e x)
  -> (r -> Either e (y, b))
mapaE = homWrap $ \st f -> chopE st $ \s a -> pure s <*> f a
```

(again, note that `mapaE` is exactly `mapa` with body lifted to `Applicative`), which can be encoded using the generalized encoding of of section 12.6.2 as

```
newtype MapperE e r f a = MapperE { runMapperE :: r -> Either e (f, a) }

instance ApplicativeLike (MapperE e r (x -> y)) where
  type C (MapperE e r (x -> y)) a b = (a, b)
  type G (MapperE e r (x -> y)) a = a -> Either e x
  type F (MapperE e r (x -> y)) b = MapperE e r y b
  f <*> g = MapperE $ mapaE (runMapperE f) g
```



```

mapDeviceGE :: MapperE String Device Device ()
mapDeviceGE = MapperE (Right . depureMap Device unDeviceLISP)
  <*> (\x -> if x then pure x else throwE' "bad")
  <*> (\x -> pure (x + 100))
  <*> (\x -> pure (x + 200))

testMapDeviceGE :: Either String Device
testMapDeviceGE = fmap runMap $ runMapperE mapDeviceGE exampleDevice

```

Moreover, note that the above body of `chopE` does not mention anything specific to `Either`, thus it can be reused for all instances described in chapter 8 without modifications. The following shows that the same property also holds for Scott-encoded data types. Similarly to section 12.4 where we had to `cons` to the left to get multiple stacks, in this case we also need to push the `Either` (or another instance from chapter 8) to the leftmost position

```

chopES :: ((Either e s -> a -> b) -> c)
         -> (s -> a -> Either e t)
         -> ((Either e t -> b) -> c)
chopES i f o = i $ \s a -> o (s >>= \s' -> f s' a)

mapaES :: (r -> (Either e (x -> y) -> a -> b) -> c)
        -> (a -> Either e x)
        -> (r -> (Either e y -> b) -> c)
mapaES = homWrap $ \st f -> chopES st $ \s a -> pure s <*> f a

```

Now, note that Scott-encoding allows not only simple consing but it also allows one to apply a function to the element in the head

```

consApp :: (s -> t)
         -> ((s -> a) -> b)
         -> (t -> a) -> b
consApp st sab ta = sab $ \s -> ta (st s)

```

which allows to straightforwardly translate the other examples

```

newtype MapperES c e r f a = MapperES
  { runMapperES :: r -> (Either e f -> a) -> c }

instance ApplicativeLike (MapperES c e r (x -> y)) where
  type C (MapperES c e r (x -> y)) a b = a -> b
  type G (MapperES c e r (x -> y)) a = a -> Either e x
  type F (MapperES c e r (x -> y)) b = MapperES c e r y b
  f <*> g = MapperES $ mapaES (runMapperES f) g

mapDeviceGES :: MapperES c String Device Device c
mapDeviceGES = MapperES (consApp Right . depureMapS Device unDeviceScott)
  <*> (\x -> if x then pure x else throwE' "bad")
  <*> (\x -> pure (x + 100))
  <*> (\x -> pure (x + 200))

testMapDeviceGES :: Either String Device
testMapDeviceGES = runMapS $ runMapperES mapDeviceGES exampleDevice

```

*-- and similarly for zip*

Of course, in practice, for Scott-encoded case one is likely to use Scott-encoded version of `Either` instead of the plain one, but the term of `chopES` above shows that it is of no consequence: as with `chopE`, `chopES` does not mention `Either`, thus that term can be reused for other instances of chapter 8 by changing the type.

**Theorem 14.** *Therefore, all of the `ApplicativeLike` structures described in this part can be extended with another index signifying errors in such a way that the resulting structures become Cartesian products of `Monad` and `ApplicativeLike`, with all the benefits described in chapter 9.*

*Proof.* By the above reasoning.

□

# Chapter 14

## Discussion

From a practical perspective, in this part 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 express 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, `maps`, and `zips` 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 8.** *By the way, note that Haskell’s `GHC.Generics` [27] is not an adequate replacement for LISP- and Scott-encoded representations used in the document: 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 part we have presented a natural generalization of the conventional `Applicative` [74] type class (which can be viewed as a generalization of conventional function application) into dependent types with generalized arrow of `Category/ArrowApply` [26, 43]. Both `Applicatives` and `Monads` [79, 80, 110] (that can be viewed as a generalization of the conventional sequential composition of actions, aka “imperative semicolon”) were similarly generalized to superapplicatives and supermonads in [11]. In particular, [11] 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 `Functors` (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 9.** *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  $a == b \Rightarrow f a == f b$ ).*

In other words, our `ApplicativeLike` can be viewed as a simpler encoding for generalized superapplicatives of [11] when those are treated syntactically rather than algebraically (since we completely ignore `Functors`).

## Part IV

# Transforming Trees with Indexed Monads

# Chapter 15

## Motivation

Conventional **Monadic** parser combinators, as noted in section 5.2, allow us to express arbitrary transformations from **Streams** to arbitrary data types. Conventional **Applicative** parsers, as noted in chapter 11, allow us to very conveniently express simple transformations from **Streams** to simple data types of a single constructor. **ApplicativeLike** structures, which are dependently typed generalizations of **Applicatives**, introduced in part III allow us to very conveniently express simple transformations from simple data types of a single constructor to **Streams**, simple **fold**s between such data types, and, as noted in section 12.5, somewhat less conveniently, arbitrary transformations between such data types. Clearly, a generalization of **Monadic** parser combinators that would allow us to express arbitrary transformations between arbitrary data types would nicely complete the picture. In this part we shall describe such a generalization.

Section 15.1 provides some motivating examples. Chapter 16 derives said generalizations of **Monadic** parser combinators. Chapter 17 provides example usages, including the implementation of the motivating examples. Chapter 18 discusses the consequences and the general picture.

### 15.1 Motivating Examples

Consider the following data type encoding for XML documents

```
data XML = Plain String
         | Node String [(String, String)] [XML]
```

where **Plain** represents inner plain text data and **Node** represents an XML node with its parameters and children nodes. For instance,

```
exampleXHTML :: XML
exampleXHTML =
  Node "html" [("lang", "en")]
    [ Node "body" []
      [ Node "h1" [] [Plain "main title"]
      , Node "div" [("class", "content")] $
        [ Node "h2" [] [Plain "internal header"]
        , Node "p" [] [Plain "some text"]
        , Node "h2" [] [Plain "another internal header"]
        , Node "p" [] [Plain "more text"]
        ]
      ]
    ]
```

Alternatively, consider the following representation of the same thing

```
data XMLElement = InnerText String
                | StartTag String [(String, String)]
```

```
    | EndTag String
```

```
type XMLStream = [XMLElement]
```

```
exampleXHTMLStream :: XMLStream
```

```
exampleXHTMLStream =  
  [ StartTag "html" [("lang", "en")]  
  , StartTag "body" []  
  , StartTag "h1" []  
  , InnerText "main title"  
  , EndTag "h1"  
  , StartTag "div" [("class", "content")]  
  , StartTag "h2" []  
  , InnerText "internal header"  
  , EndTag "h2"  
  , StartTag "p" []  
  , InnerText "some text"  
  , EndTag "p"  
  , StartTag "h2" []  
  , InnerText "another internal header"  
  , EndTag "h2"  
  , StartTag "p" []  
  , InnerText "more text"  
  , EndTag "p"  
  , EndTag "div"  
  , EndTag "html"  
  ]
```

One probably would not be surprised to find either data type and a deserializer/serializer from/to either representation in a library for handling XML-encoded data.

Now, consider, for instance, the following problem: given a `String` that contains a serialized XML document and a library that can parse such `Strings` into either of the above representations, produce a list of inner texts of `h2` headers containing a single plain text node inside. How does one go about archiving that? Conventionally, one either

- parses a given `String` into an element of `XML` type using the library, thus offloading all the “incorrectly formatted input” type of problems to the library, and then processes the result by inventing a specialized function that does a lot of `concatMap`; or
- parses a given `String` into `XMLStream` using the library, and then treats the result as an input to a conventional parser built with the help of some parser combinator library, which allows one to easily and generically extract inner texts following `StartTag "h2"` elements, but requires duplication of all the error handling for “incorrectly formatted input” type of problems the library already has; or
- parses into `XML`, with usual benefits, and then accumulates results with some generalized traversal and pattern matching mechanism.

The issue becomes even more apparent when one has a tree structure which one can not simply `traverse` (`Traversable` [26]) to collect some results. Say, for instance, one needs to parse a `String` representing a program into its abstract syntax tree (AST) and then apply some carefully crafted type-respecting rewrite rules to perform some program optimizations. Assuming one wants to be able to change the type of AST without rewriting everything, one is now conventionally required to use something like SYB [64], Uniplate [78], Multiplate [87], or Lenses [58, 61], possibly combined with some generalized pattern-matching machinery similar to [95].

Which is not to say that doing any of the above is inherently bad, but when a compiler you wrote parses its inputs into ASTs with **Monadic** and/or **Applicative** parser combinators and emits results with some **ApplicativeLike** machinery, all of which are just fancy state transformers, one starts to wonder whether the same “hammer” can be used for the term rewriting “nails” in the middle too.

## Chapter 16

# Derivation (and Cartesian Products)

The logic behind the design of a generalized structure that can express arbitrary tree transformers **Monadically** is rather straightforward. We know that conventional **Monadic** parser combinators like **Parsec**, **Attoparsec**, and **Megaparsec** are made of amalgamations of **State**

```
newtype State s a = State { runState :: s -> (a, s) }
```

with **Either** (then, usually, Scott-encoded). Therefore, since we now want state transformers that work over arbitrary trees instead of just **Streams**, and in section 12.2 we saw that all we needed to archive the same result there was to decouple two type variable entries of **s**, all we have to do now is to perform the same modification to **State**, thus archiving the exact definition of **IxState** from section 5.3.1

```
data IxState i j a = IxState { runIxState :: i -> (a, j) }
```

Finally, since here we, too, need to handle errors, we need to amalgamate **Either** into it. As before, there are several possibilities here which we shall discuss in the following sections.

Before that, however, note that we are now essentially discussing translations of conventional **Monadic** parser combinators of section 5.2 which we generalized with **Monadic** error handling in section 8.2 into indexed **Monad** (**IxMonad**) territory discussed in section 5.3.

### 16.1 Instance: Simple Indexed Monadic Parser Combinator

Adaptation of **SParser** parser combinator of section 5.2.1 to the indexed **IxMonad** case is straightforward, all the terms stay the same up to constructor and variable renames

```
newtype IxSParser e i j a = IxSParser  
  { runIxSParser :: i -> Either e (a, j) }
```

```
instance IxPointed (IxSParser e) where  
  ipure a = IxSParser $ \i -> Right (a, i)
```

```
instance IxMonad (IxSParser e) where  
  p >>=+ f = IxSParser $ \i ->  
    case runIxSParser p i of  
      Left x -> Left x  
      Right (a, j) -> runIxSParser (f a) j
```

As with **SParser**, **IxSParser** has a single implementation for **Monad** in index **e** that form a plain **Monad**.



```

throwIxSP :: e -> IxSParser e i j a
throwIxSP e = IxSParser $ \_ -> Left e

-- Note that this keeps indices as is, since it is a `Monad`,
-- not `IxMonad` in `e`
catchIxSP :: IxSParser e i j a -> (e -> IxSParser f i j a) -> IxSParser f i j a
catchIxSP m f = IxSParser $ \i ->
  case runIxSParser m i of
    Right x -> Right x
    Left e -> runIxSParser (f e) i

```

Thus, to define an analogue of `ConjoinedMonads` for `IxSParser`, following the discussion in chapter 9, we need to make a product of `Monad` with `IxMonad`. Skipping all the Haskell encoding discussion of section 7.5 and chapter 9, let us simply define this structure as

```

class MonadXIxMonad m where
  icpure  :: a -> m e i i a
  icbind  :: m e i j a -> (a -> m e j k b) -> m e i k b

  icthrow :: e -> m e i j a
  iccatch :: m e i j a -> (e -> m f i j a) -> m f i j a

-- `(>>>)` in index `a`
iandThen :: MonadXIxMonad m => m e i j a -> m e j k b -> m e i k b
iandThen f g = f `icbind` const g

-- `(>>>)` in index `e`
iorElse :: MonadXIxMonad m => m e i j a -> m f i j a -> m f i j a
iorElse f g = f `iccatch` const g

-- `ifmap` derived for MonadXIxMonad in index `a`, for later simplicity
ifmap' :: MonadXIxMonad m
  => (a -> b) -> m e i j a -> m e i j b
ifmap' f m = m `icbind` \a -> icpure (f a)

-- `iap` derived for MonadXIxMonad in index `a`, for later simplicity
iap' :: MonadXIxMonad m
  => m e i j (a -> b) -> m e j k a -> m e i k b
iap' mf m = mf `icbind` \f -> m `icbind` \a -> icpure (f a)

```

and its instance as

```

instance MonadXIxMonad IxSParser where
  icpure = ipure
  icbind = (>>=)

  icthrow = throwIxSP
  iccatch = catchIxSP

```

**Theorem 15.** *`IxSParser` is a `Monad` in index `e` and an `IxMonad` in index `a`. Moreover, operators of those instances satisfy the interaction laws identical to those of definition 3.*

*Proof.* By case analysis. □

Similarly to section 8.2.2, this structure also has type-precise `some` and `many` operators

```
icsome :: MonadXIxMonad m
  => m e i i a -> m e i i [a]
icsome p = ifmap' (:) p `iiap'` icmany p
```

```
icmany :: MonadXIxMonad m
  => m e i i a -> m f i i [a]
icmany p = icsome p `iorElse` icpure []
```

(However, unlike in section 8.2.2 here we use derived `iiap'` and `ifmap'` combinators instead of using those of `IxApplicative` thus simplifying the type class constraints.)

The most interesting thing about `IxSParser` are the types of `sepBy*` combinators

```
-- zero of more `p` separated by `sep`
isepBy :: MonadXIxMonad m => m e i i a -> m e i i b -> m f i i [a]
isepBy p sep = isepBy1 p sep `iorElse` icpure []
```

```
-- one or more `p` separated by `sep`, that is
-- `p` followed by zero or more `sep >> p`
isepBy1 :: MonadXIxMonad m
  => m e i j a -> m e j i b -> m e i j [a]
isepBy1 p sep = p `icbind` \x ->
  icmany (sep `iandThen` p) `icbind` \xs ->
  icpure (x:xs)
```

Note how in `isepBy1` the parser given in the first argument (`p`) transforms the internal state `i -> j` while the one given in the second (`sep`) does the reverse.

## 16.2 Non-instance: ... with Full Access to the State

On the other hand, adapting `EParser` to the indexed case is not so trivial. Note that a naive adaptation gives two possibilities that use different indexes for the `Left` case

```
newtype IxEParseR e i j a = IxEParseR
  { runIxEParseR :: i -> Either (e, i) (a, j) }
```

```
newtype IxEParseC e i j a = IxEParseC
  { runIxEParseC :: i -> Either (e, j) (a, j) }
```

However, for `IxEParseR` the types force the potential implementation of (`>>=`) to rollback the internal state on failure in the second argument

```
instance IxPointed (IxEParseR e) where
  ipure a = IxEParseR $ \i -> Right (a, i)
```

```
-- not really, violates laws
instance IxMonad (IxEParseR e) where
  p >>= f = IxEParseR $ \i ->
  case runIxEParseR p i of
    Left x -> Left x
    Right (a, j) -> case runIxEParseR (f a) j of
      Left (e, _) -> Left (e, i)
      Right x -> Right x
```

which violates the associativity law of `Monad` (see section 4.2.1), since, for instance

```

let
  modify = IxParserR $ \i -> Right ((), delta i)
  f = modify
  g = throw e
in
(f >>+ g) >>+ h == throw e
-- but
f >>+ (g >>+ h) == modify >>+ throw e

```

which are not equal when  $i \neq \text{delta } i$  for some  $i$ .

Meanwhile, for `IxParserC` the `(>>+)` operator simply has no implementation.

```

-- not really, undefined
instance IxMonad (IxParserC e) where
  p >>+ f = IxParserC $ \i ->
    case runIxParserC p i of
      Left (e, j) -> undefined -- neither `Left (e, i)` nor `Left (e, j)`
      Right (a, j) -> runIxParserC (f a) j

```

### 16.3 Instance: Twice Conjoinedly Indexed Monadic Parser Combinator with Full Access to the State

Interestingly, to make a working adaptation of `EParser` to the indexed case we have to add another index to the data type signature to signify `state-after-error` (denoted as `k` in the following signature)

```

newtype IxParser e i k j a = IxParser
  { runIxParser :: i -> Either (e, k) (a, j) }

```

and then take a product of two `IxMonad` instances for indexes `i j` for the index `a` (as before) and indexes `i k` for the index `e` (which is new, and weird)

```

class IxMonadXIxMonad m where
  iicpure  :: a -> m e i u i a
  iicbind  :: m e i u j a -> (a -> m e j u k b) -> m e i u k b

  iicthrow :: e -> m e i i u a
  iiccatch :: m e i j u a -> (e -> m f j k u a) -> m f i k u a

-- `(>>+)` in index `e`
iiorElse :: IxMonadXIxMonad m => m e i j u a -> m f j k u a -> m f i k u a
iiorElse f g = f `iiccatch` const g

iifmap :: IxMonadXIxMonad m
  => (a -> b) -> m e i u j a -> m e i u j b
iifmap f m = m `iicbind` \a -> iicpure (f a)

iiap :: IxMonadXIxMonad m
  => m e i u j (a -> b) -> m e j u k a -> m e i u k b
iiap mf m = mf `iicbind` \f -> m `iicbind` \a -> iicpure (f a)

```

Arguably, the resulting class signature becomes rather unreadable when encoded using the simplest encoding for GHC Haskell, thus let us also demonstrate how this class would be encoded using the imaginary idealized encoding of section 7.5.

```

class (forall j a . IxMonad (\i k e -> m e i k j a) -- `IxMonad` in `e`
  , forall k e . IxMonad (\i j a -> m e i k j a)) -- `IxMonad` in `a`
  => IxMonadXIxMonad m where

```

**Theorem 16.** *IxEParser* is an *IxMonad* in index *e* and an *IxMonad* in index *a*. Moreover, operators of those instances satisfy the interaction laws identical to those of definition 3.

*Proof.* By case analysis. □

Of especial note here is the fact that this time the two *IxMonads* of this product interact not only with the operational interaction laws of definition 3 like all other similar products but, in some sense, they also interact in types by “sharing” index *i*. Moreover, note that *IxEParser*, unlike *EParser*, admits only a single implementation of the *iicatch*

```

instance IxMonadXIxMonad IxEParser where
  iicpure a = IxEParser $ \i -> Right (a, i)
  iicbind p f = IxEParser $ \i ->
    case runIxEParser p i of
      Left x -> Left x
      Right (a, j) -> runIxEParser (f a) j

  iicthrow e = IxEParser $ \i -> Left (e, i)
  -- analogous to `catchEPC`
  iiccatch m f = IxEParser $ \i ->
    case runIxEParser m i of
      Right x -> Right x
      Left (e, j) -> runIxEParser (f e) j

```

which continues with the latest state on error. Meanwhile, the version that rolls-back has a different type:

```

-- analogous to `catchEPR`
catchIxEPR :: IxEParser e i u j a -> (e -> IxEParser f i k j a) -> IxEParser f i k j a
catchIxEPR m f = IxEParser $ \i ->
  case runIxEParser m i of
    Right x -> Right x
    Left (e, j) -> runIxEParser (f e) i

```

As usual, the types of the corresponding some and many combinators are very informative.

```

iicsome :: IxMonadXIxMonad m => m e i k i a -> m e i k k [a]
iicsome p = iifmap (:) p `iicap` iicmany p

iicmany :: IxMonadXIxMonad m => m e i k i a -> m f i u k [a]
iicmany p = iicsome p `iiorElse` iicpure []

```

Arguably, a bit *too* informative.

## Chapter 17

# Examples

For reasons discussed in chapter 18 we shall ignore `IxParser` and concentrate on examples that can be implemented with `IxSParser`.

Firstly, as usual, if we so desire, we can have our own version of (`<|>`) operator from `Alternative` with error collection in a `Monoid`

```
<|+> :: (MonadIxMonad m, Monoid e)
      => m e i j a -> m e i j a -> m e i j a
f <|+> g = f `iccatch` \e ->
          g `iccatch` \e' ->
          icthrow (e `mappend` e')
```

*-- A useful generalization*

```
class Monoid e => Failure e where
  -- expected `a` but got `b` instead
  expected    :: String -> String -> e

  -- unexpected `a`
  unexpected  :: String -> e

instance Failure [String] where
  expected a b = ["expected " ++ a ++ " got " ++ b]
  unexpected a = ["unexpected " ++ a]
```

Then, of course, we can have both usual operators to manipulate the internal state, similarly to the conventional `State Monad`

```
iget :: IxSParser e i i i
iget = IxSParser $ \i -> Right (i, i)

iput :: j -> IxSParser e i j ()
iput j = IxSParser $ \_ -> Right ((), j)
```

Then, we can have all the usual `Monadic` parser combinators (including `icsome`, `icmany`, `isepBy`, `isepBy1` already discussed in section 16.1)

```
lookAhead :: IxSParser e i j a -> IxSParser f i i (Either e a)
lookAhead p = IxSParser $ \i -> case runIxSParser p i of
  Left e      -> Right (Left e, i)
  Right (a, j) -> Right (Right a, i)

notFollowedBy :: (Failure e, Show a) => IxSParser f i j a -> IxSParser e i i ()
```

```

notFollowedBy p = lookAhead p >>=+ \ma -> case ma of
  Left _ -> icpure ()
  Right a -> icthrow $ unexpected $ show a

endOfInput :: (Failure e, Show i) => IxSParser e [i] () ()
endOfInput = IxSParser $ \i -> case i of
  [] -> Right ((), ())
  (i:_) -> Left $ expected "end of input" (show i)

anything :: Failure e => IxSParser e [i] [i] i
anything = IxSParser $ \i -> case i of
  [] -> Left $ unexpected "end of input"
  (i:is) -> Right (i, is)

-- etc

```

Now, to solve our motivational “inner text in h2 nodes” problem we just need a couple of generic combinators

```

exhaust :: IxSParser e i j a -> IxSParser e [i] () [a]
exhaust p = IxSParser $ \is -> Right (go is, ()) where
  go [] = []
  go (i:is) = case runIxSParser p i of
    Left _ -> go is
    Right (a, _) -> a:(go is)

find :: Failure e
      => IxSParser e i () a
      -> IxSParser e i [i] b
      -> IxSParser e i () [a]
find p w = (p >>=+ \a -> icpure [a])
  `iccatch`
  (\_ -> w >>=+ const (exhaust (find p w)) >>=+ \ps -> icpure (mconcat ps))

```

and a couple of trivial XML-specific combinators

```

plain :: Failure e => IxSParser e XML () String
plain = IxSParser $ \i -> case i of
  Plain s -> Right (s, ())
  Node a _ _ -> Left $ expected "inner plain text node" ("`" ++ a ++ "' node")

node :: Failure e => IxSParser e XML [XML] String
node = IxSParser $ \i -> case i of
  Node s _ w -> Right (s, w)
  _ -> Left $ expected "node" "inner plain text"

```

which, taken together, allow us to write

```

single :: (Failure e, Show i) => IxSParser e [i] i ()
single = anything >>=+ \a -> endOfInput >>=+ const (iput a)

childOf :: Failure e
         => IxSParser e XML () String -> String -> IxSParser e XML () String
childOf p n = node >>=+ \a -> if a == n
  then single >>=+ const p

```

```

else icthrow $ expected n a

allPlainH2 :: IxSParser [String] XML () [String]
allPlainH2 = find (plain `childOf` "h2") node

testIxSParser = runIxSParser allPlainH2 exampleXHTML
                == Right (["internal header","another internal header"],())

```

Moreover, note that the results of part III can also be adapted to this structure. **ApplicativeLike** structures describe generalized multi-stack machines, **IxSParser** can similarly express them by hiding those “stacks” in its indexes. In particular, note that **chopE** of chapter 13 is, essentially, the  $(i \rightarrow j) \rightarrow \text{IxSParser } e \ i \ j \ a$  that was turned on its head.

## Chapter 18

# Discussion

From a practical perspective, in this part we have shown that **Monadic** parser combinators can be generalized to “parse” and transform between arbitrary data types. From a theoretical perspective, we have demonstrated another couple of interesting Cartesian products (**MonadXIxMonad** and **IxMonadXIxMonad**) with their instances.

Note however, that while **IxEParser** is an instance of a pinnacle of indexed **Monadic** structures discussed in this work, namely **IxMonadXIxMonad**, and, clearly, that structure can be used to implement arbitrary transformations between data types, the author feels like from the parser combinator standpoint **IxEParser** is an instance of “science has gone too far”. As discussed section 5.2.3 (especially around footnote 3), in author’s opinion, non-rolling-back parsers combinators are just too hard to reason with and **IxEParser** does not give a choice in the matter of rollback semantics. One can argue that the “I-have-to-wrap-everything-with-try-combinator” problem of section 5.2.3 can be solved by implementing a variant of (`<|>`) operator of **Alternative** that would do it automatically, but even then, the non-rolling-back **catch** is similarly hard to reason with: for instance, in which state the **b** of `(a `catch` e) >> b` starts?

Nevertheless, **IxMonadXIxMonad** is an interesting Cartesian product and the types of **iicsome** and **iicmany** operators of **IxEParser** are very informative, regardless of their practicality.

Finally, note that while for purely **Monadic** parser combinators **SParser** was, essentially, a special case of **EParser** (see section 8.2), in the indexed **Monad** case **IxSParser** is not a special case of **IxEParser** since `catchIxEPR` is not an instance of **iiccatch**.



## Conclusions and Future Work

In short, in this work we have shown 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 which include `throw/try/catch` exception handling and various computations expressing transformations between data types that rather loosely (but, in important respects, usefully) follow conventional `Applicative` and `Monadic` programming idioms. Thus, indexed generalizations of `Applicative` and `Monadic` structures, most importantly, generalizations of `State` and `Either`, are rather powerful “hammers” that can handle a surprising number of different types of “nails”.

In terms of related works this work contributes the following:

- Part II, essentially, extends the work of Wadler [110] by showing that `Monads` can also be used for proper error handling (and not just “hiding errors from the higher-level interpreter”), the observation which we formalized into `ConjoinedMonads` structure in section 7.5. Similarly to how Wadler’s `Monad` instances influenced the design of modern Haskell, instances discussed in part II also hint at new language design opportunities, which we discussed in chapters 9 and 10.
- Part III extends the work of McBride and Paterson [74] on `Applicatives` by showing other interesting structures that follow the same general form of expressions but allow for more sophisticated transformations, a structure which we formalized into the `ApplicativeLike` type class in section 12.6.2 and discussed the consequences of in chapter 14.
- Part IV, essentially, extends works on parser combinators, most notably the work of Leijen and Meijer [67], to “parsing” arbitrary data types.

Note, however, that most of those results are applicable outside of Haskell. For instance, `Monadic` observations are applicable to all languages that can explicitly override a linear composition (“semicolon”) operator. Thus, for instance, they can be applied to practically any language by using a pre-processor, the language in question can be kept unaware of any `Monadic` structures. Meanwhile, `ApplicativeLike` observations can be applied to all languages with function calls. The type system does not matter unless one wants to explicitly give types to those terms, which is not a strict requirement for applying the ideas discussed in this work.

Regarding specifically the Haskell language:

- `ConjoinedMonads` of part II and other similar Cartesian products discussed throughout the work require language extensions and/or modifications to type class inference mechanisms of Haskell to make them usable (that is, to make the same functions available for reuse between “computation” and “error handling” contexts) in actual programs, as discussed section 7.5,
- `ApplicativeLike` type class can already be encoded in Haskell, though not in a particularly beautiful way, as discussed in section 12.6.2,
- a language extension or a set of Template Haskell functions the base library providing LISP- and/or Scott-encoded representations for given data types would make results presented in part III much more pleasant to use.

As a general observation, note that the space of composable state transformers, of which our `ApplicativeLike` and `IxMonadic` structures are rather trivial examples, clearly contains a lot of structures that have their applicability to programming practice completely unexplored. Most of this work, essentially, explores a single such structure (and its categorical dual), but it is fairly clear that there are other structures with similar properties (e.g., remember sections 8.3 and 12.4). Moreover, part IV hints that there are likely to exist even more generic algebraic structures classifying those unexplored state transformers.

The author feels that the following future work directions on the topic would be of particular value:

- implementation of a practical “good-enough” (section 7.5) library for GHC Haskell, and, eventually, an implementation of a dialect of Haskell with a graded `MonadXApplicative` as a base type of computations,

- application of the ideas of part III to the **Alternative** type class to cover the multi-constructor case,<sup>1</sup>
- research into syntax and semantics of “marriages” between precise and imprecise exceptions in a single language, including, but not limited to, research into simpler semantic models for  $\lambda$ -calculus with Monads [24, 111],
- research into the question of whether multiplying more than two **Monads** and **Applicatives** with non-trivial interaction laws produces interesting structures.<sup>2</sup>

---

<sup>1</sup> It is not entirely clear if this is possible, 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.

<sup>2</sup> It is clear that one can have more than one index **e** conjoined to a single **a**, but such a construction doesn’t seem to make much sense in presence of graded **Monads**. However, that fact by itself does not exclude a possibility of existence of an interesting structure for which there are non-trivial interactions between different indexes **e**.

# Bibliography

- [1] Thorsten Altenkirch and Jonathan Grattage. “A functional quantum programming language”. In: (2005). arXiv: quant-ph/0409065 (cit. on p. 10).
- [2] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992. URL: <http://www.cambridge.org/9780521033114> (cit. on p. 45).
- [3] Kenichi Asai and Oleg Kiselyov. *Introduction to Programming with Shift and Reset*. Sept. 2011 (cit. on pp. 51, 59).
- [4] R. Bailey. “A Hope Tutorial”. In: *Byte Magazine* 10.8 (Aug. 1985), pp. 235–255. ISSN: 0360-5280 (cit. on p. 59).
- [5] Henk Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. Elsevier Science, 1984 (cit. on pp. 12, 14, 15).
- [6] Henk Barendregt. “Theoretical Pearls: Self-interpretation in lambda calculus”. In: *Journal of Functional Programming* 1.2 (Apr. 1991), pp. 229–233. ISSN: 0956-7968 (print), 1469-7653 (electronic). DOI: 10.1017/S0956796800020062. URL: <https://www.cambridge.org/core/product/D16FE1FAB38CD06AD831C845156E10D5> (cit. on pp. 14, 15).
- [7] 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. 8).
- [8] Nick Benton, John Hughes, and Eugenio Moggi. “Monads and Effects”. In: *Lecture Notes in Computer Science* 2395 (2002), pp. 42–122. ISSN: 0302-9743 (print), 1611-3349 (electronic) (cit. on p. 59).
- [9] Nick Benton and Andrew Kennedy. “Exceptional syntax”. In: *Journal of Functional Programming* 11.4 (July 2001), pp. 395–410. ISSN: 0956-7968 (print), 1469-7653 (electronic). DOI: 10.1017/S0956796801004099. URL: <https://www.cambridge.org/core/product/58206FB399EDC9F197A0D53BC46E4667> (cit. on p. 59).
- [10] Gert-Jan Bottu et al. “Quantified class constraints”. In: *ACM SIGPLAN Notices* 52.10 (Oct. 2017), pp. 148–161. ISSN: 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic). DOI: 10.1145/3156695.3122967 (cit. on p. 67).
- [11] Jan Bracker and Henrik Nilsson. “Supermonads and superapplicatives”. In: *Journal of Functional Programming* (2018). Under consideration, submitted 12 December 2017 (cit. on p. 103).
- [12] Edwin Brady. “Programming and reasoning with algebraic effects and dependent types”. In: *ACM SIGPLAN Notices* 48.9 (Sept. 2013), pp. 133–144. ISSN: 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic). DOI: 10.1145/2544174.2500581 (cit. on pp. 59, 60, 81).
- [13] Nicolaas G. de Bruijn. “Lambda-Calculus Notation with Nameless Dummies: a Tool for Automatic Formula Manipulation with Application to the Church-Rosser Theorem”. In: *Indag. Math.* 34.5 (1972), pp. 381–392 (cit. on p. 14).
- [14] R. M. Burstall, D. B. MacQueen, and D. T. Sannella. “HOPE: An experimental applicative language”. In: *ACM Symposium on Lisp and Functional Programming (LFP)*. 1980, pp. 136–143 (cit. on p. 59).
- [15] C++ FAQ Authors. *Sequence Points*. 2018. URL: <http://c-faq.com/expr/seqpoints.html> (cit. on pp. 38, 60).

- [16] 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](http://www.users.waitrose.com/~hindley/SomePapers_PDFs/2006CarHin,HistlamRp.pdf) (cit. on p. 8).
- [17] Robert Corbett et al. *GNU Bison: A General-purpose Parser Generator*. 2019. URL: <https://www.gnu.org/software/bison/> (cit. on p. 51).
- [18] Haskell Brooks Curry, J. Roger Hindley, and Jonathan P. Seldin. *Combinatory Logic: Volume II*. Amsterdam, Holland: North-Holland, 1972 (cit. on p. 47).
- [19] Olivier Danvy. “Functional unparsing”. In: *Journal of Functional Programming* 8.6 (Nov. 1998), pp. 621–625. ISSN: 0956-7968 (print), 1469-7653 (electronic). URL: <https://www.cambridge.org/core/product/789945109AD2AB168B504472E6B786A0> (cit. on p. 94).
- [20] Stephen Diehl. *What I Wish I Knew When Learning Haskell*. May 2016. URL: <http://dev.stephendiehl.com/hask/> (cit. on p. 2).
- [21] Carsten Dominik, Eric Schulte, Nicolas Goaziou, et al. *Org mode for Emacs*. 2018. URL: <https://orgmode.org/> (cit. on p. 2).
- [22] Conal Elliott. “Compiling to categories”. In: *Proc. ACM Program. Lang.* 1.ICFP (Sept. 2017). DOI: 10.1145/3110271. URL: <http://conal.net/papers/compiling-to-categories> (cit. on pp. 31, 80).
- [23] *Emacs Lisp Reference Manual: Catch and Throw*. 2018. URL: [https://www.gnu.org/software/emacs/manual/html\\_node/elisp/Catch-and-Throw.html](https://www.gnu.org/software/emacs/manual/html_node/elisp/Catch-and-Throw.html) (cit. on p. 60).
- [24] Andrzej Filinski. “Representing monads”. In: *Conference record of POPL ’94, 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: Portland, Oregon, January 17–21, 1994*. Ed. by ACM. New York, NY, USA: ACM Press, 1994, pp. 446–457. ISBN: 0-89791-636-0 (cit. on p. 119).
- [25] Bryan Ford. “Parsing expression grammars: a recognition-based syntactic foundation”. In: *ACM SIGPLAN Notices* 39.1 (Jan. 2004), pp. 111–122. ISSN: 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic) (cit. on p. 51).
- [26] 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 pp. 2, 9, 25, 26, 28, 29, 31–33, 39, 40, 83, 85, 96, 103, 106).
- [27] GHC Project Authors. *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. 103).
- [28] GHC Project Authors. *GHC: The Glasgow Haskell Compiler*. 2018. URL: <https://www.haskell.org/ghc/> (cit. on p. 3).
- [29] David K. Gifford and John M. Lucassen. “Integrating Functional and Imperative Programming”. In: *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*. LFP ’86. Cambridge, Massachusetts, USA: ACM, 1986, pp. 28–38. ISBN: 0-89791-200-4. DOI: 10.1145/319838.319848 (cit. on p. 60).
- [30] Andy Gill and Edward Kmett. *Hackage: The mtl package, version 2.2.1*. 2014. URL: <https://hackage.haskell.org/package/mtl-2.2.1> (cit. on pp. 43, 49).
- [31] 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 pp. 34, 35, 37, 50, 59, 84).
- [32] M. Gogolla et al. “Algebraic and operational semantics of specifications allowing exceptions and errors”. In: *Theoretical Computer Science* 34.3 (Dec. 1984), pp. 289–313. ISSN: 0304-3975 (print), 1879-2294 (electronic) (cit. on p. 59).
- [33] J. Goguen, J. Thatcher, and E. Wagner. “An initial algebra approach to the specification, correctness, and implementation of abstract data types”. In: *Current Trends in Programming Methodology* 4 (1978). (also IBM Report RC 6487, Oct. 1976) (cit. on p. 59).

- [34] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1983. ISBN: 0-201-11371-6 (cit. on pp. 59, 60).
- [35] Gabriel Gonzalez. *Haskell for all: errors-1.0: Simplified error handling*. 2012. URL: <http://www.haskellforall.com/2012/07/errors-10-simplified-error-handling.html> (cit. on p. 66).
- [36] Gabriel Gonzalez et al. *Hackage: The errors package, version 2.3.0*. 2018. URL: <https://hackage.haskell.org/package/errors-2.3.0> (cit. on pp. 66, 67).
- [37] John B. Goodenough. “Exception handling design issues”. In: *ACM SIGPLAN Notices* 10.7 (July 1975), pp. 41–45. ISSN: 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic). DOI: 10.1145/987305.987313 (cit. on p. 59).
- [38] John B. Goodenough. “Exception Handling: Issues and a Proposed Notation”. In: *Communications of the ACM* 18.12 (Dec. 1975), pp. 683–696. ISSN: 0001-0782 (print), 1557-7317 (electronic) (cit. on p. 59).
- [39] *Hackage: Haskell Central Package Archive*. 2018. URL: <https://hackage.haskell.org/> (cit. on pp. 9, 83).
- [40] Haskell Wiki Authors. *Typeclassopedia*. 2018. URL: <https://wiki.haskell.org/Typeclassopedia> (cit. on pp. 2, 27, 83).
- [41] 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. 84).
- [42] Charles Anthony Richard Hoare. “The Emperor’s Old Clothes”. In: *Communications of the ACM* 24.2 (1981). This is the 1980 ACM Turing Award Lecture, delivered at ACM’80, Nashville, Tennessee, October 27, 1980., pp. 75–83. ISSN: 0001-0782 (print), 1557-7317 (electronic). DOI: 10.1145/358549.358561 (cit. on p. 80).
- [43] 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. 31, 96, 103).
- [44] Jose Iborra. “Explicitly Typed Exceptions for Haskell”. In: *PADL’10*. Jan. 2010 (cit. on pp. 59, 76, 80).
- [45] Pepe Iborra. *Hackage: The control-monad-exception package, version 0.11.2*. 2015. URL: <http://hackage.haskell.org/package/control-monad-exception-0.11.2> (cit. on pp. 57, 76).
- [46] IEEE. *IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) Base Definitions, Issue 6*. Revision of IEEE Std 1003.1-1996 and IEEE Std 1003.2-1992) Open Group Technical Standard Base Specifications, Issue 6. New York, NY, USA: IEEE, 2001. ISBN: 1-85912-247-7 (UK), 1-931624-07-0 (US), 0-7381-3047-8 (print), 0-7381-3010-9 (PDF), 0-7381-3129-6 (CD-ROM) (cit. on p. 60).
- [47] S. C. Johnson and B. W. Kernighan. *The Programming Language B*. Technical report 8. Murray Hill, NJ, USA: Bell Laboratories, 1973 (cit. on p. 8).
- [48] Stephen C. Johnson et al. *Yacc: A General-purpose Parser Generator*. 2017. URL: <http://bxsu/OpenBSD/usr.bin/yacc/> (cit. on p. 51).
- [49] M. P. Jones. “Functional Programming with Overloading and Higher-Order Polymorphism”. In: *Lecture Notes in Computer Science* 925 (1995), 97–?? ISSN: 0302-9743 (print), 1611-3349 (electronic) (cit. on p. 34).
- [50] Ohad Kammar, Sam Lindley, and Nicolas Oury. “Handlers in action”. In: *ACM SIGPLAN Notices* 48.9 (Sept. 2013), pp. 145–158. ISSN: 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic). DOI: 10.1145/2544174.2500590 (cit. on p. 59).
- [51] 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 pp. 51, 83).

- [52] Shin-ya Katsumata. “Parametric effect monads and semantics of effect systems”. In: *ACM SIGPLAN Notices* 49.1 (Jan. 2014). POPL ’14 conference proceedings., pp. 633–645. ISSN: 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic). DOI: 10.1145/2578855.2535846 (cit. on pp. 59, 79, 80).
- [53] 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. 8).
- [54] Oleg Kiselyov. *An argument against call/cc*. 2012. URL: <http://okmij.org/ftp/continuations/against-callcc.html> (cit. on pp. 51, 59, 61).
- [55] Oleg Kiselyov and Hiromi Ishii. “Freer monads, more extensible effects”. In: *ACM SIGPLAN Notices* 50.12 (Dec. 2015), pp. 94–105. ISSN: 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic). DOI: 10.1145/2887747.2804319 (cit. on p. 59).
- [56] Oleg Kiselyov, Amr Sabry, and Cameron Swords. “Extensible effects: an alternative to monad transformers”. In: *ACM SIGPLAN Notices* 48.12 (Dec. 2013). Haskell ’14 conference proceedings., pp. 59–70. ISSN: 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic). DOI: 10.1145/2578854.2503791 (cit. on pp. 59, 60).
- [57] Edward Kmett. *Free Monads for Less (Part 3 of 3): Yielding IO*. 2011. URL: <http://comonad.com/reader/2011/free-monads-for-less-3/> (cit. on p. 39).
- [58] Edward Kmett. *Lenses, Folds and Traversals*. 2013. URL: <http://lens.github.io/> (cit. on pp. 23, 86, 106).
- [59] Edward Kmett. *Hackage: The exceptions package, version 0.8.3*. 2015. URL: <https://hackage.haskell.org/package/exceptions-0.8.3> (cit. on p. 43).
- [60] Edward A. Kmett et al. *Hackage: The indexed package, version 0.1.3*. 2016. URL: <https://hackage.haskell.org/package/indexed-0.1.3> (cit. on p. 55).
- [61] Edward Kmett et al. *Hackage: The lens package, version 4.17*. 2018. URL: <https://hackage.haskell.org/package/lens-4.17> (cit. on pp. 23, 86, 106).
- [62] Andrew Koenig and Bjarne Stroustrup. “Exception Handling for C++ (revised)”. In: *USENIX C++ conference proceedings: C++ Conference, San Francisco, California, April 9–11, 1990*. Ed. by USENIX. San Francisco, CA: USENIX, 1990, pp. 149–176 (cit. on p. 59).
- [63] Andrei Nikolaevich Kolmogorov. “On the principle of the excluded middle”. English. In: *From Frege to Gödel (1971)*. Ed. by van Heijenoort, pp. 414–437 (cit. on p. 46).
- [64] 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 pp. 23, 86, 106).
- [65] Keunwoo Lee. *CSE341: Programming Languages: Scheme: Continuations*. 2004. URL: <http://courses.cs.washington.edu/courses/cse341/04wi/lectures/15-scheme-continuations.html> (cit. on p. 73).
- [66] 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 pp. 23, 51, 83).
- [67] Daan Leijen and Erik Meijer. *Parsec: Direct Style Monadic Parser Combinators for the Real World*. Tech. rep. July 2001. URL: <https://www.microsoft.com/en-us/research/publication/parsec-direct-style-monadic-parser-combinators-for-the-real-world/> (cit. on pp. 51, 55, 59, 118).
- [68] Sheng Liang, Paul Hudak, and Mark Jones. “Monad transformers and modular interpreters”. In: *Conference record of POPL ’95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: San Francisco, California, January 22–25, 1995*. Ed. by ACM. ACM order number: 549950. New York, NY, USA: ACM Press, 1995, pp. 333–343. ISBN: 0-89791-692-1 (cit. on pp. 34, 59).

- [69] John M. Lucassen. “Types and effects, towards the integration of functional and imperative programming”. PhD thesis. MIT Laboratory for Computer Science, Aug. 1987. URL: <http://software.imdea.org/~aleks/thesis/CMU-CS-04-151.pdf> (cit. on p. 60).
- [70] John M. Lucassen and David K. Gifford. “Polymorphic effect systems”. In: *Principles of Programming Languages (POPL)*. Jan. 1988, pp. 47–57. URL: <http://pag.lcs.mit.edu/reading-group/lucassen88effects.pdf> (cit. on p. 60).
- [71] Geoffrey Mainland. “Why It’s Nice to Be Quoted: Quasiquoting for Haskell”. In: *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*. Haskell ’07. Freiburg, Germany: ACM, 2007, pp. 73–82. ISBN: 978-1-59593-674-5. DOI: 10.1145/1291201.1291211 (cit. on p. 80).
- [72] Simon Marlow. “An Extensible Dynamically-typed Hierarchy of Exceptions”. In: *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*. Haskell ’06. Portland, Oregon, USA: ACM, 2006, pp. 96–106. ISBN: 1-59593-489-8. DOI: 10.1145/1159842.1159854 (cit. on p. 38).
- [73] matt76k. *Hackage: ponder package, version 0.0.1*. 2014. URL: <https://hackage.haskell.org/package/ponder-0.0.1> (cit. on p. 53).
- [74] 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. 10, 26, 83, 103, 118).
- [75] MirageOS Project Authors. *MirageOS: A programming framework for building type-safe, modular systems*. 2019. URL: <https://mirage.io/> (cit. on p. 9).
- [76] Neil Mitchell. *Continuations and Exceptions*. 2014. URL: <http://neilmitchell.blogspot.fr/2014/08/continuations-and-exceptions.html> (cit. on p. 73).
- [77] Neil Mitchell. *Shake*. 2018. URL: <https://github.com/ndmitchell/shake> (cit. on p. 73).
- [78] Neil Mitchell and Colin Runciman. *Uniplate*. 2007. URL: <http://community.haskell.org/~ndm/uniplate/> (cit. on pp. 23, 86, 106).
- [79] Eugenio Moggi. “Computational  $\lambda$ -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 pp. 10, 59, 60, 103).
- [80] 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 pp. 10, 59, 103).
- [81] Aleksandar Nanevski. “Functional Programming with Names and Necessity”. PhD thesis. Carnegie Mellon University, 2004. URL: <http://software.imdea.org/~aleks/thesis/CMU-CS-04-151.pdf> (cit. on pp. 59, 61).
- [82] Aleksandar Nanevski. “A Modal Calculus for Exception Handling”. In: *Intuitionistic Modal Logic and Applications Workshop (IMLA)*. A Logic in Computer Science Conference. Chicago, Illinois, USA, 2005. URL: <http://software.imdea.org/~aleks/papers/effects/imla05.pdf> (cit. on p. 59).
- [83] 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. 10).
- [84] Matthew Naylor, Colin Runciman, and Jason Reich. *The Reduceron*. 2010. URL: <https://www.cs.york.ac.uk/fp/reduceron/> (cit. on p. 10).
- [85] 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. 84).
- [86] NumPy Project Authors. *NumPy: Scientific Computing with Python*. 2019. URL: <https://numpy.org/> (cit. on p. 9).
- [87] Russell O’Connor. *Hackage: The multiplate package, version 0.0.3*. 2015. URL: <https://hackage.haskell.org/package/multiplate-0.0.3> (cit. on pp. 23, 86, 106).



- [88] 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 pp. 51, 83).
- [89] 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. 84).
- [90] Simon Peyton Jones et al. “A Semantics for Imprecise Exceptions”. In: *ACM SIGPLAN Notices* 34.5 (May 1999), pp. 25–36. ISSN: 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic) (cit. on pp. 38, 60, 76, 79).
- [91] Kent M. Pitman. “Condition Handling in the Lisp Language Family”. In: *Lecture Notes in Computer Science* 2022 (2001), 39–?? ISSN: 0302-9743 (print), 1611-3349 (electronic). URL: <http://link.springer-ny.com/link/service/series/0558/bibs/2022/20220039.htm>; <http://link.springer-ny.com/link/service/series/0558/papers/2022/20220039.pdf> (cit. on pp. 59, 60).
- [92] Gordon D. Plotkin and Matija Pretnar. “Handlers of Algebraic Effects”. In: *Programming Languages and Systems: 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22–29, 2009. Proceedings*. Ed. by Giuseppe Castagna. Vol. 5502. Lecture Notes in Computer Science. New York, NY, USA: Springer-Verlag Inc., 2009, pp. 80–94. ISBN: 3-642-00589-6 (print), 3-642-00590-X (e-book). DOI: 10.1007/978-3-642-00590-9\_7. URL: <http://www.springerlink.com/content/978-3-642-00590-9> (cit. on p. 59).
- [93] Qubes OS Project Authors. *Qubes OS: A reasonably secure operating system*. 2019. URL: <https://www.qubes-os.org/> (cit. on p. 8).
- [94] John C. Reynolds. “The Discoveries of Continuations”. In: *Lisp and Symbolic Computation* 6.3/4 (Nov. 1993), pp. 233–248. ISSN: 0892-4635 (print), 1573-0557 (electronic) (cit. on p. 45).
- [95] Morten Rhiger. “Type-safe pattern combinators”. In: *Journal of Functional Programming* 19.2 (Mar. 2009), pp. 145–156. ISSN: 0956-7968 (print), 1469-7653 (electronic). DOI: 10.1017/S0956796808007089. URL: <https://www.cambridge.org/core/product/1E3D0890F2ED1B70F80722A732756910> (cit. on pp. 94, 106).
- [96] 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. 8).
- [97] Amr Sabry. *Topics in Programming Languages: Reversible and Quantum Computing*. 2011. URL: <https://www.cs.indiana.edu/~sabry/teaching/b629/s11/> (cit. on p. 10).
- [98] 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).
- [99] Sergei Soloviev and Jan Malakhovski. “Automorphisms of Types and Their Applications”. In: *Journal of Mathematical Sciences* 240 (5 Aug. 2019), pp. 692–706 (cit. on p. 84).
- [100] Andy Sonnenburg. *Hackage: The catch-fd package, version 0.2.0.2*. 2012. URL: <http://hackage.haskell.org/package/catch-fd-0.2.0.2> (cit. on p. 57).
- [101] Michael Sperber et al. *Revised<sup>6</sup> Report on the Algorithmic Language Scheme*. 1st. New York, NY, USA: Cambridge University Press, 2010. ISBN: 0521193990, 9780521193993 (cit. on pp. 51, 59).
- [102] *Standard ML of New Jersey: The CONT signature*. URL: <http://www.smlnj.org//doc/SMLofNJ/pages/cont.html#SIG:CONT.cont:TY> (cit. on p. 51).
- [103] J. Steensgaard-Madsen. “Type Representation of Objects by Functions”. In: *ACM Transactions on Programming Languages and Systems* 11.1 (Jan. 1989), pp. 67–89. ISSN: 0164-0925 (print), 1558-4593 (electronic) (cit. on p. 47).
- [104] Bjarne Stroustrup. *The Design and Evolution of C++*. Reading, MA, USA: Addison-Wesley, 1994, pp. x + 461. ISBN: 0-201-54330-3 (cit. on pp. 59, 61).

- [105] Wouter Swierstra. “Data Types à La Carte”. In: *Journal of Functional Programming* 18.4 (July 2008), pp. 423–436. ISSN: 0956-7968. DOI: 10.1017/S0956796808006758 (cit. on p. 59).
- [106] The Go Programming Language Project Authors. *The Go Programming Language: Standard Library Packages*. 2019. URL: <https://golang.org/pkg/> (cit. on p. 9).
- [107] The Rust Programming Language Project Authors. *The Rust Standard Library*. 2019. URL: <https://doc.rust-lang.org/std/> (cit. on p. 9).
- [108] *TIOBE Index*. 2019. URL: <https://www.tiobe.com/tiobe-index/> (cit. on p. 9).
- [109] Philip Wadler. “Deforestation: transforming programs to eliminate trees”. In: *Theoretical Computer Science* 73.2 (June 1990), pp. 231–248. ISSN: 0304-3975 (print), 1879-2294 (electronic) (cit. on p. 49).
- [110] 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 pp. 10, 59, 60, 103, 118).
- [111] 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 pp. 1, 60, 80, 119).
- [112] WikiBooks Authors. *Scheme Programming: Continuations*. 2017. URL: [https://en.wikibooks.org/w/index.php?title=Scheme\\_Programming/Continuations&oldid=3168913](https://en.wikibooks.org/w/index.php?title=Scheme_Programming/Continuations&oldid=3168913) (cit. on p. 73).