Unified Notions of Generalised Monads and Applicative Functors

Jan Bracker

Thesis submitted to the University of Nottingham for the degree of Doctor of Philosophy

September 2018

Abstract

Monads and applicative functors are staple design patterns to handle effects in pure functional programming, especially in Haskell with its built-in syntactic support. Over the last decade, however, practical needs and theoretical research have given rise to generalisations of monads and applicative functors. Examples are graded, indexed and constrained monads. The problem with these generalisations is that no unified representation of standard and generalised monads or applicatives exists in theory or practice. As a result, in Haskell, each generalisation has its own representation and library of functions. Hence, interoperability among the different notions is hampered and code is duplicated.

To solve the above issues, I first survey the three most wide-spread generalisations of monads and applicatives: their graded, indexed and constrained variations. I then examine two approaches to give them a unified representation in Haskell: polymonads and supermonads. Both approaches are embodied in plugins for the Haskell compiler GHC that address most of the identified concerns. Finally, I examine category theory and propose unifying categorical models that encompass the three discussed generalisations together with the standard notions of monad and applicative.

Acknowledgements

First, I need to thank the University of Nottingham, especially the School of Computer Science, for their scholarship program that enabled me to pursue my PhD. Without their academic and financial support, non of the work presented in this thesis would have happened.

Next I need to thank my supervisor Henrik for all of his advice and support during my PhD. He always had an open ear for my troubles and time to discuss my ideas at length.

I also have to thank the Functional Programming Lab in general. It always provided a nourishing and friendly environment to work and exchange ideas.

Finally, I want to thank my good friends and my family for their moral support and advice when I was unsure where I was heading.

Contents

1 Introduction **1**
- 1.1 Problem statement . . . 2
- 1.2 Goals and contributions . . . 4
- 1.3 Thesis overview . . . 6
- 1.4 Previous work . . . 6

2 Preliminaries **8**
- 2.1 Nomenclature . . . 8
- 2.2 Haskell language extensions . . . 9
- 2.3 GHC type checker plugins . . . 10
- 2.4 Proofs . . . 10

3 Generalised monads and applicatives **12**
- 3.1 Monadic notions . . . 12
  - 3.1.1 Standard monads . . . 12
  - 3.1.2 Indexed monads . . . 14
  - 3.1.3 Graded monads . . . 16
  - 3.1.4 Constrained monads . . . 18
- 3.2 Applicative notions . . . 19
  - 3.2.1 Standard applicatives . . . 21
  - 3.2.2 Indexed applicatives . . . 21
  - 3.2.3 Graded applicatives . . . 21
  - 3.2.4 Constrained applicatives . . . 22

4 Polymonads **23**
- 4.1 Definition . . . 23
- 4.2 Representation in Haskell . . . 25
- 4.3 Examples of polymonads . . . 26
  - 4.3.1 Standard monads . . . 27
  - 4.3.2 Indexed monads . . . 28
  - 4.3.3 Graded monads . . . 29
- 4.4 Properties of polymonads . . . 31
  - 4.4.1 Uniqueness of bind operations . . . 31
  - 4.4.2 Principality of polymonads . . . 32
  - 4.4.3 Union of polymonads . . . 33

4.5 Polymonad plugin for GHC . . . 35
4.5.1 Structure . . . 35
4.5.2 Selection of bind operations . . . 36
4.5.3 Using the plugin . . . 38
4.6 Limitations and conclusion . . . 38

5 Supermonads and superapplicatives **41**
5.1 Representation in Haskell . . . 41
5.1.1 Separate type classes for the bind, ap, and return operation 42
5.1.2 Insufficient type inference . . . 43
5.1.3 Enhancing type inference and constraint solving. . . . 44
5.1.4 The supermonad and superapplicative type classes . . . 45
5.1.5 Adding constraints . . . 47
5.1.6 Practical implications of associated constraints . . . 48
5.1.7 Why `Applicative` is not a superclass of `Bind` . . . 50
5.2 Examples of supermonads and superapplicatives . . . 51
5.2.1 `Maybe` and the state transformer: a standard monad . . . 52
5.2.2 Fixed-length vectors: a graded monad . . . 53
5.2.3 Session types: an indexed monad . . . 55
5.2.4 The `Set` constrained monad . . . 56
5.2.5 Tracking resources: an application of graded applicatives and indexed monads . . . 57
5.3 Implementation and use of the GHC plugin . . . 60
5.3.1 Implementation and structure . . . 60
5.3.2 Using the plugin . . . 65
5.4 Case studies . . . 66
5.4.1 Teaching compiler . . . 66
5.4.2 Chat server and client . . . 68
5.5 Limitations and conclusions . . . 70

6 Category theoretical context **71**
6.1 Prerequisite categorical notions . . . 72
6.1.1 Discrete and codiscrete collections of morphisms . . . 72
6.1.2 Natural isomorphisms . . . 72
6.1.3 Monoidal categories . . . 73
6.1.4 Lax monoidal functors . . . 74
6.1.5 Strict 2-categories . . . 76
6.1.6 Lax 2-functors . . . 78
6.1.7 Monads and Kleisli triples . . . 80
6.1.8 Relative monads . . . 81
6.1.9 Categorical representation of constraints . . . 83
6.2 Monadic notions . . . 84
6.2.1 Standard monads . . . 86
6.2.2 Graded monads . . . 86
6.2.3 Indexed monads . . . 87

6.2.4 Unified categorical representation of parameterised monads . . . 89
6.2.5 Constrained monads . . . 89
6.3 Applicative notions . . . 92
6.3.1 Standard applicatives . . . 93
6.3.2 Graded applicatives . . . 93
6.3.3 Constrained applicatives . . . 95
6.3.4 Indexed applicatives . . . 95
6.4 Conclusion of categorical models . . . 97

7 Related work **100**
7.1 Atkey's parameterised notions of computation . . . 100
7.2 Kmett's approach to indexed monads in Haskell . . . 101
7.3 Alternate generalisations of functors, monads and applicatives . 103
7.3.1 McBride's Kleisli arrows of outrageous fortune . . . 103
7.3.2 Functor parameterised data structures . . . 104
7.3.3 Different source categories for functors . . . 105
7.3.4 Higher-order functors . . . 105
7.3.5 Another approach to constrained functors and monads . 106
7.4 Generalisations of arrows . . . 107
7.4.1 Adding indices or constraints . . . 107
7.4.2 Abstracting the involved notion of product and function 107
7.4.3 Problems when generalising arrows . . . 108
7.5 Polymonads . . . 108
7.6 Unifying applicatives, monads and arrows . . . 109
7.7 Alternatives to GHC type checker plugins . . . 109

8 Conclusions **111**
8.1 Polymonads vs supermonads . . . 111
8.2 Categorical models . . . 112
8.3 Future work . . . 113

Bibliography **115**

A How to read Agda **122**
A.1 General remarks . . . 122
A.2 Modules . . . 123
A.3 Imports . . . 124
A.4 Functions and operators . . . 125
A.5 Data types and records . . . 126
A.6 Proofs of equality . . . 127
A.7 Universal and existential quantification . . . 128

B Overview of Agda formalisations and proofs **130**
B.1 Formalisations 130
B.1.1 Formalisations of Haskell structures 130
B.1.2 Formalisations of polymonads 130
B.1.3 Formalisations of category theory 131
B.2 Proofs 132
B.2.1 Proofs involving Haskell structures 132
B.2.2 Proofs involving polymonads 133
B.2.3 Proofs within category theory 134

Chapter 1

Introduction

To support effectful programming in a functional setting without compromising purity, the Haskell community has adopted a range of computational notions: *monads* [Mog88; Wad92], *arrows* [Hug00], and *applicative functors* [MP08]. Due to the central role monads have in Haskell, the language offers the `do`-notation[1] as special syntactic support to work with monads. Because of the growing importance of arrows and applicative functors, the `do`-notation has been extended to support these notions as well [Pat01; Mar+16].

Monads and applicative functors are extremely versatile and allow the expression of a wide variety of side-effects. However, sometimes the types of monads and applicatives lack expressiveness and flexibility. The computation of a monad `M` which delivers a result of type `a` has the type `M a` in Haskell. The presence of the type constructor `M` signals the presence of side-effects, but it cannot give us more detailed guarantees about the nature of those side-effects. Haskell's `IO` monad is an example of this lack of specificity. Evidence for interaction with the outside world is given by the presence of the `IO` type constructor, but nothing more is known about the nature of this interaction. A computation that involves the `IO` monad may involve any number of effects, e.g., file system access, network communication, mutable memory, concurrency or random number generation.

Several generalisations of monads have been proposed to allow for more fine-grained type-level guarantees. The most popular generalisations are graded [Wad98; Kat14] and indexed monads [Wad94; Atk09]. These generalisations add additional indices to the type constructor of the monadic computation and allow these indices to change over the course of a computation. Both generalised monads have proven useful in several different contexts, such as, session-types [PT08], effect systems [OP14; McB11], and information flow control [DP11].

Another shortcoming of standard monads in Haskell is that they do not allow the introduction of constraints on the result types of a computation. An example of this problem is the `Set` data type that represents finite sets in Haskell. It

[1] *Haskell 2010 Language Report* - `https://www.haskell.org/onlinereport/haskell2010`

would form a monad similar to lists were it not for the fact that its implementation requires an ordering on the elements. Although several workarounds[2,3] for *constrained* monads have been proposed to rectify this problem [Hug99; ACU10; Scu+13], it remains unsatisfying not to be able to encode these kinds of monads in a direct and "obvious" fashion, especially because these workarounds are not applicable in every situation.

Unfortunately, all of the generalised notions above, i.e., graded, indexed, and constrained monads, lead to new problems.

1.1 Problem statement

At the time of writing, the graded, indexed, and constrained notions are usually provided as a library that contains a new type class or a collection of type classes that encode each notion. Hence, a programmer wanting to use such a generalised notion works with the `Monad` type class from the standard `Prelude` and then imports the encoding of the needed generalised notion in addition to that.

As a result, the following two major problems prevail when using generalised monads or applicatives:

- *Hampered code reuse:* For each type class a complete reimplementation of all monad related standard library functions is required.
- *Necessity of manual disambiguation:* Although the use of several monadic notions in conjunction within the same module is possible, the resulting code may become cluttered with disambiguations quickly.

The following paragraphs illustrate both points in more detail.

For example, if a programmer wants to work with indexed monads[4], they could import the encoding of indexed monads from the module `I`:

```
module I where
  class IndexedMonad m where
    (>>=) :: m i j a -> (a -> m j k b) -> m i k b
    return :: a -> m i i a
```

At a glance, this seems reasonable and natural, but the mentioned problems emerge when working with the imported notion. The programmer will notice that the standard library functions for `Monad` do not work with the generalised `IndexedMonad`. Hence, the programmer is forced to use the reimplementations of these functions from the module `I` (if they exist). This is tedious, because it requires the programmer to either fully qualify overlapping names from the module `I` and the prelude or to consciously use differently named functions whenever

[2] *Restricted Monads (8. Feb. 2006)* - `http://okmij.org/ftp/Haskell/types.html#restricted-datatypes`

[3] Hackage: *rmonad* - `http://hackage.haskell.org/package/rmonad`

[4] The idea and motivation behind `IndexedMonad` is explained in Section 3.1.2.

working with the generalised notion. However, the implementation and idea behind these standard library functions is literally the same for either notion. The function `liftM` and its indexed counterpart `liftIM` demonstrate this:

```
liftM  :: (Monad m)
       => (a -> b) -> m a -> m b
liftM  f ma = ma Prelude.>>= \a -> Prelude.return (f a)

liftIM :: (IndexedMonad m)
       => (a -> b) -> m i j a -> m i j b
liftIM f ma = ma I.>>= \a -> I.return (f a)
```

Furthermore, once the programmer attempts to work with the do-notation the same problems arise. The do-notation may be overloaded to work with different monadic notions (see `RebindableSyntax` in Section 2.2), but when overloaded it uses any >>= and `return` operation in scope. Therefore, importing several monadic notions in the same module requires a manual disambiguation of which notion is used within a specific do-block. A rewrite of `liftM` and `liftIM` using do-notation demonstrates this:

```
liftM :: (Monad m)
      => (a -> b) -> m a -> m b
liftM f ma = do
  a <- ma
  Prelude.return (f a)
    where (>>=) = (Prelude.>>=)

liftIM :: (IndexedMonad m)
       => (a -> b) -> m i j a -> m i j b
liftIM f ma = do
  a <- ma
  I.return (f a)
    where (>>=) = (I.>>=)
```

Now every do-block needs to be annotated with the bind (and sequence) operation to be used so that the compiler is able to type check and desugar appropriately.

Of course, these problems only occur if more than one notion is in use within the same module. However, standard monads are so commonplace in Haskell that banning them from a module just to allow the comfortable use of a monadic generalisation may be cumbersome in itself. There are many examples where a small computation involving lists or `Maybe` is useful, but would be impossible or inconvenient when only importing a generalised monad in favour of a standard monad.

One possible solution to these problems is to use only the most general monadic notion and express every other monadic notion in terms of that more general

notion. However, this solution has a drawback. The type constructors of different generalisations have different arities. For example, the `IndexedMonad` from above requires a type constructor with three arguments, whereas standard monads only require one argument to their type constructor. Though it is possible to express a standard monad in terms of an indexed monad by using unit as a placeholder in the indices, this is impractical because it requires the user to rewrite all of their type signatures to add "dummy" indices.

I chose graded, indexed, and constrained monads as the monadic generalisations that I want to support, because they appear to be the most commonly used and wide-spread notions. My survey of Hackage[5] and the scientific literature pointed to these three notions. Of course, there are several other possible generalisation of monads (and applicative functors), but the aforementioned three appear to be the most popular. In addition to my search on Hackage and in literature, I also asked the Haskell mailing list[6] and the Haskell subreddit[7] if there are other generalisations and use cases of generalised monads. Unfortunately, I did not receive a response to my question. The lack of response may be seen as evidence that I already found the relevant generalised notions. I discuss other generalisations in Section 7.3 of the related work.

1.2 Goals and contributions

A single representation that is able to encompass standard, graded, indexed, and constrained notions provides a solution to the above problems. Therefore, the goals of my work are:

1. To find a practical unified representation within Haskell that is capable of expressing all of the different monadic and applicative notions from Section 3.

2. To find the categorical design patterns underlying that unified notion.

To achieve these goals, I explore two approaches. Both approaches look similar when realised in Haskell, but their implementation and background is different.

First, I examine *polymonads* as introduced by Guts et al. [Gut+12] and Hicks et al. [Hic+14]. I take a close look at the composability of polymonads and their applicability in the context of Haskell. Ultimately, I present a language extension which allows the usage of polymonads in Haskell. Unfortunately, polymonads only support a subset of the generalised monadic notions I aim to support. It is

[5] *Hackage :: [Package]* - `http://hackage.haskell.org/`

[6] *[Haskell-cafe] Applications using generalized monads?* - `https://mail.haskell.org/pipermail/haskell-cafe/2016-May/123842.html`

[7] *Software and applications using generalized monads?* - `https://www.reddit.com/r/haskellquestions/comments/4fs80x/software_and_applications_using_generalized_monads/`

unclear if and how polymonads support applicative generalisations. Therefore, they are only of limited utility in solving the aforementioned problems.

Hence, I introduce supermonads. The development of supermonads is driven by the goal to solve all problems listed above. I develop a language extension that is able to represent all of the aforementioned generalisations and enables their seamless interaction within Haskell. I carry out two case studies to show the robustness and scalability of the supermonad approach. Furthermore, the approach also works for applicative functors and their generalisations. Thus, I also introduce and support the generalised versions of applicative functors: superapplicatives.

Finally, I explore the category-theoretic connection between the different generalised notions. This delivers a first step towards a unified categorical representation of generalised monads and applicative functors, respectively. The categorical models serve as a design pattern to which notions are allowed in the context of supermonads and superapplicatives.

In summary, the main contributions of my work are:

- A representation and implementation of polymonads in Haskell through a plugin for the Glasgow Haskell Compiler[8] (GHC).
- A formalisation of polymonads in Agda, together with proofs concerning the union of polymonads and the uniqueness of their bind operations.
- The development of generalised applicative functors based on and similar to graded, indexed, and constrained monads.
- The development of supermonads and superapplicatives as a representation for the above monadic and applicative generalisations together with a GHC plugin to support them in Haskell.
- Two case studies to demonstrate that supermonads work seamlessly in practice and to provide a stress test of their implementation as a GHC plugin.
- A survey and exploration of categorical models for graded, indexed, and constrained monads and applicatives that is independent of polymonads or supermonads.
- A proposition of unified categorical models that capture all of the generalised monads and applicatives mentioned above.
- An Agda formalisation and verification of the connections between the different categorical models and generalised notions in Haskell.

[8] *The Glasgow Haskell Compiler* - `https://www.haskell.org/ghc`

1.3 Thesis overview

Chapter 2 gives a short overview of the used nomenclature followed by an introduction of the different Haskell language extensions and compiler infrastructure that is used throughout this thesis. These preliminaries close with an explanation of the formalisation and proof techniques used for all of the proofs in my thesis.

Chapter 3 introduces all of the different monadic and applicative notions that I aim to support with the developed unifying notions. Each monadic generalisation is presented together with a motivating example to demonstrate their utility and use cases. The applicative generalisations are derived from their monadic counterparts in Section 3.2. All generalisations are introduced using their representation in Haskell.

Polymonads are presented in Chapter 4. They provide a first approach to a unified notion for monadic generalisations in Haskell. This chapter formally introduces polymonads (Section 4.1) and how I encode them in Haskell (Section 4.2). I then provide several examples of polymonads and how they are instantiated within the Haskell representation in Section 4.3. Afterwards, several properties such as uniqueness, principality, and union of polymonads are discussed due to their importance in the context of a Haskell implementation (Section 4.4). Finally, I explain how the polymonad implementation works (Section 4.5) and which limitations the polymonad approach has (Section 4.6).

Supermonads and superapplicatives are presented in Chapter 5. First, I discuss how both notions are represented in Haskell (Section 5.1). Based on that representation, examples of supermonads are given in Section 5.2. Afterwards, I explain how the implementation works in Section 5.3. Finally, I discuss two case studies to provide evidence for the practicality and robustness of the approach (Section 5.4) and then close with a discussion of the limitations in Section 5.5.

Chapter 6 presents a categorical background for the different monadic and applicative notions. First, I present and explain prerequisite category theory required to understand the categorical formalisations (Section 6.1). Afterwards, categorical models for all of the monadic and applicative notions (Section 6.2 and 6.3) are explained. Section 6.4 concludes by presenting a hierarchy of generalised monads and applicative functors, respectively.

Finally, I discuss related work in Section 7 and draw conclusions in Section 8.

1.4 Previous work

Most of the work presented in this thesis has already been published in previous work by Henrik Nilsson and me:

- The work on polymonads presented in Section 4 is adopted from the article “Polymonad Programming in Haskell” [BN15].
- The work on supermonads presented in Section 5 is adopted from the article “Supermonads: One Notion to Bind Them All” [BN16].

- The work on superapplicatives in Section 5 and on categorical models in Section 6 is adopted from the article "Supermonads and superapplicatives"[9] [BN18] which, at the time of writing, is under consideration for publication in the Journal of Functional Programming.

I was the lead author in all of the above work. The implementation, formalisation, and proofs were done solely by me.

[9] *Supermonads and superapplicatives* - http://jbracker.de/publications/2017-BrackerNilsson-SupermonadsAndSuperapplicatives-UnderConsideration.pdf

Chapter 2

Preliminaries

To understand this thesis, the reader is expected to have a working knowledge of standard Haskell 2010[1] and basic category theory (categories, functors, and natural transformations). All other concepts are explained in the respective chapters.

This chapter gives a preparatory overview of the used terminology and nomenclature (Section 2.1) and the Haskell language and compiler extensions (Section 2.2 and 2.3) that are required. In addition, background information about the proofs in my thesis is provided in Section 2.4.

2.1 Nomenclature

Considering that the terminology and naming of different monadic generalisations is somewhat varying in my previous work [BN15; BN16] and the literature more broadly, I want to establish what I consider to be consistent terms, which I will use throughout my thesis.

Used name	Other names
standard monad	classical monad
graded monad	effect monad
indexed monad	parameterised or Hoare monad
constrained monad	restricted monad

"Standard monad" refers to the well established notion of a monad in Haskell or category theory. "Graded", "indexed", and "constrained monads" are generalisations of standard monads. I introduce and discuss all of these different monadic notions in Section 3.1.

I use the term *parameterised monad* collectively for standard, graded, and indexed monads. The terms "monad" and "applicative functor" can be exchanged in all of the above terminology to name the corresponding applicative variation.

[1] *Haskell 2010 Language Report* - `https://www.haskell.org/onlinereport/haskell2010`

The term "applicative functor" is unwieldy. Therefore, I use the phrase "applicative" or "applicatives" synonymously with it from this point onward.

2.2 Haskell language extensions

The implementations and the Haskell code presented in this thesis require several language extensions and a plugin mechanism that are only available in GHC[2]. Therefore, the use of GHC as a Haskell compiler is required for all of the presented work in the following chapters.

The following list contains the most important language extensions that are required to understand and run any of the Haskell code in this thesis. Further information on each language extension is provided in the referenced sections of the GHC User's Guide[3] (GHCUG).

RebindableSyntax: Allows syntactic sugar, such as the do-notation, to be translated in terms of user-specified functions, i.e., custom bind and sequence operations (GHCUG 9.3.15). This extension disables the implicit import of the standard prelude. Therefore, a custom prelude has to be provided when using rebindable syntax.

MultiParamTypeClasses: Allows defining type classes with more than one argument (GHCUG 9.8.1.1).

OverlappingInstances: Allows to declare multiple instances that overlap with each other, i.e., that several instances can possibly match the same constraint (GHCUG 9.8.3.6).

EmptyDataDecls: Allows the definition of data types without constructors (GHCUG 9.4.1).

TypeFamilies: Facilitates type-level programming by effectively providing type-level functions. This extension is required to allow the use of associated type synonyms [CKJ05; Sch+08] to specify type-level information that is specific to a type-class instance (GHCUG 9.9).

DataKinds: Allows data types to be used on the type/kind level instead of just the value/type level (GHCUG 9.10). Once enabled any data type may be used in a type/kind context. It may be necessary to escape the constructors and types used at the type/kind level by prefixing them with an apostrophe when ambiguity occurs.

PolyKinds: Generalises kind inference to support polymorphic kinds in the form of variables at the kind level (GHCUG 9.11).

[2] *The Glasgow Haskell Compiler* - `https://www.haskell.org/ghc`

[3] *Glasgow Haskell Compiler User's Guide (8.2.1)* - `http://downloads.haskell.org/~ghc/8.2.1/docs/html/users_guide`

ConstraintKinds: Allows types to contain constraints by introducing a kind for constraints[4]. Together with associated type synonyms, this allows constraints specific to a type-class instance (GHCUG 9.14.5).

KindSignatures: Implied by `TypeFamilies`. Adds syntactic support for specifying the kind of a type. For example, in the head of type class declarations or in the arguments of associated type synonyms (GHCUG 9.15.4).

2.3 GHC type checker plugins

GHC supports a plugin interface to extend its constraint solver. The plugins are provided to GHC as standard Haskell modules during compilation. Type checker plugins have been used to implement type system extensions such as type-level natural numbers [Dia15] and units of measure [Gun15]. I use the plugin mechanism to integrate polymonads and supermonads into Haskell [BN15; BN16]. I content myself with a brief explanation here, referring the reader to the earlier work and the GHC user's guide[5] for details.

GHC type checks code in program fragments, e.g., top-level function definitions. For each fragment, type checking and inference produces three sets of constraints. These three sets represent given, derived, and wanted constraints: Given constraints are provided by the programmer or inferred as part of a type signature, derived constraints are constraints that arise from another plugin, and wanted constraints are those constraints that require solving. For most practical purposes the derived constraints can be treated as given constraints, because the plugin that produced them made sure that they should hold. The constraint solver solves wanted constraints iteratively. If the constraint solver is not able to solve a constraint or make progress, it asks available plugins for help. Each plugin then processes all of the available constraints and either provides evidence for them or creates new constraints to guide the constraint solver during its next iteration.

2.4 Proofs

Unless noted otherwise, all definitions and results in this work are formalised and verified with the proof assistant Agda[6] [Nor07] in version 2.5.3 together with the Agda standard library in version 0.14. This gives me confidence in my results, but also implies some limitations.

Many proofs in the following chapters relate to structures in Haskell or category theory. To my knowledge there is no complete formal or categorical model

[4] *Constraint Kinds for GHC (10. September 2011)* - `http://blog.omega-prime.co.uk/2011/09/10/constraint-kinds-for-ghc/`

[5] *Glasgow Haskell Compiler User's Guide (8.2.1), Section 11.3.4* - `https://downloads.haskell.org/~ghc/8.2.1/docs/html/users_guide/extending_ghc.html#typechecker-plugins`

[6] *The Agda Wiki* - `http://wiki.portal.chalmers.se/agda`

of Haskell to base these proofs on. Therefore, I formalised the Haskell structures that I work with in the category Set of types and total functions. Whenever I speak of Haskell structures that are equivalent or in one-to-one correspondence to other structures the proofs are based on a formalisation of these structures in Set.

To relate the proofs back to Haskell, it has to be assumed that the formalised Haskell structures are terminating, total, and do not contain bottom. Due to my direct translation of Haskell structures into Agda the expressive power of these formalisations is limited in the aforementioned way. Since proper instances of monads or applicatives usually fulfil all of those requirements, these assumptions do not weaken the results significantly. It should also be noted that modelling aspects of Haskell such as partiality in Agda provides a significant overhead to the burden of formalisation and proof. Therefore, my choice of a direct translation with the above limitations is mostly pragmatic and to avoid distractions.

Most of my Agda proofs utilise function extensionality, i.e., two functions are considered equal if for all inputs their outputs always match. Since function extensionality cannot be proven in Agda I postulate it[7]. Unless noted otherwise, this is the only postulate that is used in my Agda formalisations and proofs.

The Agda source code of the proofs is available in a public Git repository[8]. Whenever I refer to a proof or formalisation, the Agda module containing it is referenced as a footnote.

[7] Agda formalisation: `Extensionality`

[8] GitHub: *jbracker/polymonad-proofs* - `https://github.com/jbracker/polymonad-proofs`

Chapter 3

Generalised monads and applicatives

This chapter gives a detailed overview of the monadic and applicative generalisation unified by my work. There are several other generalisations, but my work focuses on the, to my knowledge, most popular variations: graded, indexed, and constrained. Other generalisations are discussed in Section 7 on related work.

The discussion of monadic and applicative notions often refers to some n-ary type constructors K and its arguments $a_1, \ldots, a_n$. I refer to K as the *base constructor* and $a_1, \ldots, a_{n-1}$ as the *indices* of K. The *result type* of the monadic or applicative computation is a_n.

3.1 Monadic notions

This section introduces three popular generalisations of monads: indexed, graded, and constrained monads. Each notion is introduced with an example and its usual representation in Haskell. The ultimate aim is to define a monadic notion that unifies all of these generalised notions with standard monads.

An overview of all operations and laws associated with each of the monadic notions is given in Figure 3.1 and 3.2, respectively. These overviews are provided as a reference point to directly compare different monadic notions with each other. All parts of the table are discussed and explained in the following sections.

3.1.1 Standard monads

To highlight the differences between the generalisations, I first reintroduce standard monads [Mog88]. In Haskell, they are represented by a type class containing the bind and return operation:

```
class (Functor m) => Monad (m :: * -> *) where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

Monad	**Type constructor**	bind : $\forall\ \alpha\ \beta.$	return : $\forall\ \alpha.$
Standard	$\mathsf{M} : * \to *$	$\mathsf{M}\ \alpha \to (\alpha \to \mathsf{M}\ \beta) \to \mathsf{M}\ \beta$	$\alpha \to \mathsf{M}\ \alpha$
Constrained	$\mathsf{M} : * \to *$	$(\mathsf{BindCts}\ \alpha\ \beta) \Rightarrow$ $\mathsf{M}\ \alpha \to (\alpha \to \mathsf{M}\ \beta) \to \mathsf{M}\ \beta$	$(\mathsf{ReturnCts}\ \alpha) \Rightarrow$ $\alpha \to \mathsf{M}\ \alpha$
Indexed	$\mathsf{M} : I \to I \to * \to *$	$\forall\ i\ j\ k.$ $\mathsf{M}\ i\ j\ \alpha \to (\alpha \to \mathsf{M}\ j\ k\ \beta) \to \mathsf{M}\ i\ k\ \beta$	$\forall\ i.$ $\alpha \to \mathsf{M}\ i\ i\ \alpha$
Graded	$\mathsf{M} : E \to * \to *$	$\forall\ i\ j.$ $\mathsf{M}\ i\ \alpha \to (\alpha \to \mathsf{M}\ j\ \beta) \to \mathsf{M}\ (i \diamond j)\ \beta$	$\alpha \to \mathsf{M}\ \varepsilon\ \alpha$

I - Kind of the indices.

E - Kind of the elements of a monoid.

$\diamond$ - Operation of the monoid E.

ε - Neutral element of the monoid E.

Figure 3.1: Types of the bind and return operations for different monadic notions.

	Standard	**Constrained**	**Indexed**	**Graded**
Left identity: `return` a >>= $f\ =\ f\ a$				
		$(\mathsf{BindCts}_l\ \alpha\ \beta, \mathsf{ReturnCts}_l\ \alpha) \Rightarrow$		
a :	α	α	α	α
f :	$\alpha \to \mathsf{M}\ \beta$	$\alpha \to \mathsf{M}\ \beta$	$\alpha \to \mathsf{M}\ j\ k\ \beta$	$\alpha \to \mathsf{M}\ j\ \beta$
Right identity: m >>= `return` $=\ m$				
		$(\mathsf{BindCts}_l\ \alpha\ \alpha, \mathsf{ReturnCts}_l\ \alpha) \Rightarrow$		
m :	$\mathsf{M}\ \alpha$	$\mathsf{M}\ \alpha$	$\mathsf{M}\ i\ j\ \alpha$	$\mathsf{M}\ i\ \alpha$
Associativity: $(m$ >>= $f)$ >>= $g\ =\ m$ >>= $(\lambda x \to f\ x$ >>= $g)$				
		$(\ \mathsf{BindCts}_l\ \alpha\ \beta, \mathsf{BindCts}_{lr}\ \beta\ \gamma$ $,\ \mathsf{BindCts}_r\ \alpha\ \gamma) \Rightarrow$		
m :	$\mathsf{M}\ \alpha$	$\mathsf{M}\ \alpha$	$\mathsf{M}\ i\ j\ \alpha$	$\mathsf{M}\ i\ \alpha$
f :	$\alpha \to \mathsf{M}\ \beta$	$\alpha \to \mathsf{M}\ \beta$	$\alpha \to \mathsf{M}\ j\ k\ \beta$	$\alpha \to \mathsf{M}\ j\ \beta$
g :	$\beta \to \mathsf{M}\ \gamma$	$\beta \to \mathsf{M}\ \gamma$	$\beta \to \mathsf{M}\ k\ l\ \gamma$	$\beta \to \mathsf{M}\ k\ \gamma$

$_l$ - Constraint originates from the left-hand side of the equation.

$_r$ - Constraint originates from the right-hand side of the equation.

Figure 3.2: Laws of different monadic notions.

Instances of this class are expected to obey the monad laws (Figure 3.2) with suitable a, f, g, and m. As indicated by the superclass, each standard monad has an underlying functor that provides the `fmap` function which maps the result type of a monadic computation with a given function:

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

Section 5.1.7 explains why I do not mention the recently added[1] `Applicative` superclass for `Monad` in Haskell.

The versatility of standard monads is well known. Monads are used to structure state, exceptions, parsing, non-determinism, concurrency, continuations, and many other side-effects [Mog91; Wad92] as well as embedded domain-specific languages (EDSLs) [BG14].

3.1.2 Indexed monads

Indexed monads encode the idea of Hoare logic [Hoa69] in the type system. They allow to state conditions or states, that have to be true before and after a monadic computation, as a part of the type through additional indices.

I choose session types[2] [PT08] to demonstrate how the need for indexed monads arises. The goal of session types is to verify the proper execution of a communication protocol on the type level. The following types are introduced to express the type-level communication protocol:

```
data Read a p
data Write a p
data Done

-- First read an integer, then write a string and
-- finally finish communication.
type SomeProtocol = Read Int (Write String Done)
```

The type `SomeProtocol` expresses a simple protocol at the type level using the `Read`, `Write`, and `Done` type. To provide operations that execute the protocol, a knowledge of the executed protocol and how much of the protocol has been executed is required. A popular way to write such functions is to use Hoare logic to show the state of the protocol before and after an operation. Thus, the type `Session i j a` may be interpreted to encode the execution of a protocol: `i` and `j` represent the protocol before and after the computation and `a` is the result type of the computation. For example, the `read` and `write` operation can be typed as follows:

[1] *Functor-Applicative-Monad Proposal* - https://wiki.haskell.org/Functor-Applicative-Monad_Proposal

[2] Hackage: *simple-sessions* - http://hackage.haskell.org/package/simple-sessions

```
read :: Session (Read a i) i a
write :: a -> Session (Write a i) i ()
```

Note that a protocol, such as `SomeProtocol`, can be interpreted as a stack of operations that need to be performed to execute the protocol. Hence, the `read` function requires the protocol stack to contain a `Read` operation before it is executed and removes the `Read` operation from the stack after it is done. The `write` function reduces the protocol stack in the same manner.

Since `Session` represents a side-effect, it would be convenient if it formed a monad to allow us to sequence operations and use the `do`-notation. The bind operation for `Session` has the following type that encodes the law of composition in Hoare logic:

```
(>>=) :: Session i j a
      -> (a -> Session j k b)
      -> Session i k b
```

The return operation does not execute the protocol and therefore represents the law for empty statements:

```
return :: a -> Session i i a
```

Due to the additional two indices, `Session` cannot form a standard monad, but it does form an indexed monad [Wad94; Atk09]. In Haskell, indexed monads are usually represented through the following type class:

```
class IndexedMonad (m :: k -> k -> * -> *) where
  (>>=)  :: m i j a -> (a -> m j k b) -> m i k b
  return :: a -> m i i a
```

Note that the kind of `m` uses the polymorphic kind variable `k` to specify the kind of the indices. A kind variable is used, because it may be instantiated with any kind, allowing instances to choose their indices freely. The kind variable also ensures that both indices have the same kind.

The bind operation may only compose two computations if the intermediate state `j` matches. The result of the bind operation has the state before the first computation and the state after the second. The `return` operation has no side-effects. Therefore, the `return` operation does not change the state.

The laws for indexed monads are exactly the same as for standard monads (Figure 3.2). Only the types of the involved variables a, f, g, and m changes. An overview of the types of the involved variables for each generalisation is given in Figure 3.2.

In contrast to standard monads, indexed monads have a *family* of underlying functors: One for each possible pair of indices. The family of underlying functors can be expressed with a generalised `IndexedFunctor` type class:

```
class IndexedFunctor (f : k -> k -> * -> *) where
  fmap :: (a -> b) -> f i j a -> f i j b
```

The `IndexedFunctor` type class can then be used as a superclass for `IndexedMonad` to reflect the relationship in Haskell.

It is also possible to express the family of underlying functors with the standard `Functor` class. For an indexed monad `M`, the family of associated functors can be given through the following instance:

```
instance Functor (M i j) where
  fmap = ...
```

However, this approach is only able to express the superclass constraints for `IndexedMonad` properly if advanced type system extensions such as the recently presented quantified class constraints [Bot+17] are available. In the absence of such extensions, as is the case at the time of writing, the `Functor` superclass constraints for `IndexedMonad` cannot be enforced due to the differently kinded type constructors that are required for `Functor` and `IndexedMonad`, respectively.

Other examples of indexed monads include composable continuations [Wad94] and typed state [Atk09]. Several packages on Hackage provide implementations of indexed monads[3,4,5]. The *indexed* package provides the analogue of the above `IndexedMonad` and `IndexedFunctor` type classes.

3.1.3 Graded monads

Graded monads introduce one additional index to their type that is inhabited by a monoid to keep track of invariants or internal state statically.

Vectors that encode their length as part of their type are an example of where the need for graded monads arises. A vector has the type `Vector n a` where `n` is the number of elements in the vector and `a` is the type of the elements. Many of the well-known functions for lists can also be defined for `Vector`, but giving a monad instance for `Vector` is problematic. For lists, the bind operation is implemented through `concatMap`:

```
concatMap :: [a] -> (a -> [b]) -> [b]
```

The type of `concatMap` for vectors has a similar but slightly different shape:

```
concatMap :: Vector n a
          -> (a -> Vector m b)
          -> Vector (n * m) b
```

Note that there are two different lengths that need to be multiplied in the resulting vector. Therefore, the version of `concatMap` for `Vector` does not provide a standard bind operation, although its type is similar.

The type of `return` for `Vector` causes similar issues:

[3]Hackage: *indexed* - `http://hackage.haskell.org/package/indexed`
[4]Hackage: *monad-param* - `http://hackage.haskell.org/package/monad-param`
[5]Hackage: *index-core* - `http://hackage.haskell.org/package/index-core`

```
return :: a -> Vector 1 a
```

An instance of `Monad` for `(Vector 1)` cannot provide a bind operation with the type demonstrated above.

There are many more examples where the same problem arises: information flow control [DP11], heterogeneous state and fine-grained composable control of side-effects [OP14]. In all cases, there is a need to encode certain information or invariants (vector length) about the computation in an additional index. The bind operation then combines the additional indices of the argument computations in the index of the result computation.

Graded monads [Wad98; Kat14; OP14] introduce said additional index in their type constructor. The new index has a monoidal structure and is used to encode the information or invariants of a computation. When two computations are composed using the bind operation, their effects are combined together using the monoid operation (multiplication). The return operation has no side-effects and therefore always has the neutral element of the monoid as index (1).

The monoidal structure cannot be captured at the type level in standard Haskell. Therefore, many implementations[6,7] use associated type synonyms to express the neutral element (`Unit`) and the monoid operation (`Plus`):

```
class GradedMonad (m :: k -> * -> *) where
  type Unit m :: k
  type Plus m (i :: k) (j :: k) :: k
  type Inv  m (i :: k) (j :: k) :: Constraint

  (>>=)  :: (Inv m i j)
         => m i a -> (a -> m j b) -> m (Plus m i j) b
  return :: a -> m (Unit m) a
```

Again, the kind variable `k` determines the kind of the monoid's values. A polymorphic kind variable is required, because, as with indexed monads, this gives the user a choice to use values lifted to the kind level, e.g. lists or natural numbers, as monoid. Depending on the monoid, constraints on the indices may be required to ensure that the instance works as expected. These constraints can be specified through the associated constraint `Inv`. Note that this is different from constraints on the result types, which lead to the constrained monads that are discussed in the next section.

Just as indexed monads, graded monads also have an underlying family of functors: one for each element of the monoid. Whether a superclass constraint for those functors can be specified depends on whether a generalised functor class or the standard one is used. The arguments for either case are analogous to those made for indexed monads.

The laws, again, are exactly the same as for standard monads (Figure 3.2) with a, f, g, and m of appropriate type.

[6]Hackage: *effect-monad* - `http://hackage.haskell.org/package/effect-monad`
[7]Hackage: *monad-param* - `http://hackage.haskell.org/package/monad-param`

3.1.4 Constrained monads

The final generalisation I want to include introduces constraints on the result types of the monadic computations.

A well-known example of this generalisation is Haskell's representation of unordered finite sets `Set`[8] [Hug99]. The `Set` type would form a standard monad just as lists do, but cannot because it requires an ordering (`Ord`) on its elements to allow an efficient representation [Ada93]. This leads to the following types for `fmap`, >>= and `return`:

```
fmap   :: Ord b => (a -> b) -> Set a -> Set b
return :: a -> Set a
(>>=)  :: Ord b => Set a -> (a -> Set b) -> Set b
```

These signatures can be accommodated through associated constraints:

```
class ConstrainedMonad (m :: * -> *) where
  type BindCts m (a :: *) (b :: *) :: Constraint
  type ReturnCts m (a :: *) :: Constraint

  (>>=)  :: (BindCts m a b) => m a -> (a -> m b) -> m b
  return :: (ReturnCts m a) => a -> m a
```

The laws are again the same as for standard monads (Figure 3.2). Note that my work on categorical models for constrained monads suggests that the constraints have to obey additional laws for consistency as is explained in Section 6.1.9 and 6.2.5, although these laws are only important if the constraints on `a` and `b` are different from each other.

Constrained monads necessitate the introduction of constrained functors to implement their underlying functor. Therefore, a redefined type class for `Functor` is required:

```
class Functor (f :: * -> *) where
  type FunctorCts f (a :: *) (b :: *) :: Constraint

  fmap :: (FunctorCts f a b) => (a -> b) -> f a -> f b
```

This new functor type class is necessary, because, unlike indexed or graded monads that just introduce additional indices, the standard functor type class cannot express constraints on the result types.

With the generalised `ConstrainedMonad` and `Functor` type class it is possible to give appropriate instances for `Set`.

```
instance Functor Set where
  type FunctorCts Set a b = (Ord b)
  fmap = S.map
```

[8] Hackage: *containers* - `http://hackage.haskell.org/package/containers`

```
instance ConstrainedMonad Set where
  type BindCts Set a b = (Ord b)
  type ReturnCts Set a = ()

  m >>= f = S.unions (S.map f m)
  return = S.singleton
```

The qualifier `S` refers to the modules that provides the `Set` data type and its associated functions. Note that the example instances only require `Ord` constraints on the second result type `b` in `fmap` and `>>=`. The `return` function does not enforce an ordering on its result type, because `S.singleton` does not require an ordering either.

There are other examples of constrained monads; e.g., finite quantum vectors [VAS06]. Domain specific languages [PAS12; BG14] provide a recurring need for constrained monads as well.

I am aware of two packages[9,10] on Hackage that provide support for constrained monads.

3.2 Applicative notions

Over the last decade, applicatives have emerged as an increasingly popular computational notion alongside monads. There are strong connections between monads and applicatives, as substantiated by the latest changes in the Haskell standard library, where `Applicative` is now a superclass of `Monad`.

This section develops and introduces the associated generalised applicatives of the monadic notions discussed in the previous section.

I am not aware of any literature or project that uses or requires the generalised applicatives discussed in the following sections. However, it is possible to derive a generalised applicative from each monadic notion. This derivation is based on how a standard applicative can be derived from a standard monad and transfers this process to indexed, graded, and constrained monads to get their corresponding applicatives.

As in the previous section an overview of all the different applicative operations and laws is provided in Figure 3.3 and 3.4, respectively. Both tables are discussed in the following sections and provide a reference point to compare the different applicative notions with each other.

[9] Hackage: *rmonad* - `http://hackage.haskell.org/package/rmonad`

[10] Hackage: *constraint-classes* - `http://hackage.haskell.org/package/constraint-classes`

Applicative	**Type constructor**	ap : $\forall\ \alpha\ \beta.$	pure : $\forall\ \alpha.$
Standard	$\mathsf{M} : * \to *$	$\mathsf{M}\ (\alpha \to \beta) \to \mathsf{M}\ \alpha \to \mathsf{M}\ \beta$	$\alpha \to \mathsf{M}\ \alpha$
Constrained	$\mathsf{M} : * \to *$	$(\mathsf{ApCts}\ \alpha\ \beta) \Rightarrow$ $\mathsf{M}\ (\alpha \to \beta) \to \mathsf{M}\ \alpha \to \mathsf{M}\ \beta$	$(\mathsf{PureCts}\ \alpha) \Rightarrow$ $\alpha \to \mathsf{M}\ \alpha$
Indexed	$\mathsf{M} : I \to I \to * \to *$	$\forall\ i\ j\ k.$ $\mathsf{M}\ i\ j\ (\alpha \to \beta) \to \mathsf{M}\ j\ k\ \alpha \to \mathsf{M}\ i\ k\ \beta$	$\forall\ i.$ $\alpha \to \mathsf{M}\ i\ i\ \alpha$
Graded	$\mathsf{M} : E \to * \to *$	$\forall\ i\ j.$ $\mathsf{M}\ i\ (\alpha \to \beta) \to \mathsf{M}\ j\ \alpha \to \mathsf{M}\ (i \diamond j)\ \beta$	$\alpha \to \mathsf{M}\ \varepsilon\ \alpha$

I - Kind of the indices.

$\diamond$ - Operation of the monoid E.

E - Kind of the elements of a monoid.

ε - Neutral element of the monoid E.

Figure 3.3: Types of the operators of different generalisations of applicatives.

	Standard	**Constrained**	**Indexed**	**Graded**
Identity: `pure id <*>` $u = u$				
		$(\mathsf{ApCts}_l\ \alpha\ \alpha, \mathsf{PureCts}_l\ (\alpha \to \alpha)) \Rightarrow$		
u :	$\mathsf{M}\ \alpha$	$\mathsf{M}\ \alpha$	$\mathsf{M}\ i\ j\ \alpha$	$\mathsf{M}\ i\ \alpha$
Composition: `pure (.) <*>` u `<*>` v `<*>` $w = u$ `<*>` $(v$ `<*>` $w)$				
		($\mathsf{ApCts}_l\ (\beta \to \gamma)\ ((\alpha \to \beta) \to (\alpha \to \gamma))$, $\mathsf{ApCts}_l\ (\alpha \to \beta)\ (\alpha \to \gamma)$, $\mathsf{ApCts}_l\ \alpha\ \gamma$, $\mathsf{PureCts}_l\ ((\beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \gamma))$, $\mathsf{ApCts}_r\ \beta\ \gamma$, $\mathsf{ApCts}_r\ \alpha\ \beta$) $\Rightarrow$		
u :	$\mathsf{M}\ (\beta \to \gamma)$	$\mathsf{M}\ (\beta \to \gamma)$	$\mathsf{M}\ i\ j\ (\beta \to \gamma)$	$\mathsf{M}\ i\ (\beta \to \gamma)$
v :	$\mathsf{M}\ (\alpha \to \beta)$	$\mathsf{M}\ (\alpha \to \beta)$	$\mathsf{M}\ j\ k\ (\alpha \to \beta)$	$\mathsf{M}\ j\ (\alpha \to \beta)$
w :	$\mathsf{M}\ \alpha$	$\mathsf{M}\ \alpha$	$\mathsf{M}\ k\ l\ \alpha$	$\mathsf{M}\ k\ \alpha$
Homomorphism: `pure` f `<*> pure` $a =$ `pure` $(f\ a)$				
		($\mathsf{ApCts}_l\ \alpha\ \beta$, $\mathsf{PureCts}_l\ (\alpha \to \beta)$, $\mathsf{PureCts}_l\ \alpha$, $\mathsf{PureCts}_r\ \beta) \Rightarrow$		
f :	$\alpha \to \beta$	$\alpha \to \beta$	$\alpha \to \beta$	$\alpha \to \beta$
a :	α	α	α	α
Interchange: u `<*> pure` $a =$ `pure` $(\$\ a)$ `<*>` u				
		($\mathsf{ApCts}_l\ \alpha\ \beta$, $\mathsf{PureCts}_l\ \alpha$, $\mathsf{ApCts}_r\ (\alpha \to \beta)\ \beta$, $\mathsf{PureCts}_r\ ((\alpha \to \beta) \to \beta)) \Rightarrow$		
u :	$\mathsf{M}\ (\alpha \to \beta)$	$\mathsf{M}\ (\alpha \to \beta)$	$\mathsf{M}\ i\ j\ (\alpha \to \beta)$	$\mathsf{M}\ i\ (\alpha \to \beta)$
a :	α	α	α	α

$_l$ - Constraint originates from the left-hand side of the equation.

$_r$ - Constraint originates from the right-hand side of the equation.

Figure 3.4: Laws of different generalisations of applicatives.

3.2.1 Standard applicatives

Drawing from work on parser combinators by Swierstra and Duponcheel [SD96], standard applicatives were introduced by McBride and Paterson [MP08]. They consists of the ap (<*>) and `pure` operation. The ap operation applies an encapsulated function to an encapsulated value and `pure` simply lifts a value into the applicative:

```
class Applicative (f :: * -> *) where
  (<*>) :: f (a -> b) -> f a -> f b
  pure  :: a -> f a
```

Just as monads, applicatives need to fulfil as set of laws (Figure 3.4) and they also have a variety of use cases. The many popular applications include parsers, concurrency [Mar+14], database query languages [Gio+11], and other EDSLs.

An applicative can be derived for any standard monad `m` in the following way:

```
(<*>) :: m (a -> b) -> m a -> m b
mf <*> ma = mf >>= \f -> fmap f ma

pure :: a -> m a
pure = return
```

The monad laws then imply the applicative laws[11]. This derivation can be used as a technique to derive generalised applicatives from the corresponding generalised monads.

3.2.2 Indexed applicatives

Deriving the applicative of an indexed monad delivers the following result[12]:

```
class IndexedApplicative (f :: p -> p -> * -> *) where
  (<*>) :: f i j (a -> b) -> f j k a -> f i k b
  pure  :: a -> f i i a
```

Notice, that the ap and pure operation resemble the composition and empty statement law of Hoare logic in exactly the same way as bind and return operation did.

An indexed applicative also has a family of associated functors analogous to that of the indexed monad.

3.2.3 Graded applicatives

The graded applicative can also be derived as expected[13]:

[11] Agda proof: `Haskell.Applicative`
[12] Agda proof: `Haskell.Parameterized.Indexed.Applicative.FromIndexedMonad`
[13] Agda proof: `Haskell.Parameterized.Graded.Applicative.FromGradedMonad`

```
class GradedApplicative (f :: k -> * -> *) where
  type Unit f :: k
  type Plus f (i :: k) (j :: k) :: k
  type Inv  f (i :: k) (j :: k) :: Constraint

  (<*>) :: (Inv f i j)
        => f i (a -> b) -> f j a -> f (Plus f i j) b
  pure  :: a -> f (Unit f) a
```

As for graded monads, associated type synonyms and constraints have to be provided to allow the declaration of the type-level monoid for the instance.

Again, a graded applicative has a family of associated functors analogous to that of the graded monad.

3.2.4 Constrained applicatives

Like the previous two generalised applicatives, the constrained applicative can be derived from the notion of a constrained monad[14]:

```
class ConstrainedApplicative (f :: * -> *) where
  type ApCts f (a :: *) (b :: *) :: Constraint
  type PureCts f (a :: *) :: Constraint

  (<*>) :: (ApCts f a b) => f (a -> b) -> f a -> f b
  pure  :: (PureCts f a) => a -> f a
```

The new associated constraints `ApCts` and `PureCts` are introduced for two reasons. First, depending on the implementation, a set of constraints different from those on the bind or return operation may be required. Second, the generalised applicatives are not required to be a superclass of the generalised monads as is explained in Section 5.1.7.

The formalised derivation[14] works with the categorical representation of constrained monads and constrained applicatives. Therefore, reading Section 6.1.9, 6.2.5, and 6.3 may be helpful to understand the formalisation and why it applies.

As Section 6.1.9 and 6.3.3 explain in more detail, the constraints have to obey additional laws for consistency. These additional laws arise because applicatives are more then simple functors in category theory, they are monoidal functors.

Note that, as for constrained monads, the underlying functor has to be represented with the constrained `Functor` class introduced in Section 3.1.4.

[14] Agda proof: `Theory.Functor.Monoidal.Properties.FromRelativeMonad`

Chapter 4

Polymonads

This chapter introduces polymonads [Hic+14] as a possible approach to unify monadic generalisations in Haskell. Polymonads only present a first stepping stone to a unified notions as they do not support generalised applicatives, constrained monads, or parameterised monads with non-phantom indices.

In the following sections I introduce polymonads (Section 4.1) and show how my work integrates them in Haskell (Section 4.2). Section 4.3 presents illustrative formal and practical examples of polymonads in Haskell. Afterwards I explain important properties of polymonads (Section 4.4) and their implementation as a plugin for GHC (Section 4.5). Finally, I recapitulate how the Haskell implementation has achieved my goals and which limitations it has (Section 4.6).

The source code for the polymonad implementation in Haskell and the examples presented in the following sections are available in a public Git repository[1].

4.1 Definition

A polymonad consists of a pair $(\mathscr{M}, \Sigma)$ of two sets together with associated laws. The set $\mathscr{M}$ contains unary type constructors; it always includes the constructor Id, where $\mathsf{Id}\ \tau = \tau$. The set Σ contains bind operations that have the type

$$\forall\ \alpha\ \beta.\ \mathsf{M}\ \alpha \to (\alpha \to \mathsf{N}\ \beta) \to \mathsf{P}\ \beta$$

with $\mathsf{M}, \mathsf{N}, \mathsf{P} \in \mathscr{M}$. The type of the bind operations in Σ is a generalisation of the type for the standard bind operation. Polymonads thus generalise the notion of a monad in a way that allows the involved type constructors and their indices to vary over a computation as long as the prerequisite laws are satisfied. In the interest of brevity, following Hicks et al. [Hic+14], the following notation is used when referring to the type of bind operations:

$$(\mathsf{M}, \mathsf{N}) \rhd \mathsf{P} \quad \stackrel{\mathsf{def}}{=} \quad \forall\ \alpha\ \beta.\ \mathsf{M}\ \alpha \to (\alpha \to \mathsf{N}\ \beta) \to \mathsf{P}\ \beta,$$

[1] GitHub: *jbracker/polymonad-plugin* - `https://github.com/jbracker/polymonad-plugin`

Definition 4.1 provides a formal definition of polymonads[2]. The definition is slightly altered from its original form given by Hicks et al. [Hic+14] to make it more precise.

Definition 4.1 (Polymonad)**.** A *polymonad* $(\mathscr{M}, \Sigma)$ consists of a collection $\mathscr{M}$ of unary type constructors, with a distinguished element $\mathsf{Id} \in \mathscr{M}$, such that $\mathsf{Id}\ \tau = \tau$, and a collection Σ of bind operations such that the laws below hold. For all $\mathsf{M}, \mathsf{N}, \mathsf{P}, \mathsf{Q}, \mathsf{R}, \mathsf{S}, \mathsf{U} \in \mathscr{M}$:

Functor

$\exists\, b : (\mathsf{M}, \mathsf{Id}) \rhd \mathsf{M}.\ b \in \Sigma$ and
$\forall\, b : (\mathsf{M}, \mathsf{Id}) \rhd \mathsf{M}.\ [b \in \Sigma \implies b\ m\ (\lambda\ y.\ y) = m]$

Paired morphisms

$[\exists\, b_1 : (\mathsf{M}, \mathsf{Id}) \rhd \mathsf{N}.\ b_1 \in \Sigma] \iff [\exists\, b_2 : (\mathsf{Id}, \mathsf{M}) \rhd \mathsf{N}.\ b_2 \in \Sigma]$
and
$\forall\, b_1 : (\mathsf{M}, \mathsf{Id}) \rhd \mathsf{N}, b_2 : (\mathsf{Id}, \mathsf{M}) \rhd \mathsf{N}.$
$[\{\ b_1, b_2\ \} \subseteq \Sigma \implies b_1\ (f\ v)\ (\lambda\ y.\ y) = b_2\ v\ f]$

Diamond

$[\exists\, \mathsf{P} \in \mathscr{M}. \exists\, b_1 : (\mathsf{M}, \mathsf{N}) \rhd \mathsf{P}, b_2 : (\mathsf{P}, \mathsf{R}) \rhd \mathsf{T}.\ \{\ b_1, b_2\ \} \subseteq \Sigma]$
$\iff$
$[\exists\, \mathsf{S} \in \mathscr{M}. \exists\, b_3 : (\mathsf{N}, \mathsf{R}) \rhd \mathsf{S}, b_4 : (\mathsf{M}, \mathsf{S}) \rhd \mathsf{T}.\ \{\ b_3, b_4\ \} \subseteq \Sigma]$

Associativity

$\forall\, b_1 : (\mathsf{M}, \mathsf{N}) \rhd \mathsf{P}, b_2 : (\mathsf{P}, \mathsf{R}) \rhd \mathsf{T},$
$b_3 : (\mathsf{N}, \mathsf{R}) \rhd \mathsf{S}, b_4 : (\mathsf{M}, \mathsf{S}) \rhd \mathsf{T}.$
$[\{\ b_1, b_2, b_3, b_4\ \} \subseteq \Sigma \implies b_2\ (b_1\ m\ f)\ g = b_4\ m\ (\lambda\ x.\ b_3\ (f\ x)\ g)]$

Closure

$\exists\, b_1 : (\mathsf{M}, \mathsf{N}) \rhd \mathsf{P}, b_2 : (\mathsf{S}, \mathsf{Id}) \rhd \mathsf{M},$
$b_3 : (\mathsf{T}, \mathsf{Id}) \rhd \mathsf{N}, b_4 : (\mathsf{P}, \mathsf{Id}) \rhd \mathsf{U}.$
$[\{\ b_1, b_2, b_3, b_4\ \} \subseteq \Sigma \implies \exists\, b : (\mathsf{S}, \mathsf{T}) \rhd \mathsf{U}.\ b \in \Sigma]$

The functor law ensures that there is a bind operation resembling a functor mapping for each type constructor; it also assures that the mapping function only operates on the computed value and does not produce side-effects, as to be expected from a functor. Thereby, it resembles the functor prerequisite of standard monads. The presented functor law slightly deviates from the one presented by Hicks et al. [Hic+14]; it includes a separate quantification for the equation. In the original version the quantification of the equation is missing and it therefore seems as if the equation only needs to hold for the one existing bind operation. Both versions of Definition 4.1, the original and the presented one, are equivalent as proven in the formalisation[3].

[2] Agda formalisation: `Polymonad.Definition`
[3] Agda proof: `Hicks.Equivalency`

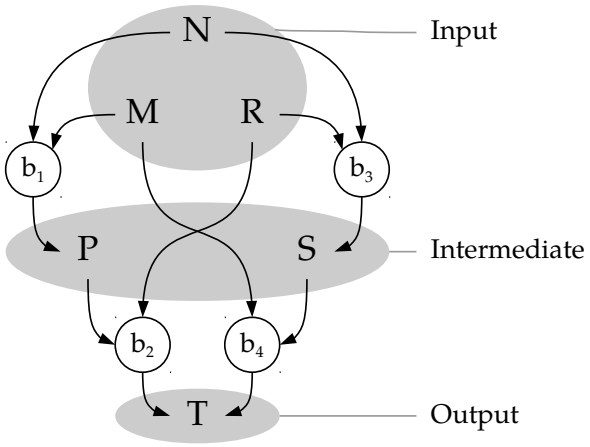


Figure 4.1: Control flow within the diamond and associativity law for polymonads.

The paired morphisms law ensures that, if there is a morphism from one type constructor to another, then the alternative way of expressing that morphism also exists with unchanged semantics.

The diamond and associativity laws are tied together. They express that a sequence of three monadic operations may be associated either way, and that the choice of intermediate monadic type (P or S) that may arise is irrelevant. The control flow and the intermediate types are illustrated in Figure 4.1.

The closure law, finally, expresses, that if a sensible composition of bind operations yields another possible bind operation, then that bind operation also exists.

From this point onward Id will always refer to the identity type constructor:

$$\mathsf{Id}\ \tau \stackrel{\mathsf{def}}{=} \tau.$$

The unary type constructors of a polymonad are often partial applications of type constructors of higher arity. To refer to this (finite) set of "generating" constructors, I define:

Definition 4.2 (Polymonad basis)**.** The *basis* of a polymonad is the smallest set of type constructors form which all unary type constructors of a polymonad can be obtained by partial application.

4.2 Representation in Haskell

A key advantage of implementing polymonads in Haskell is that they allow the do-notation to be used without having to be explicit about which monadic generalisation to use (cf. Section 1.1). For this to work the polymonad implementation uses the `RebindableSyntax` language extension (Section 2.2) to replace

the standard monad bind, sequence, and return operations with ones specific to polymonads. The representation of polymonads in Haskell thus needs to provide a replacement for the core functions used to translate the do-notation: >>=, >>, return, and fail. The standard >>= and >>-operator are replaced by their counterparts in the Polymonad type class:

```
class Polymonad m n p where
  (>>=) :: m a -> (a -> n b) -> p b
  (>>)  :: m a -> n b -> p b
  ma >> mb = ma >>= \_ -> mb
```

The Polymonad type class is derived from the set of bind operations that is part of each polymonad (Definition 4.1). The set of bind operations for a specific polymonad in Haskell is determined by the instances of the Polymonad type class.

The functions return and fail are regular polymorphic functions, implemented in terms of the Polymonad type class:

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

return :: (Polymonad Identity Identity m) => a -> m a
return x = Identity x >>= Identity

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

As the return function already illustrates, the Identity type from the base library provides the distinguished Id type constructor in Haskell. The representation uses Identity, because it is isomorphic to Id.

Using a type class to represent the bind operations of polymonads is justified and does not constitute a limitation. Section 4.4 expands on this topic when the uniqueness of bind operations is discussed (Section 4.4.1). Overlapping instances are not a problem for the same reason. In fact, as the examples in the next section show, overlapping instances are necessary.

4.3 Examples of polymonads

This section illustrates how different generalised monads can be instantiated using the Polymonad type class.

To implement a polymonad, all of its bind operations, as required by the polymonad laws (Definition 4.1), need to be provided as instances of the Polymonad type class.

A trivial example of a polymonad is the identity polymonad[4] with one bind operation.

[4] Agda proof: Polymonad.Identity

Example 4.3 (Identity polymonad)**.** Define the function `bindId` as

$$\texttt{bindId} : (\mathsf{Id}, \mathsf{Id}) \triangleright \mathsf{Id}$$
$$\texttt{bindId}\ x\ f \stackrel{\mathsf{def}}{=} f\ x$$

Then $\mathscr{M}_{\mathsf{Id}} \stackrel{\mathsf{def}}{=} \{\ \mathsf{Id}\ \}$ and $\Sigma_{\mathsf{Id}} \stackrel{\mathsf{def}}{=} \{\ \texttt{bindId}\ \}$ form the identity polymonad $(\mathscr{M}_{\mathsf{Id}}, \Sigma_{\mathsf{Id}})$ with Id as the distinguished identity type constructor.

Using the `Polymonad` type class in Haskell the identity polymonad can be instantiated through a single instance:

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

The following subsections illustrate how standard, graded, and indexed monads also form polymonads.

Note that all of the examples in this section were previously mentioned in an unpublished paper by Guts et al. [Gut+12] and the "Polymonadic Programming" paper [Hic+14]. In addition to restating them, I formalised and verified them.

4.3.1 Standard monads

Given a standard monad a polymonad can be defined in terms of it[5], i.e., there is an injection from standard monads into polymonads.

Example 4.4 (Polymonad formed by a standard monad)**.** If M is a monad with `return` and >>= as its return and bind operations, then M forms the polymonad $(\mathscr{M}_{\mathsf{M}}, \Sigma_{\mathsf{M}})$, where:

$$\begin{aligned}
\mathscr{M}_{\mathsf{M}} &\stackrel{\mathsf{def}}{=} \{\ \mathsf{Id}, \mathsf{M}\ \} \\
\Sigma_{\mathsf{M}} &\stackrel{\mathsf{def}}{=} \{\ \texttt{bindId} : (\mathsf{Id}, \mathsf{Id}) \triangleright \mathsf{Id}, \\
&\qquad (\texttt{>>=}) : (\mathsf{M}, \mathsf{M}) \triangleright \mathsf{M}, \\
&\qquad (\lambda m\ f.\ \texttt{return}\ (f\ m)) : (\mathsf{Id}, \mathsf{Id}) \triangleright \mathsf{M}, \\
&\qquad (\lambda m\ f.\ m \texttt{ >>= } (\texttt{return} \circ f)) : (\mathsf{M}, \mathsf{Id}) \triangleright \mathsf{M}, \\
&\qquad (\lambda m\ f.\ f\ m) : (\mathsf{Id}, \mathsf{M}) \triangleright \mathsf{M}\ \}
\end{aligned}$$

To compose computations and lift pure values into a monadic computation the >>=-operator and the `return`-function are required. Therefore, they supply the binds of type $(\mathsf{M}, \mathsf{M}) \triangleright \mathsf{M}$ and $(\mathsf{Id}, \mathsf{Id}) \triangleright \mathsf{M}$. The functor law then requires the existence of the $(\mathsf{M}, \mathsf{Id}) \triangleright \mathsf{M}$ and $(\mathsf{Id}, \mathsf{Id}) \triangleright \mathsf{Id}$ bind operation and the paired morphism law requires the $(\mathsf{Id}, \mathsf{M}) \triangleright \mathsf{M}$ bind operation. Once we have all five bind operations all of the other laws hold as a result.

Given Example 4.4 polymonad instances that work for all monads can be written:

[5] Agda proof: `Haskell.Monad.Polymonad`

```
instance (P.Monad m) => Polymonad m m m where
  m >>= f = m P.>>= f

instance (P.Monad m) => Polymonad Identity Identity m where
  (Identity a) >>= f = P.return ( runIdentity (f a) )

instance (P.Monad m) => Polymonad m Identity m where
  m >>= f = m P.>>= (P.return . runIdentity . f)

instance (P.Monad m) => Polymonad Identity m m where
  (Identity a) >>= f = f a
```

The qualifier `P` refers to the standard prelude. Note that if M is a monad, the `Polymonad m m m` instance provides two of the necessary bind operations of type $(\mathsf{Id},\mathsf{Id}) \triangleright \mathsf{Id}$ and $(\mathsf{M},\mathsf{M}) \triangleright \mathsf{M}$, respectively.

Also note, that the `Polymonad Identity Identity m` instance, provides the instance necessary to use the polymonad `return` function. In addition, the instance for $(\mathsf{M},\mathsf{Id}) \triangleright \mathsf{M}$ is implemented exactly as the functor law, that is implied by the monad laws[6], would suggest:

$$\texttt{fmap}\ f\ xs\ =\ xs \texttt{ >>= return} \circ f$$

All of the above instances overlap with each other, e.g., all of them match the constraint `Polymonad Identity Identity Identity`. Therefore, declaring these instances requires the language extension `OverlappingInstances`. This overlapping is not a problem, because, as I prove in Section 4.4.1, all bind operations involving the same type constructors are extensionally equal to each other. Hence, any of the instances can be picked in the case of overlapping, which is exactly what the developed plugin does (Section 4.5).

These instances allow programmers to easily port existing monadic code. The example also illustrates how the identity polymonad is part of every polymonad and why it requires special attention when the union of polymonads is presented in Section 4.4.3.

4.3.2 Indexed monads

Given a indexed monad a polymonad can be defined in terms of it[7], i.e., there is an injection from indexed monads into polymonads. This injection works similarly to the one defined in Example 4.4.

Example 4.5 (Polymonad formed by an indexed monad)**.** If M is a indexed monad with indices in I, `return` as its return and >>= as its bind operations,

[6] Agda proof: `Theory.Monad.Definition`

[7] Agda proof: `Haskell.Parameterized.Indexed.Polymonad`

then M forms the polymonad $(\mathscr{M}_{\mathsf{M}}, \Sigma_{\mathsf{M}})$, where:

$$
\begin{aligned}
\mathscr{M}_{\mathsf{M}} &\stackrel{\text{def}}{=} \{\ \mathsf{Id}, \mathsf{M}_{\mathsf{i},\mathsf{j}} \mid \mathsf{i},\mathsf{j} \in I\ \} \\
\Sigma_{\mathsf{M}} &\stackrel{\text{def}}{=} \{\ \mathtt{bindId} : (\mathsf{Id}, \mathsf{Id}) \rhd \mathsf{Id}, \\
&\qquad (\mathtt{>>=}) : (\mathsf{M}_{\mathsf{i},\mathsf{j}}, \mathsf{M}_{\mathsf{j},\mathsf{k}}) \rhd \mathsf{M}_{\mathsf{i},\mathsf{k}}, \\
&\qquad (\lambda m\ f.\ \mathtt{return}\ (f\ m)) : (\mathsf{Id}, \mathsf{Id}) \rhd \mathsf{M}_{\mathsf{i},\mathsf{i}}, \\
&\qquad (\lambda m\ f.\ m \mathtt{>>=} (\mathtt{return} \circ f)) : (\mathsf{M}_{\mathsf{i},\mathsf{j}}, \mathsf{Id}) \rhd \mathsf{M}_{\mathsf{i},\mathsf{j}}, \\
&\qquad (\lambda m\ f.\ f\ m) : (\mathsf{Id}, \mathsf{M}_{\mathsf{i},\mathsf{j}}) \rhd \mathsf{M}_{\mathsf{i},\mathsf{j}} \\
&\qquad \mid \mathsf{i},\mathsf{j},\mathsf{k} \in I\ \}
\end{aligned}
$$

Hence, the `Polymonad` instances are similar to those for standard monads:

```
instance (I.IndexedMonad m) =>
         Polymonad (m i j) (m j k) (m i k) where
  m >>= f = m I.>>= f

instance (I.IndexedMonad m) =>
         Polymonad Identity Identity (m i i) where
  (Identity a) >>= f = I.return ( runIdentity (f a))

instance (I.IndexedMonad m) =>
         Polymonad (m i j) Identity (m i j) where
  m >>= f = m I.>>= (I.return . runIdentity . f)

instance (I.IndexedMonad m) =>
         Polymonad Identity (m i j) (m i j) where
  (Identity a) >>= f = f a
```

The qualifier `I` refers to the module that provides the `IndexedMonad` type class. Note that the type class `Polymonad` is able to represent indexed monads and standard monads alike. There is no need to introduce different type classes for different monadic notions anymore. In fact, if the `IndexedMonad` type class were not there, then the instances could be written for any concrete indexed monad in the same manner. This promotes code reuse, because library functions and utilities for polymonads can be provided once, rather than separately for each monadic notion.

Note that the `Polymonad Identity Identity Identity` instance is still necessary. That instance is not listed for brevity.

4.3.3 Graded monads

Finally, any graded monad [Wad98; Kat14; OP14] also allows the definition of a polymonad in terms of it. Again there is an injection from the graded monads

into the polymonads[8]. As explained in Section 3.1.3, a graded monad provides a composable way of modelling effects by adding a monoidal index that describes the possible effects of a computation. The definition of a polymonad in terms of a graded monad is similar to the definitions in terms of standard or indexed monads.

Example 4.6 (Polymonad formed by a graded monad)**.** If M is a graded monad with indices from the monoid $(M, \diamond, e)$, `return` as its return and >>= as its bind operations, then M forms the polymonad $(\mathscr{M}_{\mathsf{M}}, \Sigma_{\mathsf{M}})$, where:

$$
\begin{aligned}
\mathscr{M}_{\mathsf{M}} &\stackrel{\text{def}}{=} \{\ \mathsf{Id}, \mathsf{M}_{\mathsf{i}} \mid \mathsf{i} \in M\ \} \\
\Sigma_{\mathsf{M}} &\stackrel{\text{def}}{=} \{\ \mathtt{bindId} : (\mathsf{Id}, \mathsf{Id}) \rhd \mathsf{Id}, \\
&\qquad (>\!\!>\!\!=) : (\mathsf{M}_{\mathsf{i}}, \mathsf{M}_{\mathsf{j}}) \rhd \mathsf{M}_{\mathsf{i} \diamond \mathsf{j}}, \\
&\qquad (\lambda m\ f.\ \mathtt{return}\ (f\ m)) : (\mathsf{Id}, \mathsf{Id}) \rhd \mathsf{M}_{\mathsf{e}}, \\
&\qquad (\lambda m\ f.\ m >\!\!>\!\!= (\mathtt{return} \circ f)) : (\mathsf{M}_{\mathsf{i}}, \mathsf{Id}) \rhd \mathsf{M}_{\mathsf{i}}, \\
&\qquad (\lambda m\ f.\ f\ m) : (\mathsf{Id}, \mathsf{M}_{\mathsf{i}}) \rhd \mathsf{M}_{\mathsf{i}} \\
&\qquad \mid \mathsf{i}, \mathsf{j} \in M\ \}
\end{aligned}
$$

As before, the `Polymonad` instances are similar to those of standard and indexed monads, but for graded monads the instance head requires additional constraints to account for the realisation of the type-level monoid in Haskell.

```
instance ( E.GradedMonad m
         , h ~ E.Plus m f g, E.Inv m f g) =>
         Polymonad (m (f :: k)) (m (g :: k)) (m (h :: k))
         where
  m >>= f = m E.>>= f

instance (E.GradedMonad m, h ~ E.Unit m) =>
         Polymonad Identity Identity (m (h :: k))
         where
  (Identity a) >>= f = E.return ( runIdentity (f a) )

instance ( E.GradedMonad m, E.Inv m f (Unit m)
         , f ~ E.Plus m f (E.Unit m)) =>
         Polymonad (m (f :: k)) Identity (m (f :: k))
         where
  m >>= f = m E.>>= (E.return . runIdentity . f)

instance (E.GradedMonad m) =>
         Polymonad Identity (m (g :: k)) (m (g :: k))
         where
  (Identity a) >>= f = f a
```

[8] Agda proof: `Haskell.Parameterized.Graded.Polymonad`

The identifier `E` refers to the module that provides the `GradedMonad` type class from Section 3.1.3. The associated constraints `Plus`, `Unit` and `Inv` provide the type-level monoid of kind `k`. Most of these constraints are a direct consequence of the types involved in the implementation. However, one particular constraint from the third instance may cause confusion:

```
f ~ Plus m f (Unit m)
```

This constraint appears to be recursive and therefore may seem unsolvable. However, the constraint is solvable if the `GradedMonad` instance ensures that its instantiations of `Unit` and `Plus` follow the monoid laws. Specifically, the constraint encodes the right identity of a monoid. Therefore, `Plus m f (Unit m)` always evaluates to `f` in a law abiding instance and the constraint becomes trivial.

The use of type equality constraints is required to specify the shape of the indices, because type synonym application is not allowed to appear on the right hand side of an instance head.

4.4 Properties of polymonads

Before I can explain how the GHC plugin for polymonads works, I need to cover several important properties that are central to the implementation of polymonads in Haskell.

4.4.1 Uniqueness of bind operations

A polymonad has a collection of bind operations Σ. When a polymonad is instantiated in Haskell, the bind operations from Σ become instances of the `Polymonad` type class. In standard Haskell there can only be one type class instance per combination of types in the instance head. With language extensions this restriction can be weakened by allowing overlapping instances. However, these specifics of the type class system raise questions:

- If there are several bind operations with the same type in a polymonad, is it enough to only encode one of them or are all instances necessary?
- If there are several overlapping instances that could all possibly provide the implementation of the same bind operation, does this cause a conflict?

Fortunately, the polymonad laws imply that bind operations with the same type have the same behaviour, i.e., they are extensionally equal[9]:

Proposition 4.7 (Uniqueness of bind operations)**.** Let $(\mathscr{M}, \Sigma)$ be a polymonad. Then, for all $\mathsf{M}, \mathsf{N}, \mathsf{P} \in \mathscr{M}$:

$$\forall\, b_1, b_2 : (\mathsf{M}, \mathsf{N}) \rhd \mathsf{P}.\ [\{\ b_1, b_2\ \} \subseteq \Sigma \implies b_1 = b_2]\,.$$

[9] Agda proof: `Polymonad.UniqueBinds`

Hence, it suffices if there is only one instance of a type class as it simply enforces the above result even if there are several bind operations that the instance represents.

Conversely, if there is a constraint without type variables and several overlapping type class instances that match it, then an arbitrary matching instance may be picked without jeopardising the runtime behaviour.

Of course, both of these statements only hold if the instances are law abiding.

4.4.2 Principality of polymonads

The central property required to implement polymonads is the principality of a polymonad[10].

Definition 4.8 (Principal polymonad)**.** A polymonad $(\mathscr{M}, \Sigma)$ is a *principal polymonad* if and only if for any set $F \subseteq \mathscr{M}^2$ with $F \neq \emptyset$, and any $\{ \mathsf{M}_1, \mathsf{M}_2 \} \subseteq \mathscr{M}$ such that

- $\forall\, (\mathsf{M}, \mathsf{M}') \in F.\ \exists\, b : (\mathsf{M}, \mathsf{M}') \rhd \mathsf{M}_1.\ b \in \Sigma$ and
- $\forall\, (\mathsf{M}, \mathsf{M}') \in F.\ \exists\, b : (\mathsf{M}, \mathsf{M}') \rhd \mathsf{M}_2.\ b \in \Sigma$,

then there exists $\hat{\mathsf{M}} \in \mathscr{M}$ such that

- $\exists\, b_1 : (\hat{\mathsf{M}}, \mathsf{Id}) \rhd \mathsf{M}_1, b_2 : (\hat{\mathsf{M}}, \mathsf{Id}) \rhd \mathsf{M}_2.\ \{ b_1, b_2 \} \subseteq \Sigma$ and
- $\forall\, (\mathsf{M}, \mathsf{M}') \in F.\ \exists\, b : (\mathsf{M}, \mathsf{M}') \rhd \hat{\mathsf{M}}.\ b \in \Sigma$.

$\hat{\mathsf{M}}$ is the principal join of F and is denoted as $\bigsqcup F$.

The notion of principality is important, because the principal join is used by the polymonad implementation to pick a type constructor for the ambiguous variables it encounters. In essence, if a polymonad is principal, this means that whenever there is a choice of type constructor, the overall effect is invariant under this choice, and it is furthermore always possible to make a canonical choice.

As an example, assume there are two bind operations with types $(\mathsf{M}, \mathsf{N}) \rhd \mathsf{t}$ and $(\mathsf{t}, \mathsf{Id}) \rhd \mathsf{P}$, where $\mathsf{M}, \mathsf{N}, \mathsf{P} \in \mathscr{M}$ and t is an ambiguous variable that could be resolved to a number of type constructors. The principality of the polymonad implies that there has to exist a type constructor that is suitable for t and that is $\bigsqcup F$. That this choice yields a correct and sound solution was shown by Hicks et al. [Hic+14].

The above Definition 4.8 of principality deviates from the original definition by Hicks et al. [Hic+14] in that it insists that F is non-empty. This restriction is important, because if $F = \emptyset$ the preconditions are always fulfilled and that means there has to exist an $\hat{\mathsf{M}}$ for any chosen M_1 and M_2. Personal communication with the authors of the "Polymonadic Programming" paper [Hic+14] confirmed that this behaviour is not intended.

[10] Agda formalisation: `Polymonad.Principal`

I have verified that standard monads[11] and parameterised monads[12] with phantom indices are principal polymonads. These results were mentioned in an unpublished paper by [Gut+12] and the "Polymonadic Programming" paper [Hic+14].

4.4.3 Union of polymonads

The polymonad implementation is based on the knowledge of which polymonad it is working with. The definition of principality (4.8) is only applicable in the context of a specific polymonad. However, the bind operations of polymonads are represented through type class instances in Haskell. Therefore, there is no clear distinction between different polymonads in Haskell: all there is, is a collection of `Polymonad` instances. Hence, it is unclear how to determine which bind operations belong to which polymonad and how to separate them before solving. As the examples show, such a separation is problematic, because some of the instances are shared between different polymonads.

Instead of trying to separate the instances I examined the union of polymonads, as given by the collection of type class instances in Haskell. Indeed, such a union constitutes a polymonad in its own right[13,14].

Under the assumptions that Id is the distinguished type constructor in *all* polymonads, the union of two polymonads is a polymonad itself.

Definition 4.9 (Unionable polymonad)**.** Let $(\mathscr{M}, \Sigma)$ be a polymonad. The polymonad $(\mathscr{M}, \Sigma)$ is called *unionable*[15] if the following condition holds:

$$\forall\, b : (\mathsf{M}, \mathsf{N}) \rhd \mathsf{Id}.\ [b \in \Sigma \implies \mathsf{M} = \mathsf{Id} \wedge \mathsf{N} = \mathsf{Id}]$$

Proposition 4.10 (Union of polymonads)**.** Let $(\mathscr{M}_1, \Sigma_1)$ and $(\mathscr{M}_2, \Sigma_2)$ be two unionable polymonads with the same distinguished type constructor Id such that

$$\mathscr{M}_{\mathsf{Id}} = \mathscr{M}_1 \cap \mathscr{M}_2 \text{ and } \Sigma_{\mathsf{Id}} = \Sigma_1 \cap \Sigma_2.$$

Then $(\mathscr{M}_1 \cup \mathscr{M}_2, \Sigma_1 \cup \Sigma_2)$ with the distinguished type constructor Id also forms a unionable polymonad.

Note that the union of two polymonads is unionable itself. Therefore, any number of initial polymonads that are unionable and involve the identity polymonad can be unioned together to form one big unionable polymonad. Hence, if the individual polymonads in Haskell follow these restrictions there is no need to separate them.

The condition of being unionable and having the identity polymonad in common is mild, because, at least in the context of Haskell, each of the excluded

[11] Agda proof: `Haskell.Monad.Principal`
[12] Agda proof: `Haskell.Parameterized.Indexed.PhantomMonad`
[13] Agda proof: `Polymonad.Union`
[14] Agda proof: `Polymonad.Union.Unionable`
[15] Agda formalisation: `Polymonad.Unionable`

bind operations with type $(\mathsf{M},\mathsf{N}) \triangleright \mathsf{Id}$ resembles a function to run the computations of the polymonad without giving the additional arguments that are usually required to execute the side-effects modelled by it. As far as I am aware, the only polymonads that can provide such bind operations are the polymonads derived from the identity-monad (or monads isomorphic to it) and Haskell's `ST`-monad. The bind operation of the identity polymonad is allowed by Definition 4.9 and therefore the identity polymonad remains unionable. The `ST`-monad can still be made a unionable polymonad if it does not introduce bind operations of the form $(\mathsf{M},\mathsf{N}) \triangleright \mathsf{Id}$. Although the `ST`-monad is generalised by the `IO` monad the same problem does not arise there, because there is no safe way to compute a pure value from an `IO` computation. Even if the use of unsafe functions, such as `unsafePerformIO`, is considered, both, `ST` and `IO`, remain unionable polymonads if they do not introduce bind operations of the form $(\mathsf{M},\mathsf{N}) \triangleright \mathsf{Id}$.

The property of being unionable is important because, without it, the polymonad laws require bind operations that involve type constructors from $\mathscr{M}_1$ and $\mathscr{M}_2$ as well as the existence of bind operations with type $(\mathsf{M},\mathsf{N}) \triangleright \mathsf{Id}$. Many standard and generalised monads fail to meet that requirement.

Finally, it is important that the union of polymonads preserves principality, because principality is key to the implementation. The proof that the union of polymonads preserves principality makes two assumptions about the nature of subsets. These assumption are required, because the formalisation of the subset notion in Agda does not allow to inspect the subsets elements or partition the subset. Therefore, Proposition 4.13 makes the following two assumptions about subsets that are obviously true in set theory:

Assumption 4.11. Let $\mathscr{M}_1$ and $\mathscr{M}_2$ be sets of type constructors. If $F \subseteq (\mathscr{M}_{\mathsf{Id}} \cup \mathscr{M}_1 \cup \mathscr{M}_2)^2$ then one of the following statement will hold:

- $F \subseteq (\mathscr{M}_{\mathsf{Id}} \cup \mathscr{M}_1)^2$ or
- $F \subseteq (\mathscr{M}_{\mathsf{Id}} \cup \mathscr{M}_2)^2$ or
- there exists a pair in F that contains a type constructor from $\mathscr{M}_1$ and there also exists a pair in F that contains a type constructor from $\mathscr{M}_2$.

Assumption 4.12. Let $\mathscr{M}$ be a set of type constructors. If $F \subseteq (\mathscr{M}_{\mathsf{Id}} \cup \mathscr{M})^2$ then one of the following statement will hold:

- $F = \emptyset$ or
- $F = \{\ (\mathsf{Id},\mathsf{Id})\ \}$ or
- F contains at least one pair that is not equal to $(\mathsf{Id},\mathsf{Id})$.

Given that Assumption 4.11 and 4.12 hold the union of polymonads preserves principality[16]:

[16] Agda proof: `Polymonad.Union.Principal`

Proposition 4.13 (Polymonad union preserves principality)**.** Let $(\mathscr{M}_1, \Sigma_1)$ and $(\mathscr{M}_2, \Sigma_2)$ be polymonads with the same distinguished type constructor `Id` that are unionable and principal. Then their union $(\mathscr{M}_1 \cup \mathscr{M}_2, \Sigma_1 \cup \Sigma_2)$ is a principal polymonad as well.

4.5 Polymonad plugin for GHC

Programming with polymonads quickly leads to ambiguity errors. The inferred constraints contain type variables that do not appear in the overall type of expressions. The following example illustrates the problem:

```
test :: Identity Bool
test = Identity True >>= return
```

During type checking, GHC infers the constraints `Polymonad Identity m Identity` and `Polymonad Identity Identity m` for >>= and `return` respectively, and then the following overall type for `test`:

```
( Polymonad Identity m Identity
, Polymonad Identity Identity m
) => Identity Bool
```

Note the ambiguous type variable `m`: it appears in the constraints, but not in the type. There is thus no way to decide what type to instantiate it with in order to pick the right `Polymonad` instance. This is why programming with polymonads requires special support.

Another issue that may arise are overlapping instances. An example for this can be seen in the standard monad instances presented in Section 4.3.1. In case of the `Identity` type constructor several of these instances may match, meaning the plugin needs to choose one. The uniqueness of bind operations (Proposition 4.7) ensures that an instance may be chosen arbitrarily: the choice is inconsequential.

Both issues can be addressed during the constraint solving step of GHC's type checker. GHC supports a plugin interface to extend the constraint solver (Section 2.3). The polymonad plugin is implemented using this interface to aid GHC in solving `Polymonad` constraints.

The polymonad plugin is fully implemented[17], but there are still some examples for which issues arise. Those issues are discussed in Section 4.6. At present, the polymonad plugin requires GHC in version 7.10.

4.5.1 Structure

The plugin goes through the following steps when asked for assistance:

[17] GitHub: *jbracker/polymonad-plugin* - `https://github.com/jbracker/polymonad-plugin`

1. Identify the `Polymonad` class and `Identity` type constructor, because it cannot proceed without them.

2. If there are `Polymonad` constraints that do not contain any ambiguous type variables, choose an instance for them and provide their evidence.

3. Look at the `Polymonad` instances and constraints to invoke the detection algorithm and figure out which polymonads are currently available. There can be several polymonads involved in the constraints because local `do`-blocks with other polymonads may be nested within a `do`-block. The following steps are then applied to each polymonad and its associated constraints:

4. Try to simplify the constraints using the simplification rules presented in Hicks et al. [Hic+14]. This reduces the number of constraints to solve and also makes error messages more readable. If simplification made any progress, give control back to GHC. It can then try to solve the constraints using the newly available information. If GHC gets stuck again, it asks the plugin for further assistance.

5. If the simplification did not make any progress, invoke the solving algorithm that is based on the coherence proof of Hicks et al. [Hic+14].

4.5.2 Selection of bind operations

The simplification and solving algorithm presented by Hicks et al. [Hic+14] is based on the knowledge of the current polymonad. Since principal polymonads preserve principality under union, a detection algorithm to determine the correct set of bind operations for a polymonad is not necessary. However, such an algorithm may still be useful. First, it improves the overall efficiency of the plugin. Second, detecting the correct set of bind operations may also be necessary in the setting of future generalisations (Section 4.6).

Assumptions

The detection algorithm assumes that the basis (Definition 4.2) of a polymonad has no elements in common with any other polymonad basis except for `Id`. Further, it is also assumed that all type constructors in the basis of the polymonad in question are used in a `do`-block.

The latter may seem to be overly restrictive. However, typically, and in particular for all polymonads I have come across, the basis contains just a single element beside `Id`. Of course, the basis of the *union* of two polymonads would contain more than one element beside `Id` (under the assumption of no sharing), but that is not a problem: the detection algorithm simply returns one of the constituent polymonads rather than their union. If necessary, the programmer can

always translate a polymonad with a basis with more than one type constructor (beside Id) into a polymonad with a single one by careful indexing.

Algorithm

At the beginning of the detection process there are three sets to work with: the set of all `Polymonad` instances Σ_{Haskell} in Haskell, the set of given constraints C_{Given} that were assumed in type signatures, and the set of wanted constraints C_{Wanted} that GHC inferred and is not able to solve. C_{Given} also contains the derived constraints, which were derived during GHC's constraint solving. Hence, the derived constraints are treated as if they were given for the purposes of this algorithm.

To separate the wanted constraints for each polymonad from each other the algorithm constructs the following directed graph $G \stackrel{\text{def}}{=} (V, E)$:

$$V \stackrel{\text{def}}{=} \{\, \mathsf{K} \mid \mathsf{K} \in \{\, \mathsf{M}, \mathsf{N}, \mathsf{P} \mid (\mathsf{M}, \mathsf{N}) \rhd \mathsf{P} \in C_{\text{Wanted}} \,\}, \mathsf{K} \neq \mathsf{Id} \,\}$$

$$E \stackrel{\text{def}}{=} \{(\mathsf{K}, \mathsf{L}) \mid (\mathsf{K}, \mathsf{L}) \in \{\, (\mathsf{M}, \mathsf{P}), (\mathsf{N}, \mathsf{P}) \mid (\mathsf{M}, \mathsf{N}) \rhd \mathsf{P} \in C_{\text{Wanted}} \,\}, \mathsf{K} \neq \mathsf{Id}, \mathsf{L} \neq \mathsf{Id}\}$$

The nodes in each weakly connected component $G_i = (V_i, E_i)$ of G represent the set of unary type constructors that belong to the same polymonad with the exception of Id. Id is excluded from the graph as it is part of every polymonad and thereby would cause all of the components in G to be interconnected.

The process works because constraints belonging to different polymonads do not share type constructors, while constraints belonging to the same polymonad do. The monadic computations are composed through the bind operation and therefore the result of one bind operation is the input of another bind operation. The constraints are thus linked by common type constructors. The wanted constraints that belong to each determined polymonad are those that formed the edges in the weakly connected component.

The function `baseCons`, that is used in the following parts of the algorithm, takes a set of unary type constructors as argument and returns their basis. This function is required, because the unary type constructors in V_i are not limited to the indices they are applied to in the wanted constraints.

Now that the algorithm determined the type constructors $\texttt{baseCons}(V_i)$ and the wanted constraints C^i_{Wanted} of the involved polymonads, it has to determine which given constraints and instances belong to these polymonads. To do so, the algorithm goes through all of the available instances $\sigma \in \Sigma_{\text{Haskell}}$ and checks if there is a substitution of the type constructors in $\texttt{baseCons}(V_i) \cup \{\, \mathsf{Id} \,\}$ for the arguments in the head of σ such that the instance head and context is instantiated. If that is the case the algorithm keeps the instance for the current polymonad. The same process can be applied to the given constraints in C_{Given}.

At the end of this process the detection algorithm delivers a collection of polymonads $(\texttt{baseCons}(V_i) \cup \{\, \mathsf{Id} \,\}, C^i_{\text{Given}} \cup \Sigma^i_{\text{Haskell}})$ and their wanted constraints C^i_{Wanted}. The wanted constraints can then be simplified and solved using this information.

```
{-# LANGUAGE RebindableSyntax #-}
{-# OPTIONS_GHC -fplugin Control.Polymonad.Plugin #-}

module ExamplePolymonadModule where

import Control.Polymonad.Prelude
```

Figure 4.2: Example of a module header that enables the use of polymonads.

Correctness

Formally proving the correctness of this algorithm is complicated and remains future work. The key points for the correctness are stated in the remainder of this section.

The algorithm cannot collect more or fewer type constructors than belong to that polymonad due to the assumption that every do-block uses all type constructors in the basis of its corresponding polymonad.

Further, the assumption that polymonads do not share type constructors, ensures that all relevant instances are found.

4.5.3 Using the plugin

To use the polymonad implementation in Haskell, the programmer has to do three things:

1. Enable the GHC language extension `RebindableSyntax` (Section 2.2).
2. Import `Control.Polymonad.Prelude`. This module provides all the functionality of the standard `Prelude`, except that the standard monad functions are replaced with their corresponding polymonad versions.
3. Activate the polymonad type checker plugin by inserting the following line at the top of the module:

   ```
   {-# OPTIONS_GHC -fplugin Control.Polymonad.Plugin #-}
   ```

An example of a module that performs these steps is presented in Figure 4.2.

4.6 Limitations and conclusion

This chapter has explained how polymonads are able to remedy the inconvenience of manually specifying which monadic generalisation is used in the do-notation. Polymonads achieve this through the `Polymonad` type class in conjunction with a GHC plugin and the rebindable syntax language extension. The

`Polymonad` type class also allows writing functions that work for all polymonads and thus for all generalisations supported by them.

However, polymonads do not provide a complete solution. The "Polymonadic Programming" paper [Hic+14] already identifies a few limitations of the approach. In particular, it is assumed that the indices of a polymonad are phantom; i.e., that they do not influence the runtime behaviour of the polymonad. Lifting this restriction would be useful as that would allow for polymonads to support the full generality of what can be expressed by graded and indexed monads in Haskell. It may be possible to lift this restriction, although the authors of the previous papers and I suspect that non-phantom indices break principality. Consequently, the plugin may pick the wrong instances and produce programs yielding unintentional results if non-phantom indices are used. The constraint solving provided by the plugin would thus need to be changed and generalised. On the other hand, the detection and separation algorithm (or some variation of it) could turn out to be useful in a setting where principality does not hold more generally.

Another limitation of the theory of polymonads presented by Hicks et al. [Hic+14] is that it only allows constraints on the indices of type constructors, but not on the result types. Constrained monads thus do not fit into the present polymonad framework. The introduction of constrained polymonads in practice may be possible by using techniques similar to those used when introducing constrained monads, e.g., associated constraint synonyms. However, it remains future work to ensure that such an approach does not break the polymonad theory or the constraint solving process.

Furthermore, it is unclear how and if the natural generalisations of applicatives are supported by the polymonad theory.

Finally, GHC supports many Haskell extensions. Of relevance here are the extensions to the type system, such as type families. For polymonads, I have only focused on supporting those type system extensions that were needed for the examples, but I have yet to carry out comprehensive testing. At present, I am aware of one problem related to the production of evidence for some polymonad instances, leading to warnings in the core linting step of GHC. The discussion of the problem on the GHC mailing list led to the conclusion that the evidence produced by the plugin revealed a bug in the implementation of GHC. I have reported the issue in the GHC bug tracker[18].

However, all examples in the polymonad repository work as intended. When compiling modules using the polymonad plugin, I recommend enabling the option `-dcore-lint` to ensure that such errors do not get ignored silently.

Future work may be able to lift these limitations. However, I focused my research on an alternate approach: supermonads and superapplicatives. The reason for this shift is that polymonads are far more general than what is required to describe the monadic notions I found relevant when surveying the literature. As a result of this generality the theoretical and technical limitations I discuss

[18] `https://ghc.haskell.org/trac/ghc/ticket/11435`

above arise. In addition, polymonads have complex and unfamiliar laws when compared with monads and their generalisations. All of these reasons led me to explore supermonads and superapplicatives instead of pursuing polymonads further.

Chapter 5

Supermonads and superapplicatives

This chapter introduces supermonads and superapplicatives as an approach to unify the different generalisation discussed in Section 3. As discussed at the end of the last section, I discontinued my work on polymonads in favour of supermonads. I developed supermonads and superapplicatives with the requirements of the generalisations I aim to support in mind. Their representation is inspired by polymonads, but the approach to solve the arising issue of ambiguity is focused on simply reinstantiating type inference and constraint solving capabilities that were lost in the process of generalising the representation.

Section 5.1 discusses how supermonads and superapplicatives are encoded in Haskell and which problems arise from their representation. Examples of how different generalised monads and applicative can be made a supermonad or superapplicative are presented in Section 5.2. Section 5.3 gives a detailed account and explanation of how the plugin that extends Haskell to support supermonads and superapplicatives works. To show that the supermonad approach scales well and is simple to apply to larger examples, Section 5.4 presents two case studies. Finally, Section 5.5 discusses remaining limitations of the supermonad approach.

The source code of the supermonad implementation and its examples is available in a public Git repository[1].

5.1 Representation in Haskell

In Chapter 3 several generalisations of monads and applicatives are presented. Although each of these notions can be represented and used in Haskell, problems arise once they are used in conjunction with each other. Each notion has a separate type class. Therefore, standard library functionality has to be duplicated for each notion. In addition, using several notions side by side in the same module may be tedious and error prone, especially when it comes to the `do`-notation. Although it is possible to use the `do`-notion with different implementations of the bind and return operation, the presence of several different bind and return oper-

[1] GitHub: *jbracker/supermonad* - `https://github.com/jbracker/supermonad`

ations requires manual disambiguation for each do-block or the use of qualified names.

Supermonads and superapplicatives foster code reuse by obviating the need to give custom class definitions and adapted versions of the standard library functions for each monadic and applicative notion. In addition, supermonads and superapplicatives remove the need for manual disambiguation when working with more than one notion at a time.

In the following subsections, the general idea of how to represent each of the different generalisations with a fixed set of type classes is explored (Section 5.1.1). Then, Section 5.1.2 investigates why Haskell's type inference is insufficient when confronted with said set of type classes. Based on the insights of that investigation, Section 5.1.3 discusses how to remedy this shortcoming of the encoding. Then, Section 5.1.4 and 5.1.5 present the type classes in their current form with and without support for constrained monads and applicatives. Afterwards, Section 5.1.6 explains why the supermonad library opts for an additional encoding without support for constrained monads and applicatives. Finally, Section 5.1.7 discusses why superapplicatives are not a superclass of supermonads.

A categorical classification of supermonads and superapplicatives is deferred to Chapter 6. Until then, it suffices to be aware that the laws of supermonads and superapplicatives are exactly the same as for the notions that they intend to capture (Figure 3.2 and Figure 3.4); only the type of the involved variables changes.

5.1.1 Separate type classes for the bind, ap, and return operation

First, it is important to understand why each notion covered in Section 3.1 and 3.2 requires a separate type class in Haskell. The type classes of these generalisations follow this scheme:

```
class SomeNotion m where ...
```

Here `m` is the type constructor used throughout the monadic or applicative operations. It is impossible to give a single type class of this shape that captures all of the mentioned monadic notions, because the type constructor `m` has a different arity in each case and thus a different kind. However, the above type class only allows a type constructor of one specific kind and arity.

Solutions to this problem can be found in the work by Kmett[2,3] and the work on polymonads (Chapter 4). Both introduce a type class for the bind operation that is similar to the following:

```
class Bind m n p where
  (>>=) :: m a -> (a -> n b) -> p b
```

[2]Hackage: *monad-param* - `http://hackage.haskell.org/package/monad-param`

[3]*Parameterized Monads in Haskell (13. July 2007)* - `http://comonad.com/reader/2007/parameterized-monads-in-haskell/`

In this generalised type class, when defining an instance, `m`, `n`, and `p` may be partially applied versions of the type constructor in the instance head. Thus, the number of indices does not matter anymore, because after partial application all type constructors are unary.

This approach necessitates splitting the single monad type class into two, one for bind and one for return. The split is necessary because it would be unclear which of the three type constructors, `m`, `n`, or `p`, should be used for the return operation. It may even be the case that non of the partially applied constructors suits the return operation. Hence, the return operation is moved into the `Return` type class:

```
class Return m where
  return :: a -> m b
```

Note that, in contrast to the polymonad approach, the `Bind` type class is supposed to capture the single bind operation associated with a specific monadic notion. Hence, the return operation needs to be represented as a separate type class and cannot be encoded as another instance of `Bind`.

The same technique can be used to provide a type class for the ap operation:

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

Again, the pure operation now requires a separate type class. Since the pure and return operation coincide, the `Return` type class can be reused for this purpose and `pure` can be supplied as an alias of `return`:

```
pure :: (Return m) => a -> m a
pure = return
```

5.1.2 Insufficient type inference

Although these new type classes allow expressing all of the different monadic and applicative notions mentioned before, GHC's type inference does not suffice to resolve their constraints in many situations.

In Haskell 2010[4], type inference is guaranteed for almost all features of the language. However, to implement the `Bind` and `Applicative` class the language extension `MultiParamTypeClasses` is required. Since both classes have three distinct, unrelated arguments, GHC's type inference has no way of knowing that all constructors are intended to be partial applications of the same base constructor. Therefore, when inferring the type of an operation involving `Bind` or `Applicative`, GHC may consider some of the type constructors variables ambiguous.

[4] *Haskell 2010 Language Report* - `https://www.haskell.org/onlinereport/haskell2010`

Additionally, the separation of the operations into several different classes often means that it is unclear which `Return` instance to use, because the use of a bind or ap operation no longer determines a corresponding return or pure operation.

The following function illustrates the type inference problem with a simple example using the `Maybe` type:

```
plus3 :: Int -> Maybe Int
plus3 i = (Just 3) >>= \j -> return (i + j)
```

The function `plus3` adds three to a given integer and wraps the process into the `Maybe` monad. GHC's type inference infers the following type from the body of the function:

```
(Bind Maybe m Maybe, Return m) => Int -> Maybe Int
```

The first `Maybe` of the `Bind` constraint can be inferred from the expression `Just 3` and the second can be inferred through unification with the type signature. However, the type system has no clue as to which `Return` instance is meant and therefore infers the most general type possible: `m`. The inferred type `m` is ambiguous, because it does not occur on the right-hand side of =>. Therefore, the compiler aborts with an error message: there is no unambiguous way of instantiating `m` without jeopardising the runtime behaviour of `plus3`.

The case studies in Section 5.4 show that this issue is commonplace in programs involving the above generalised representation. In addition, the case studies show that if the monadic notion is parameterised, the type of the indices cannot be inferred either. The indices were previously inferred through unification with the type signature of the bind or return operation but that is not possible anymore, because the ambiguity prevents choosing an appropriate instance on which to base that inference.

5.1.3 Enhancing type inference and constraint solving.

To address the insufficient type inference capabilities for the `Bind`, `Applicative`, and `Return` constraints, Kmett adds a functional dependency and a specialised version of the return operation that always operates on the `Identity` monad:

```
return :: a -> Identity a
return a = Identity a

class Return m where
  returnM :: a -> m a

class (Functor m, Functor n, Functor p) =>
      Bind m n p | m n -> p where
  (>>=) :: m a -> (a -> n b) -> p b
```

The correct choice of return operation restores type inference in some, but not all, cases.

To see how well Kmett's approach works a retrofitted version of the first case study (Section 5.4.1) uses his library[5]. Many situations remain that require manual type annotations to solve ambiguous variables. In addition, the correct return operation (`return` or `returnM`) also had to be chosen depending on the context. Both of these tasks are tedious. What is actually required, instead of a functional dependency, is the ability to deduce any two of the type constructors from the third, in any order. Adding more functional dependencies to address this issue would quickly become so restrictive that the type class would only capture standard monads.

This discussion shows that the following capabilities are lost by introducing the new type classes:

- By moving the bind, ap, and return operations into separate type classes, the connection between the operations is not visible for the compiler anymore. Hence, the compiler does not know which bind, ap, and return operations belong to the same monad or applicative anymore.
- The knowledge that all three type constructors in the `Bind` or `Applicative` class are partial applications of the same type constructor is also lost.
- Finally, the ability to infer the indices through unification with the different operation type signatures is lost, because this is only possible if the specific instance in use is known.

To my knowledge, it is only possible to address these issues inside Haskell itself by manually adding annotations to guide the type inference. The case studies in Section 5.4 show that providing these annotations is a tedious and extensive task. Therefore, I introduce the unifying notions of *supermonad* and *superapplicative* as a *language extension* implemented as a GHC plugin.

5.1.4 The supermonad and superapplicative type classes

The notion of a supermonad is embodied by the `Bind` and `Return` classes as presented above. Superapplicatives are represented through the `Applicative` and `Return` type classes. With these type classes in place the type system needs to be extended by incorporating knowledge about supermonads and superapplicatives. Concretely, this is realised by teaching GHC's type checker about the new classes and their underlying assumptions. GHC offers a plugin mechanism which is well suited to this end (see Section 2.3). The goal of the plugin is to allow GHC to infer the types of any supermonad or superapplicative computation that it would have been able to infer if that computation was using one of the specialised type classes from Section 3.1 or 3.2. Thus, any type inference capabilities lost in the process of generalisation are restored.

[5]Hackage: *monad-param* - `http://hackage.haskell.org/package/monad-param`

The new classes, `Bind`, `Applicative`, and `Return`, are declared as follows:

```
class (Functor m, Functor n, Functor p) =>
      Bind m n p where
  type BindCts m n p :: Constraint
  type BindCts m n p = ()
  (>>=) :: (BindCts m n p)
        => m a -> (a -> n b) -> p b

class (Functor m, Functor n, Functor p) =>
      Applicative m n p where
  type ApplicativeCts m n p :: Constraint
  type ApplicativeCts m n p = ()
  (<*>) :: (ApplicativeCts m n p)
        => m (a -> b) -> n a -> p b

class (Functor m) => Return m where
  type ReturnCts m :: Constraint
  type ReturnCts m = ()
  return :: (ReturnCts m) => a -> m a
```

Generalising from standard monads and applicatives, `Functor` constraints on each of the partially applied type constructors are also introduced. As discussed in Section 3.1 and 3.2 each notion has an associated functor or family of functors. The `Functor` constraints cannot guarantee that all of the necessary functor instances actually exist, but they at least hint at their necessity.

The associated type synonyms `BindCts`, `ApplicativeCts`, and `ReturnCts` are added to allow for custom constraints on the *indices* of the type constructors. These constraints are especially important to support graded monads and applicatives. By default all associated constraints are empty to ease instantiation. Due to the default empty constraint, programmers only need to implement custom constraints when these are actually required for an instance.

The above type classes do not support constraints on the *result types* and thus cannot be instantiated for constrained monads and applicatives. The integration of result type constraints is discussed in the following section. Their integration is simple and does not require changes to the plugin, but there are some practical implications.

Not all instantiations of the type classes constitute valid supermonads or superapplicatives. Therefore, some contextual constraints that give guidance and ensure that the type checker plugin can restore type inference are imposed:

- For every base constructor of a supermonad or superapplicative there is exactly one `Return` instance and exactly one `Bind` or `Applicative` instance (or both).
- The constructors of a `Bind` or `Applicative` instance are all partial applications of the same base constructor.

These contextual constraints are enforced by the plugin. The programmer is still required to ensure that their instances follow the monad and applicative laws and that all associated functor instances exist.

As presented in "Supermonads: One Notion to Bind Them All" [BN16] the standard monad laws can be generalised to suite supermonads and give guidance as to which instances are valid. As a guideline, every supermonad should fulfil the standard monad laws with the types of the involved variables adjusted to fit the generalised notion. The paper [BN16] gives more insight as to how those laws look when specified with the full generality of supermonads in mind. For superapplicatives the same should be true: The standard applicative laws (adjusted for the generalised notion) should always be fulfilled. But in the case of superapplicatives the laws still have to be explored in future work.

It could be argued that it is unfortunate that a couple of classes have been imbued with special meaning, as opposed to the relevant constraints being stated manifestly in the source code. However, the approach taken by supermonads is not without precedent. For example, the deriving mechanism is, in its basic form, limited to a handful of classes with meaning known to the compiler, and a situation where additional instances may invalidate contextual constraints occurs also for language extensions such as overlapping instances. There may be different ways to realise a generalised notion or some essentially equivalent notion in the future. What this work establishes is that there is at least one practical way of integrating a unified representation of monadic and applicative notions into Haskell.

5.1.5 Adding constraints

To support constrained monads and applicatives, the `Bind`, `Applicative`, and `Return` class need to be made even more general by adding the result types `a` and `b` as additional arguments to the associated constraints.

```
class (Functor m, Functor n, Functor p) =>
      Bind m n p where
  type BindCts m n p (a :: *) (b :: *) :: Constraint
  type BindCts m n p a b = ()
  (>>=) :: (BindCts m n p a b)
        => m a -> (a -> n b) -> p b

class (Functor m, Functor n, Functor p) =>
      Applicative m n p where
  type ApplicativeCts m n p (a :: *) (b :: *) :: Constraint
  type ApplicativeCts m n p a b = ()
  (<*>) :: (ApplicativeCts m n p a b)
        => m (a -> b) -> n a -> p b

class (Functor m) => Return m where
```

```
  type ReturnCts m (a :: *) :: Constraint
  type ReturnCts m a = ()
  return :: (ReturnCts m a) => a -> m a
```

Without the constrained notions the `Functor` type class from the standard library can be reused, but constrained monads and applicatives also require constrained functors. Therefore, a replacement for the standard functor class, that allows constraints on the result types, is required.

```
class Functor f where
  type FunctorCts f (a :: *) (b :: *) :: Constraint
  type FunctorCts f a b = ()
  fmap :: (FunctorCts f a b) => (a -> b) -> f a -> f b
```

5.1.6 Practical implications of associated constraints

The associated constraints of each new type class have some practical implications, especially if they include constraints on the result types.

The evaluation of associated type synonyms is only possible if all arguments are known. This limitation becomes an issue when writing code that is polymorphic in the used supermonad. Type checking such polymorphic code is only possible if all of the involved `BindCts`, `ApplicativeCts` and `ReturnCts` are listed in the type signature, because they cannot be evaluated to determine the constraints they refer to. For example:

```
liftM2 :: ( Bind m p p, Bind n p p
          , BindCts m p p a c, BindCts n p p b c
          , Return p, ReturnCts p c)
       => (a -> b -> c) -> m a -> n b -> p c
liftM2 f ma nb = do
  a <- ma
  b <- nb
  return (f a b)
```

This is a simple conversion of the `liftM2` function from the `base` library to allow for supermonads with constrained result types. The programmer needs to list `BindCts` constraints for all the possible result types of the bind operations that occur in the function body. Especially for long polymorphic functions this may become irritating quickly. This may still be an issue even without constraints on the result types, as all of the different bind operations that are involved in the function need to be listed in the constraints.

This problem is exacerbated for classes and instances that are polymorphic in the used supermonad. For example, consider a naive adaptation of `MonadPlus` for unconstrained supermonads:

```
class (Bind d d d, Return d) => MonadPlus d where
  mzero :: (BindCts d d d, ReturnCts d) => d a
```

```
mplus :: (BindCts d d d, ReturnCts d)
      => d a -> d a -> d a
```

During the class definition it is unclear if the given constraints are sufficient to implement the member functions in a given instance. It is particularly unclear which constrained result types will be involved if the above definition is extended for constrained monads. Therefore, custom individual constraints need to be allowed for every function in the class:

```
class (Bind d d d, Return d) => MonadPlus d where
  type MZeroCts d :: * -> Constraint
  type MPlusCts d :: * -> Constraint
  mzero :: (MZeroCts d a) => d a
  mplus :: (MPlusCts d a) => d a -> d a -> d a
```

An instance could then look as follows:

```
instance MonadPlus M where
  type MZeroCts M a = ( BindCts M M M a String
                      , ReturnCts M a, ...)
  type MPlusCts M a = ( BindCts M M M Int a
                      , FunctorCts M a Bool, ...)
  mzero = ...
  mplus = ...
```

The process of adding these constraints and associated type synonyms is mechanical and it should be possible to automate it provided the right technology, but without automation this process remains error prone and tedious.

The described problems vanish in code that is not polymorphic in the used supermonad or superapplicative. If the code is not polymorphic, all arguments to `BindCts`, `ApplicativeCts`, and `ReturnCts` are known, meaning they can be evaluated and the constraints they produce checked.

To mitigate the issues related to polymorphic functions and classes, the supermonad library offers two representations of supermonads and superapplicatives: one that supports constraints on the result types and one that does not. Thus the programmer is given a choice: if constraints on result types are needed, this is possible, but a bit of extra care is required; if not, programming can be streamlined by opting for the version without constraints on the result types.

This approach may lead to code duplication, because libraries that support one notion need to be copied and adjusted to also suite the other notion. However, I think the benefit of allowing programmers to work with supermonads in a more convenient fashion to gain experience with them, especially while they are new, outweighs this disadvantage.

Another implication arose while implementing the standard library functions with the new type classes. Type signatures that involve only two type constructors may have different sets of constraints. For example,

```
void :: ( Bind m n n, BindCts m n n a ()
        , Return n, ReturnCts n () )
     => m a -> n ()
void ma = ma >>= (\_ -> return ())
```

can also be written as

```
void :: ( Bind m m n, BindCts m m n a ()
        , Return m, ReturnCts m () )
     => m a -> n ()
void ma = ma >>= (\_ -> return ())
```

This means that sometimes there is a choice of which constraints to use and there is no apparent advantage or disadvantage to either set of constraints. The supermonad library makes an effort to be systematic when it comes to this kind of choice. The library always repeats the second type constructor if this kind of choice occurs for a function, e.g., it uses `Bind m n n` instead of `Bind m m n`.

5.1.7 Why `Applicative` is not a superclass of `Bind`

Note that the `Applicative` class is not a superclass of `Bind`. The reason for this is that the representation of applicatives in Haskell is different (but equivalent) to their representation in category theory (see Section 6.3). In Haskell two operations fully characterise an applicative:

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

In contrast, in category theory an applicative is usually modelled as a lax monoidal functor (see Section 6.1.4). Translating the categorical representation to Haskell results in the following set of operations:

```
tensor :: f a -> f b -> f (a , b)
unit   :: f ()
fmap   :: (a -> b) -> f a -> f b
```

Without constraints on the result types, both representations are equivalent. However, constraints on the result types can make one representation more useful than the other depending on the use case.

For example, each of these functions can be implemented for the `Set` data type introduced in Section 3.1.4.

```
(<*>) :: (Ord b) => Set (a -> b) -> Set a -> Set b
ff <*> fa = S.foldr (\f fb -> S.union (S.map f fa) fb)
                    S.empty ff

pure :: a -> Set a
pure = S.singleton
```

```
tensor :: (Ord a, Ord b) => Set a -> Set b -> Set (a , b)
tensor fa fb = S.foldr unionMap S.empty fa
  where
    unionMap a fab = S.union (S.map (\b -> (a,b)) fb) fab

unit :: Set ()
unit = S.singleton ()

fmap :: (Ord b) => (a -> b) -> (Set a -> Set b)
fmap = S.map
```

The above <*> operator provides a valid `Applicative` instance for `Set`. However, notice that one of the arguments (`Set (a -> b)`) is a set of functions. In Haskell, it is not possible to define a total ordering on general functions (`a -> b`), at least not in the form that the `Ord` class requires. The only way to create a set of functions is thus by either creating an empty set or by creating a singleton set with exactly one function, because all of the other functions to create or combine `Set` values require an ordering on the elements. Therefore, the practical relevance of <*> is limited in the context of `Set`.

On the other hand, the interface consisting of `tensor`, `unit` and `fmap` does not have the above limitation. Thus, for `Set`, a representation of applicatives as lax monoidal functors in Haskell would be more useful and practical than the classical representation.

However, when looking at other examples, e.g., parser combinators, the classical representation with <*> is useful and practical in contrast to the lax monoidal functor representation [SD96; MP08].

Putting both interfaces into one type class is problematic for two reasons. First, combining the interfaces forces the user to always implement both interfaces, although this may not be possible or useful depending on the involved constraints. Second, functions defined in terms of a combined interface may use any of the above functions and therefore are only useful in certain contexts without this being immediately obvious to the caller.

Therefore, the supermonad library gives the user the freedom not to implement the applicative interface by not making it a superclass of `Bind`. This also opens the possibility to offer the lax monoidal representation as an optional alternative to the `Applicative` class at some point in the future.

5.2 Examples of supermonads and superapplicatives

Section 3.1 and 3.2 present the monadic and applicative notions that the supermonad approach aims to support. This section demonstrates how the motivating examples from the above sections instantiate supermonads and superapplicatives.

5.2.1 Maybe and the state transformer: a standard monad

A standard monad, e.g., Maybe, can instantiate the supermonad and superapplicative interface as follows:

```
instance Return Maybe where
  return = P.return

instance Applicative Maybe Maybe Maybe where
  (<*>) = (P.<*>)

instance Bind Maybe Maybe Maybe where
  (>>=) = (P.>>=)
```

The qualifier P refers to the standard Prelude. The instances were implemented directly in terms of the original Maybe-monad above. The Return instance together with the Applicative instance forms the superapplicative and the Return instance together with the Bind instance forms the supermonad.

Unfortunately, a direct implementation in terms of the original notion does not always work: if a standard monad is defined in terms of another monad, e.g., a monad transformer, it is necessary to reimplement the bind and return function to ensure that the nested monadic notion is also a supermonad. For example, the StateT monad transformer is implemented as follows:

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

instance (Return m) => Return (StateT s m) where
  type ReturnCts (StateT s m) = ReturnCts m
  return x = StateT ( \s -> return (x, s) )

instance ( Bind m n p ) =>
         Bind (StateT s m) (StateT s n) (StateT s p) where
  type BindCts (StateT s m) (StateT s n) (StateT s p)
    = (BindCts m n p)
  m >>= k = StateT ( \s -> runStateT m s >>=
                    \(a, s') -> runStateT (k a) s' )
```

The constraints have to be defined using BindCts and ReturnCts, which could be left empty in the previous example. These constraints ensure that the bind and return operation of the nested monad exist. The state monad transformer is also generalised to allow for any kind of nested supermonad by using the three separate type constructors m, n and p instead of the same. For brevity, the Applicative instance is omitted.

Other monad based functions, such as lift, get, or put, can be used in supermonad computations. However, due to their Monad constraint they are only compatible with supermonads that are also standard monads and hence also have

to be rewritten with supermonads in mind for a full integration. Reimplementing these functions for supermonads is unproblematic, but generalising their type class abstractions to use supermonads has the practical implications discussed in Section 5.1.6.

5.2.2 Fixed-length vectors: a graded monad

The `Vector` data type from the introduction of graded monads in Section 3.1.3 provides a good demonstration of how a graded monad instantiates the supermonad interface. For comparison the graded monad type class and instances provided by the `effect-monad` package[6] are presented here. The supermonad is then implemented in terms of the representation from that package to highlight the changes when transitioning from one representation to the other.

Before implementing the instances for the `Vector` data type, type-level natural numbers and the associated type-level function to multiply them need to be introduced:

```
data Nat = Z | S Nat

type family Mult (n :: Nat) (m :: Nat) :: Nat where ...
```

Note that `Nat` is lifted from the type/value level to the kind/type level via the language extension `DataKinds` (Section 2.2). The `Vector` data type provides the following functions:

```
data Vector (n :: Nat) a where ...

map :: (a -> b) -> Vector n a -> Vector n b
map f v = ...

concatMap :: Vector n a
          -> (a -> Vector m b)
          -> Vector (Mult n m) b
concatMap v f = ...

singleton :: a -> Vector ('S 'Z) a
singleton a = ...
```

The quotes in front of constructors, e.g., `'S`, are GHC notation to lift a value-level constructor to the type level. The details of the implementation are omitted for brevity[7].

The `effect-monad` package provides the following type class for graded monads which is virtually the same as the `GradedMonad` type class from Section 3.1.3:

[6] Hackage: *effect-monad* - `http://hackage.haskell.org/package/effect-monad`

[7] *Implementation of* `Vector` - `https://github.com/jbracker/supermonad/blob/master/examples/monad/effect/Vector.hs`

```
class Effect (m :: k -> * -> *) where
  type Unit m :: k
  type Plus m (i :: k) (j :: k) :: k
  type Inv m (i :: k) (j :: k) :: Constraint
  type Inv m i j = ()

  return :: a -> m (Unit m) a
  (>>=)  :: (Inv m i j)
         => m i a -> (a -> m j b) -> m (Plus m i j) b
```

The associated type synonyms `Unit` and `Plus` together with the kind variable `k` represent the type-level monoid of the graded monad. The variable `k` is the carrier of the monoid, `Unit` provides the neutral element and `Plus` defines the binary operation to combine two elements. The constraints specified with `Inv` are necessary to ensure that the monoid elements have all properties necessary to perform the `Plus` operation. Given this interface, an instance of `Effect` can be provided for `Vector`:

```
instance Effect Vector where
  type Unit Vector = 'S 'Z
  type Plus Vector n m = Mult n m
  type Inv Vector n m = ()

  return = singleton
  (>>=) = concatMap
```

The supermonad can now be given in terms of the functions provided by the `Effect` type class (qualified with `E` to prevent name clashes):

```
instance Functor (Vector n) where
  fmap f xs = map f xs

instance ( nm ~ Plus Vector n m ) =>
         Bind (Vector n) (Vector m) (Vector nm)
         where
  type BindCts (Vector n) (Vector m) (Vector nm)
          = Inv Vector n m
  (>>=) = (E.>>=)

instance Return (Vector ('S 'Z)) where
  return = E.return

instance ( nm ~ Plus Vector n m ) =>
         Applicative (Vector n) (Vector m) (Vector nm)
         where
  type ApplicativeCts (Vector n) (Vector m) (Vector nm)
          = Inv Vector n m
  mf <*> ma = mf E.>>= \f -> fmap f ma
```

Again, the original implementation of bind and `return` can be reused without alteration. Note that it is not possible to replace `nm` with `Plus Vector n m` in the instance arguments, because GHC does not allow type synonym applications in the instance arguments. In this example the `BindCts` are used to represent the `Inv` constraints. Also notice that it is simple to provide the `Applicative` instance for `Vector` based on the `Bind` instance. The `effect-monad` package does not provide an interface for graded applicatives.

As mentioned in the beginning, the `Bind`, `Applicative` and `Return` instances were implemented in terms of the interface provided by the `Effect` instance. This, more abstract implementation, demonstrates the transition to the supermonad representation and provides a guide how any graded monad can become a supermonad or superapplicative.

5.2.3 Session types: an indexed monad

As an example involving indexed monads I chose to instantiate supermonad instances for the `Session` indexed monad from the `simple-sessions` package[8]. The motivational example to introduce indexed monads in Section 3.1.2 gave a brief introduction to the `Session` type. The `Session` monad implements session types and uses the implementation of indexed monads provided by the `indexed` package[9]. The `indexed` package provides a complete type class hierarchy for indexed monads and applicatives.

```
class IxFunctor f where
  imap :: (a -> b) -> f j k a -> f j k b

class IxFunctor m => IxPointed m where
  ireturn :: a -> m i i a

class IxPointed m => IxApplicative m where
  iap :: m i j (a -> b) -> m j k a -> m i k b

class IxApplicative m => IxMonad m where
  ibind :: (a -> m j k b) -> m i j a -> m i k b
```

The `Session` type has instances for all of these classes. The supermonad instances for this notion can be given by partially applying the `Session` type constructor in the instance head and reusing the existing instances.

```
instance Functor (Session i j) where
  fmap = imap
```

[8]Hackage: *simple-sessions* - `http://hackage.haskell.org/package/simple-sessions`

[9]Hackage: *indexed* - `http://hackage.haskell.org/package/indexed`

```
instance Applicative (Session i j)
                     (Session j k)
                     (Session i k) where
  (<*>) = iap

instance Bind (Session i j)
              (Session j k)
              (Session i k) where
  ma >>= f = ibind f ma

instance Return (Session i i) where
  return = ireturn
```

The implementation of the supermonad and superapplicative instances for `Session` are simple and similar to those of the graded monad example. Note that in this example the instances are concrete for the `Session` data type instead of abstract for any `IxMonad`, `IxApplicative`, or `IxFunctor`.

5.2.4 The `Set` constrained monad

To give an example of a constrained monad, I implement the supermonad instances for the `Set` data type from the introductory example in Section 3.1.4. These instances need to use a constrained monad for `Set`, because many of the operations involving `Set` require an ordering constraint (`Ord`) on the elements. The module `Data.Set`[10] that provides the implementation and functions is referred to as `S` in the following instances.

```
instance Functor Set where
  type FunctorCts Set a b = (Ord b)
  fmap = S.map

instance Bind Set Set Set where
  type BindCts Set Set Set a b = (Ord b)
  s >>= f = S.unions (S.map f s)

instance Return Set where
  return = S.singleton
```

Both, the `Functor` and the `Bind` instance require an `Ord` constraint on `b`. In this case, neither requires any constraints on `a`, and no constraints are needed for the `Return` instance either, because the `singleton` function works for any type. In general, however, all constraints may be needed. The `Applicative` instance is omitted, because, as discussed in Section 5.1.6, it is of limited utility for `Set`.

[10] Hackage: *containers* - `http://hackage.haskell.org/package/containers`

5.2.5 Tracking resources: an application of graded applicatives and indexed monads

Section 3.2 has shown that applicatives can be generalised in the same way as monads. As an illustration, this section presents an example where a graded applicative is used in conjunction with an indexed monad (and applicative) to track the maximal resource usage for a set of concurrent processes. One application is deadlock avoidance using the Banker's algorithm [Dijnd].

The Banker's algorithm can be used with a fixed set of different kinds of resources, where each only provides a certain number of instances. If resources are allocated to processes as they are requested, solely based on availability, processes can easily deadlock. This can be avoided if allocation is subject to approval by the Banker's algorithm. The idea is that each process declares its maximal resource need up-front. A request to allocate resources is then only approved if, assuming the request is granted, it would still be possible to run all processes to completion under the worst case scenario of all processes simultaneously requesting the remainder of their needs.

Of course, this only works if the stated maximal resource need is an upper bound on the actual resource need for each process. If the resource needs have to be calculated manually, and then manually kept up-to-date as the code evolves, simple mistakes may arise in the process. However, if the maximal resource need is part of the type of a process and thus checked automatically, at compile time, this problem can be avoided.

This example illustrates how this can be done for processes expressed monadically using an indexed monad, where the pre-state is the pair of the maximal resource need and resources held before the monadic action, and the post-state is a pair of the maximal resource need and the resources held after the action. The processes are then lifted into a graded applicative, where the index corresponds to the maximal resource need, allowing processes to be run concurrently subject to their resource requests being granted.

First, type-level functions to compute the maximum and to check the order of type-level naturals are required:

```
type family Max (n :: Nat) (m :: Nat) :: Nat where ...
type family Leq (n :: Nat) (m :: Nat) :: Bool where ...
```

The type-level natural numbers `Nat` are reused from the graded monad example in Section 5.2.2.

For this example, it is assumed that there are two kinds of resources, `A` and `B`. For each process, the monad needs to keep track of the current maximal use and the current allocation for each kind of resource, that is, four quantities:

```
data ProcRes = ProcRes Nat Nat Nat Nat
```

The convention is that the first two represent the maximum and current allocation for resource `A`, and the last two the corresponding quantities for resource `B`.

The monadic representation of processes is indexed with this state, with the first index representing the pre-state and the second representing the post-state:

```
data Proc (i :: ProcRes) (j :: ProcRes) a = Proc ...
```

Further it is assumed that monadic actions to claim and release a resource of a specific kind, with the type indices tracking the changes in the maximal resource use and current allocation, exist. The claim operations also return an identifier for the allocated resource instance:

```
type ResId = Int

claimA :: Proc ('ProcRes ma ra mb rb)
               ('ProcRes (Max ma ('S ra)) ('S ra) mb rb)
               ResId

releaseA :: ResId -> Proc ('ProcRes ma ('S ra) mb rb)
                          ('ProcRes ma ra mb rb)
                          ()

claimB :: Proc ('ProcRes ma ra mb rb)
               ('ProcRes ma ra (Max mb ('S rb)) ('S rb))
               ResId

releaseB :: ResId -> Proc ('ProcRes ma ra mb ('S rb))
                          ('ProcRes ma ra mb rb)
                          ()
```

The `Functor`, `Return`, `Applicative`, and `Bind` instances for `Proc` have the following outline:

```
instance Functor (Proc i j) where
  fmap f p = ...

instance Return (Proc i i) where ...
  return a = ...

instance Applicative (Proc i j) (Proc j k) (Proc i k)
    where ...
  pf <*> pa = ...

instance Bind (Proc i j) (Proc j k) (Proc i k)
    where ...
  pa >>= f = ...
```

Note that `Proc` forms an indexed applicative as well as an indexed monad, but this example is only concerned with the monadic interface.

The next step is to introduce the notion of an *executable* process constituting a (possible) unit to be scheduled. This example only provides an applicative interface, as the objective is to run these concurrently to the extent desirable, subject to resource allocation constraints being fulfilled. The idea for this applicative interface is inspired by the concurrent query language presented by Marlow et al. [Mar+14]. It is indexed by the maximal resource need and forms a graded applicative as the combined need of two executable processes is the maximum of the constituents' needs (the representation of the need and the maximum operation forms a monoid). Whether to schedule a single unit, with the constituents running one after the other, or to schedule the constituents concurrently is for the scheduler to decide, and this example does not concern itself further with it. The combined need is the maximum either way. The needs of whichever units end up being scheduled concurrently are what is communicated to the Banker's algorithm through an underlying scheduler state that also keeps track of current allocations for each executable process:

```
data MaxRes = MaxRes Nat Nat

data Exec (i :: MaxRes) a = Exec ...

instance Functor (Exec i) where
  fmap f e = ...

instance Return (Exec ('MaxRes 'Z 'Z)) where ...
  return a = ...

instance ( 'MaxRes ma1 mb1 ~ m1, 'MaxRes ma2 mb2 ~ m2
         , Max ma1 ma2 ~ maR, Max mb1 mb2 ~ mbR
         , 'MaxRes maR mbR ~ mR) =>
         Applicative (Exec m1) (Exec m2) (Exec mR) where
  ef <*> ea = ...
```

The function for lifting a process to an executable process has the following signature. It reflects the requirements that a process starts without any allocated resources and must free all allocated resources before terminating:

```
process :: Proc ('ProcRes 'Z 'Z 'Z 'Z)
                ('ProcRes ma 'Z mb 'Z) a
        -> Exec ('MaxRes ma mb) a
```

Finally, the function for running an executable process, which potentially spawns a number of concurrent processes, given specific availability for each kind of resource, has the following signature:

```
run :: ( ToNatV ma', ToNatV mb'
       , Leq ma' ma ~ 'True, Leq mb' mb ~ 'True)
    => NatV ma -> NatV mb
    -> Exec ('MaxRes ma' mb') a -> IO a
run ma mb ea = ...
```

Note that `run` needs to ensure that the available resources of each kind suffice to satisfy the maximal need of any one process, which is reflected by the overall maximal need. The type `NatV` and the class `ToNatV` mediate between value-level and type-level naturals, specifically allowing resource availability to be stated as arguments to `run` as well as reflecting maximal needs (`ma'` and `mb'`) back to the value level to enable Banker's algorithm to only grant claims when it is safe to do so.

5.3 Implementation and use of the GHC plugin

As explained in Section 5.1.2, by splitting the bind, ap, and return operations into different classes, and by allowing the use of different partially applied type constructors, the direct connection between the operations and the type constructors has been broken. This introduces ambiguities. This section explains how the supermonad plugin for GHC resolves these ambiguities and aids type inference for the `Bind`, `Applicative`, and `Return` type class by exploiting knowledge about supermonads and additional contextual constraints. As a result, the plugin is able to infer the type of the any monadic or applicative notion from Section 3.1 or 3.2 that is encoded with supermonads as if they were written with their corresponding specialised type classes.

At the time of writing, the supermonad plugin has been tested using GHC 7.10.3, 8.0.1, 8.2.1, and 8.4.2. It will definitely not work with versions of GHC lower than 7.10, because the plugin infrastructure was still under development before that version.

5.3.1 Implementation and structure

The type inference capabilities the plugin needs to support are:

- Restore the connection between the bind, ap, and return operation for each supermonad or superapplicative.
- Enforce and use the knowledge that all three type constructors in the head of `Bind` or `Applicative` instances are partial applications of the same base constructor.
- Infer the indices of partially applied type constructors through unification with the type signature of the bind, ap, or return operation that involves them.

During the explanation of the algorithm, cases are distinguished based on the base constructors that are found in the supermonad constraints. Therefore, base constructors that are not type variables are referred to as *manifest constructors*, base constructors that are ambiguous type variables are referred to as *ambiguous constructors*, and base constructors that are unambiguous type variables are referred to as *variable constructors*.

Note that the plugin cannot enforce that the `Bind`, `Applicative`, and `Return` instances actually form valid supermonads or superapplicatives according to the categorical model (Section 6). This especially means that the plugin does not check or enforce that the equational laws associated with each notion are satisfied. The users still have to confirm the laws and correctness of their instances for themselves. The plugin only enforces side-conditions that are relevant to the solving process, as is described in the following sections.

Assumptions

The plugin assumes that a supermonad or superapplicative in Haskell consists of exactly one `Bind` (and/or `Applicative`) and one `Return` instance. This assumption is true for all of the monadic and applicative notions introduced in Section 3.

Since it is not possible to enforce the above assumption directly in Haskell, the plugin checks that each base constructor has exactly one `Return`, `Bind`, and `Applicative` instance. However, if a `Bind` instance exists the corresponding `Applicative` instance is optional and vice versa. The plugin then also checks that the arguments of the instances are applications of the same base constructor. If all instances conform with this assumption, the plugin creates an association between each base constructor and its single `Bind`, `Applicative`, and `Return` instance to enable a quick lookup of the appropriate instances for a given base constructor.

Another assumption is that a monadic and applicative computation only ever involves a single supermonad or superapplicative. For example, it is not allowed to use several different supermonads within one `do`-block. This does not prohibit nesting of computations: it just means that lifting monadic or applicative computations into each other needs to be stated explicitly.

Algorithm

After checking these contextual constraints, the actual solving algorithm is executed. The algorithm is composed of the following steps:

1. Construct a graph that connects two wanted supermonad or superapplicative constraints if and only if they share an ambiguous constructor. Each connected component of the constructed graph is called a *constraint group*.

2. For each constraint group, solve the ambiguous constructors:

- If the group involves only one manifest and no variable constructors, all ambiguous constructors are set to that manifest constructor.
- If the group involves no manifest constructor and at least one variable constructor, all possible associations between the ambiguous and the variable constructors are checked. If there is exactly one satisfiable association, use it. Otherwise, abort.
- In any other case abort.

3. Check each solved constraint for ambiguous indices. If such indices are found, unify the constraint that contains them with the associated instance of the used base constructor and thereby solve the ambiguous indices.

Explanation of Step 1

The constraints of a program fragment may involve constraints from different computations or `do`-blocks. Therefore, the separation of constraints into groups is necessary to ensure that the constraints that are being solved together belong to the same computation. To achieve this the plugin groups them by overlapping ambiguous constructors. These constructors can only overlap between two constraints if they are actually used within the same computation. Not capturing all constraints from a specific computation is not a problem: smaller groups can always be solved separately. If this solving process leads to a conflict, e.g., both groups use a different manifest constructor, then GHC will notice this when conflicting results are produced by the plugin.

For example:

```
f = do a <- [1,2,5]                          -- 1
       b <- maybeToList ( do                 -- 2
         c <- return (a == 1)                -- 3
         if c then return 1 else Nothing )   -- 4
       return (a + b)                        -- 5
```

The monadic computation in `f` uses lists and a nested computation with `Maybe`. The constraints inferred from `f` are:

```
1: Bind [] m n            3: Bind s t Maybe
2: Bind [] p m            3: Return s
5: Return p               4: Return t
```

The constraints involve the five ambiguous constructors `m`, `n`, `p`, `s`, and `t`. Figure 5.1 shows the graph produced by the separation step. Within the graph the three constraints on the left form one connected component and the three constraints on the right form another. These connected components reflect exactly the outer list computation and the nested `Maybe` computation, respectively.

Note that a constraint group may also contain `Applicative` constraints alongside `Bind` and `Return` constraints.

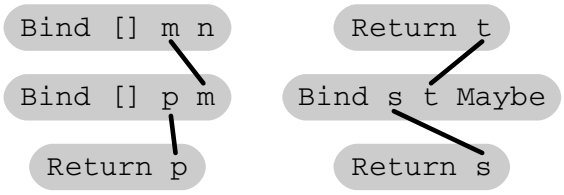


Figure 5.1: Graph produced by the separation algorithm from the example.

Explanation of Step 2

If only one manifest and no variable constructors are within the constraint group, the manifest constructor can be equalised with all ambiguous constructors, because it designates the supermonad or superapplicative this group is working with. The example from Step 1 demonstrates this. In Figure 5.1 the ambiguous constructors `m`, `n`, and `p` are equalised with the manifest constructor `[]`, whereas `s` and `t` are equalised with `Maybe`.

Finding more than one manifest constructor in a constraint group is nonsensical, because that would imply that the programmer is using several different supermonads or superapplicatives within the same computation. Therefore, the plugin aborts in this case.

If there are manifest and variable constructors involved with the constraint group the plugin also needs to abort. Again this situation is nonsensical, because the programmer is already designating the supermonad or applicative that is used throughout the computation with the manifest constructors, which means there should not be any variable constructors.

If there is no manifest constructor and at least one variable constructor in use, the constraint group originates from a function that is polymorphic in the supermonad and superapplicative being used. An example for this would be the monadic function `forever`:

```
forever :: (Bind m n n, BindCts m n n) => m a -> n b
forever ma = ma >>= ( \_ -> forever ma )
```

To declare this function in the most general form the type needs to involve two different variable constructors (`m` and `n`), because, depending on the supermonad in use, `m` and `n` are not necessarily the same. The function cannot be more precise about the partial applications that form `m` and `n` either, because their arity and relationship depends on the instantiating monadic notion. Thus, the programmer is required to use several variable constructors to express the function's type.

To solve the ambiguous constructors the plugin needs to check all associations between ambiguous and variable constructors and see which associations are satisfiable by the given constraints. If there is only one satisfiable association, the plugin knows that it is the one intended by the programmer. If there are several possible associations the plugin needs to abort, because the function's

type is ambiguous and committing to one association may result in unintended runtime behaviour.

This process can be illustrated with the function `forever`. GHC infers the following constraints:

```
Bind m s n -- From the use of (>>=).
Bind m s s -- From the use of 'forever'.
```

The variable constructors `m` and `n` of the first constraint are inferred by unification with the type signature of `forever`. The recursive call of `forever` leads to the second constraint, which contains `m` due to the application to `ma`. Since there is no further information available GHC infers the most general type for the missing constraint arguments, resulting in the introduction of the ambiguous constructor `s`. However, due to the shape of the constraints given by the type signature of `forever`, GHC can infer that the second and third argument need to be equal.

From the ambiguous constructor `s` and the variable constructors `m` and `n` the plugin can construct two possible associations: $\{\ \mathtt{s} \mapsto \mathtt{m}\ \}$ and $\{\ \mathtt{s} \mapsto \mathtt{n}\ \}$. As only one of the associations is satisfiable by the given constraints for `forever`, i.e., $\{\ \mathtt{s} \mapsto \mathtt{n}\ \}$, the plugin uses that association to solve the ambiguous constructor and ignores the other association.

The runtime of checking all associations is exponential in the number of ambiguous and variable constructors. However, the experience from implementing the standard library functions suggests that this is not a problem in practice as functions that are polymorphic in the used supermonad tend to be short and their types only contain small numbers of variables.

Examples with multiple satisfiable associations can be constructed by adding constraints that are not necessary to solve the ambiguous constructors. However, in practice I have not encountered a polymorphic function with several satisfiable associations. I suspect this is due to the fact that the library functions only provide the minimal amount of constraints necessary to satisfy their polymorphic types.

In cases where the runtime becomes an issue, it can be reduced by giving additional type annotations throughout the function, thus reducing the number of variables. To support programmers with this issue, future versions of the plugin could print warnings to show which computations involve many variables.

Explanation of Step 3

The final step solves indices through unification with the instance that is supposed to be used. It is motivated by the last observation: if this is not done, there may be ambiguous type variables left in the indices of partially applied base constructors that prevent GHC from solving the constraint with an instance.

For example, if a `Return (Vector n)` constraint resulted from Step 2, the plugin knows that the graded monad `Vector` is used. Thus, it can lookup the `Return` instance of the `Vector` type constructor and solve `n` by unifying

the instance arguments with the constraint arguments, which results in `n` being equalised with `1` (encoded as `'S 'Z`).

That this process works is ensured by the assumption that there is exactly one `Bind` (and/or `Applicative`) and one `Return` instance per base constructor. If there were several, it would be unclear which one to use for this step. If there is no instance the indices cannot be unified at all. The plugin ensures that both (or all three) instances exist for each base constructor.

In conclusion, the presented arguments and the conducted case studies (Section 5.4) give confidence that the plugin restores GHC's ability to infer the type of supermonad and superapplicative computations.

5.3.2 Using the plugin

To use the plugin in a module, the programmer has to do four things:

- Enable the GHC language extension `RebindableSyntax`:

  ```
  {-# LANGUAGE RebindableSyntax #-}
  ```

- Activate the supermonad plugin by inserting the following line at the top of the module:

  ```
  {-# OPTIONS_GHC -fplugin Control.Super.Monad.Plugin #-}
  ```

- Import `Control.Super.Monad.Prelude`. This module provides all the functionality of the standard `Prelude`, except that the parts of the prelude relating to standard monads and applicatives are replaced with their appropriate counterparts for supermonads and superapplicatives.

- Finally, the user has to implement instances of the `Bind`, `Applicative`, and `Return` classes for all of their supermonads and superapplicatives. Most of the standard library and monad transformer instances are provided by the supermonad library already.

An example of a module that performs the first three steps can be seen in Figure 5.2. The supermonad library repository[11] contains several examples that demonstrate how supermonads can be used as well.

A collection of supermonad and superapplicative related functions that are not in the standard prelude is provided in the following module:

```
Control.Super.Monad.Functions
```

To work with the constrained variant of supermonads and superapplicatives use the alternative prelude:

```
Control.Super.Monad.Constrained.Prelude
```

[11] GitHub: *jbracker/supermonad* - `https://github.com/jbracker/supermonad`

```
{-# LANGUAGE RebindableSyntax #-}
{-# OPTIONS_GHC -fplugin Control.Super.Monad.Plugin #-}

module ExampleModule where

import Control.Super.Monad.Prelude
```

Figure 5.2: Example of a module header that enables the use of supermonads and superapplicatives.

Non-prelude functions of the constrained variant can be found in the following module:

```
Control.Super.Monad.Constrained.Functions
```

As mentioned before the plugin has been tested with the following versions of GHC: 7.10.3, 8.0.1, 8.2.1, and 8.4.2.

5.4 Case studies

To provide evidence for the practicality of the supermonads approach, and to check that everything works as intended, I carried out two case studies. The case studies also constitute a stress test of the plugin on a larger code base. The source code of these case studies is also available in the supermonad repository[12].

5.4.1 Teaching compiler

The first case study applies supermonads and superapplicatives to a teaching compiler. The compiler is made up of 25 modules containing more than 3800 lines of code (not counting blank lines and comments). Most of that code uses the do-notation to express computations involving standard monads. The code uses a range of custom and predefined monads and involves monad transformers as well as fixed points, i.e., recursive do-notation. Therefore, the compiler provides a good stress test for the plugin and a possibility to see if there are any problems when using supermonads and superapplicatives to replace the standard notions.

To adapt the compiler to use supermonads, the first three steps of Section 5.3.2 were applied to each module and instances of the `Bind`, `Applicative`, and `Return` classes for each of the custom monads defined in the compiler were provided as well. The `DFT` type is used to add the handling of failures to the compiler and provides a good example.

```
newtype DFT m a = DFT { unDFT :: m (Maybe a) }
```

[12] GitHub: *jbracker/supermonad* - `https://github.com/jbracker/supermonad`

Originally, the monad instance had the following form:

```
instance ( Monad m ) => Monad (DFT m) where
  m >>= f = DFT ( unDFT m >>= \ma -> case ma of
      Nothing -> return Nothing
      Just a  -> unDFT (f a) )
  return a = DFT ( return (Just a) )
```

Without changing the implementation this can be translated into the following supermonad instances:

```
instance ( Bind m n p, Return n ) =>
         Bind (DFT m) (DFT n) (DFT p) where
  type BindCts (DFT m) (DFT n) (DFT p)
       = (BindCts m n p, ReturnCts n)
  m >>= f = DFT ( unDFT m >>= \ma -> case ma of
      Nothing -> return Nothing
      Just a  -> unDFT (f a) )

instance (Return m) => Return (DFT m) where
  return a = DFT ( return (Just a) )
```

The instances were also generalised at the same time. This allows arbitrary supermonads to be wrapped in `DFT`, because the `Bind m n p` constraint was used instead of `Bind m m m`.

In addition, the functions and classes that are polymorphic in their monad had to be modified. Their `Monad m` constraints had to be replaced with `Bind m m m` and `Return m` constraints. In addition, the new bind constraints `BindCts m m m` had to be added to every function involving a bind operation.

One example where these changes were necessary is the `Diagnostic` type class.

```
class ( Applicative d, Monad d ) => Diagnostic d where
  emitD :: String -> d ()
  (|||) :: d a -> d a -> d a
  -- ...
```

The class was made applicable to supermonads through the naive process described in Section 5.1.6.

```
class ( Applicative d d d, Bind d d d, Return d ) =>
      Diagnostic d where
  emitD :: (ApplicativeCts d d d, BindCts d d d, ReturnCts d)
        => String -> d ()
  (|||) :: (ApplicativeCts d d d, BindCts d d d, ReturnCts d)
        => d a -> d a -> d a
  -- ...
```

There was no need to change any of the instances.

Note that only the naive conversion was applied, because the classes were written having standard monads specifically in mind. Generalising them to apply to generalised monads would require a careful redesign. Depending on how general the adjusted classes are, it may be necessary to list the required constraints of each type class function in an individual associated constraint as demonstrated in Section 5.1.6.

Altogether switching to supermonads and superapplicatives did not require many changes. In 22 of the 25 modules the only required change was adding the compiler directives and the import of the supermonad library (~3 lines per module). The remaining three modules additionally required `Bind`, `Applicative`, and `Return` instances for monads defined in them. These instances required about 10-15 additional lines for each of the five monads. Finally, some type classes such as `Diagnostic` and some polymorphic functions had to be modified which required changing about 20 lines in total. All of these changes only involved the work that is described above.

As can be seen, porting code from standard to supermonads almost exclusively involved adjusting for the `Bind`, `Applicative`, and `Return` type classes and activating the plugin in each module. Type inference was not affected by the change to supermonads and the adjustments to use the supermonad classes were reasonably mechanical.

5.4.2 Chat server and client

The second case study involves generalised and standard monads. Unfortunately, the only examples I found using generalised monads did no longer compile. Therefore, I decided to implement my own application: a chat server. It uses session types as presented by Pucella and Tov [PT08] in their `simple-sessions` library[13]. As the library does not support network communication, the example uses communication among threads. Other participants in a chat are simulated using bots.

The chat server is made up of 5 modules containing more than 500 lines of code (not counting blank lines and comments). Most of that code uses the `do`-notation to express computations involving the standard monads `IO` and `STM` in addition to the indexed monad `Session`.

I first implemented the chat server without supermonads to provide a point of reference for comparison after refactoring to use supermonads.

The non-supermonad implementation only relies on `RebindableSyntax` and requires approximately 40 lines (~8%) of additional annotations to specify which bind and return operation to use in computations involved with the generalised `Session` monad. If the bind and return operations used by the `Session` monad were not named differently from the standard operations, the amount of anno-

[13] Hackage: *simple-sessions* - `http://hackage.haskell.org/package/simple-sessions`

tations required would have been considerably higher: In that case, annotations would have been necessary for all of the monadic computations involving standard monads, in addition to those already present for computations that involve the indexed monad `Session`.

The refactoring to use supermonads only required the expected changes:

- Import of the custom prelude and activation of the plugin in all modules.
- Removal of the additional annotations that were previously necessary to specify which bind and return operation to use.
- Implementation of supermonad instances for the generalised `Session` monad.

The removal of annotations made the implementation more concise. For example, when using nested monadic computations it is not possible to use the `where`-notation to add annotations. Therefore, it is necessary to use local `let` bindings, which cluttered the code:

```
do
  -- ...
  run ( let (>>=) = (Prelude.>>=)
            (>>) = (Prelude.>>)
    in do {- ... -} )
  -- ...
```

The annotations are necessary because the `RebindableSyntax` extension replaces the operations from the standard monad class with any functions in scope that use the names >>=, >>, `return`, and `fail`. Thus, if there are several different monadic notions in scope, every `do`-block requires disambiguation.

After refactoring to use supermonads, the local `let` bindings can be removed altogether:

```
do
  -- ...
  run ( do
    {- ... -} )
  -- ...
```

In conclusion, the refactoring to use supermonads allowed for a more concise implementation by obviating the need for annotations, thus saving 40 lines (~8%) of code. Additionally, it also allowed the shared use of standard library functions such as `unless`, `when` and `void` for standard as well as generalised monads.

Again, it can be seen how supermonads ease the use of different monadic notions in the same application and enable the reuse of code.

5.5 Limitations and conclusions

This chapter explained how supermonads and superapplicatives provide a unified interface for all of the monadic and applicative notions introduced in Section 3. They remedy the inconvenience of manually specifying which monadic generalisation is to be used within the `do`-notation. Supermonads achieve this through a collections of type classes, i.e., `Bind`, `Applicative`, and `Return`, in conjunction with a GHC plugin and the rebindable syntax language extension. These type classes also allow writing functions that work for all supermonads or superapplicatives and thus for all generalisations supported by them.

In contrast to polymonads, supermonads and superapplicatives provide a solution to all of the problems and goals presented in Section 1.1 of the introduction. The major limitation of the supermonad approach is that the number of constraints and their complexity may become burdensome for the programmer involved with writing libraries and polymorphic code based on them (see Section 5.1.6). A more detailed comparison between polymonads and supermonads is given in Section 8.1. In summary, both notions are probably incomparable, because there are generalisations that polymonads can express and supermonads cannot and the same is also true the other way around.

The other shortcoming is the lack of theoretic foundation for the supermonad approach. A possible categorical foundation that solves this shortcoming is discussed in Section 6. As mentioned in Section 5.1.4, one of my previous papers [BN16] provides laws for supermonads that are based on the standard monad laws. These laws provide a useful guideline to judge whether a supermonad instance is valid.

Finally, as with polymonads only those extensions of the language (supported by GHC) that were relevant to the implementation were tested to work properly in the context of supermonads. Comprehensive testing for other language extensions has yet to be carried out, though I am not aware of any GHC related problems resulting from the use of supermonads.

Chapter 6

Category theoretical context

The original formalisation of supermonads in previous work [BN16] was based solely on the unified representation of the different monadic notions in Haskell. As such, the formalisation did not relate to the common categorical models usually used to model functors, applicatives and monads in Haskell. In the following, to address this shortcoming, I introduce a model for supermonads and superapplicatives that is founded in category theory.

Consequently, the notion of supermonad or superapplicative refers to the general approach and technique of a unified encoding and implementation of the different monadic and applicative notions in a specific language, such as Haskell. The semantics and the supported generalisations should be characterised by the categorical notions presented in this section.

In Section 6.1 the prerequisite categorical definitions that are required to understand the following sections are introduced. Based on these definitions, Section 6.2 and 6.3 show how the different monadic and applicative notions in Haskell relate to category theory and develop categorical adaptations that precisely capture them. The discussion leads to categorical structures that capture all of the monadic and applicative notions, respectively. As a result, a hierarchy of monadic and applicative notions that shows how all of them are related to each other is presented in Section 6.4.

I hope the categorical notions and relationships developed in this section prove to be as useful design patterns for future language design as standard functors, applicatives, and monads have been thus far.

As the target audience for my thesis are functional programmers only a basic understanding of category theory [Mac69; Pie91; Awo06] is expected: familiarity with categories[1], functors[2], and natural transformations[3]. All other required notions are introduced. The following naming conventions are used in the following sections.

[1] Agda formalisation: `Theory.Category.Definition`
[2] Agda formalisation: `Theory.Functor.Definition`
[3] Agda formalisation: `Theory.Natural.Transformation`

Categories	$\mathbb{C}, \mathbb{D}, \mathbb{E}, \ldots$
Functors	$F, G, H, \ldots$
Natural transformations or isomorphism	$\eta, \mu, \theta, \iota, \ldots$
Objects or 0-cells	$a, b, c, d, \ldots$
Morphisms or 1-cells	$f, g, h, \ldots$
2-cells	$\alpha, \beta, \gamma, \ldots$

The propositions presented in this chapter will describe how two structures are related to each other. Most of the time this relationship will be in form of a one-to-one correspondence.

Definition 6.1 (One-to-One Correspondence, $\leftrightarrow$)**.** Two structures are in *one-to-one correspondence* if there is a bijection between these structures. Hence, each instance of one structure gives rise to an instance of the other and vice versa. The symbol $\leftrightarrow$ is used to denote a one-to-one correspondence between two structures, e.g., $A \leftrightarrow B$.

6.1 Prerequisite categorical notions

This section gives an intuition and a full definition for each prerequisite structure. The intuition should be enough to understand the following sections that use the introduced categorical structures.

6.1.1 Discrete and codiscrete collections of morphisms

A category is *discrete* [Mac69; Pie91; Awo06] if the identity morphisms for each object are the only morphisms.

Definition 6.2 (Discrete category)**.** Let $\mathbb{C}$ be a category. $\mathbb{C}$ is called a *discrete* category iff:

- $\mathsf{Hom}_{\mathbb{C}}(a,a) = \{\ \mathsf{id}_a\ \}$ for all $a \in \mathsf{Obj}_{\mathbb{C}}$ and
- $\mathsf{Hom}_{\mathbb{C}}(a,b) = \emptyset$ for all $a, b \in \mathsf{Obj}_{\mathbb{C}}$ such that $a \neq b$.

A category is *codiscrete* if it contains exactly one morphism for each pair of objects.

Definition 6.3 (Codiscrete category)**.** Let $\mathbb{C}$ be a category. $\mathbb{C}$ is called a *codiscrete* category iff $|\mathsf{Hom}_{\mathbb{C}}(a,b)| = 1$ for all $a, b \in \mathsf{Obj}_{\mathbb{C}}$.

6.1.2 Natural isomorphisms

A natural transformation $\eta : F \xrightarrow{\cdot} G$ is a *natural isomorphism*[4] [Mac69; Pie91; Awo06] if its components $\eta_a : F(a) \to G(a)$ are guaranteed to have an inverse

[4] Agda formalisation: `Theory.Natural.Isomorphism`

$\eta_a^{-1} : G(a) \to F(a)$. In other words, if a natural transformation provides a mapping between two functors then a natural isomorphism provides a "bijective" mapping between them.

Definition 6.4 (Natural isomorphism)**.** Let $\mathbb{C}$ and $\mathbb{D}$ be categories and $F, G : \mathbb{C} \longrightarrow \mathbb{D}$ functors between them. A natural transformation $\eta : F \xrightarrow{\cdot} G$ is a *natural isomorphism* if the morphism η_a is invertible for all $a \in \mathsf{Obj}_{\mathbb{C}}$, i.e., there exists η_a^{-1} for each $a \in \mathsf{Obj}_{\mathbb{C}}$ such that

$$\eta_a \circ_{\mathbb{D}} \eta_a^{-1} = \mathsf{id}_{G(a)} \text{ and } \eta_a^{-1} \circ_{\mathbb{D}} \eta_a = \mathsf{id}_{F(a)},$$

Hence, η_a is an isomorphism for all a. η is denoted to be a natural isomorphism by writing $\eta : F \xrightarrow{\cong} G$.

6.1.3 Monoidal categories

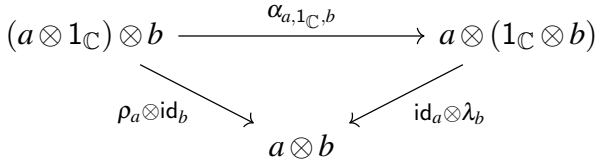
A category $\mathbb{C}$ consists of a collection of objects $\mathsf{Obj}_{\mathbb{C}}$ and a collection of morphisms $\mathsf{Hom}_{\mathbb{C}}(-,-)$ between these objects. The category laws ensure that certain morphisms exist and that they behave as expected when composed. In contrast, the objects have no associated laws or structure. A *monoidal category*[5] [Bén63; Mac69] introduces the structure of a monoid on the objects of a category. A functor $\otimes : \mathbb{C} \times \mathbb{C} \longrightarrow \mathbb{C}$, called the *tensor product*, is introduced as the monoidal operation and there needs to be one object $1_{\mathbb{C}}$, called the *tensor unit*, that represents the neutral element of the monoid. There are three natural isomorphisms that ensure the tensor product is weakly associative and obeys left and right identity. Finally, two laws ensure that the natural isomorphisms behave as expected.

Definition 6.5 (Monoidal category)**.** A monoidal category is a category $\mathbb{C}$ equipped with

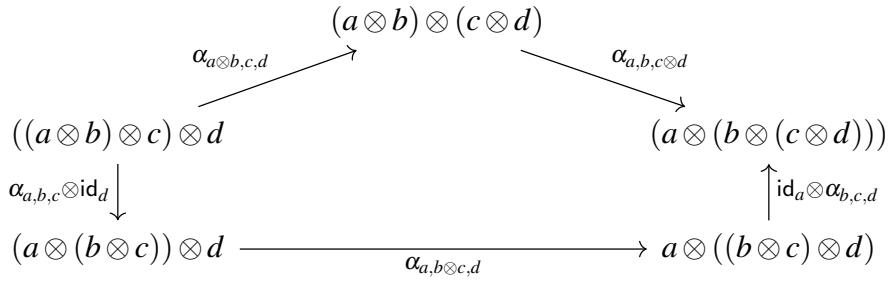
- a functor $\otimes : \mathbb{C} \times \mathbb{C} \longrightarrow \mathbb{C}$ called the *tensor product*,
- an object $1_{\mathbb{C}} \in \mathsf{Obj}_{\mathbb{C}}$ called the *unit object* or *tensor unit*,
- a natural isomorphism $\alpha : ((- \otimes -) \otimes -) \xrightarrow{\cong} (- \otimes (- \otimes -))$ with components of the form $\alpha_{a,b,c} : (a \otimes b) \otimes c \to a \otimes (b \otimes c)$ called the *associator*,
- a natural isomorphism $\lambda : (1_{\mathbb{C}} \otimes -) \xrightarrow{\cong} (-)$ with components of the form $\lambda_a : 1_{\mathbb{C}} \otimes a \to a$ called the *left unitor*, and
- a natural isomorphism $\rho : (- \otimes 1_{\mathbb{C}}) \xrightarrow{\cong} (-)$ with components of the form $\rho_a : a \otimes 1_{\mathbb{C}} \to a$ called the *right unitor*,

[5] Agda formalisation: `Theory.Category.Monoidal`

such that the diagram for the *triangle identity*



and the diagram for the *pentagon identity*



both commute for all $a, b, c, d \in \mathsf{Obj}_{\mathbb{C}}$.

Example 6.6 (Unit)**.** A trivial example of a monoidal category is the unit category $\mathbf{1}$ with exactly one object and morphism.[6]

Example 6.7 (Monoid)**.** A monoid $(M, \diamond, e)$ forms a monoidal category Mon_M. The carrier M provides the objects and the morphisms are discrete. The tensor product is then provided by the monoidal operation $\diamond$ and the tensor unit is the neutral element e.[7]

Example 6.8 (Set)**.** The category Set is another example of a monoidal category. Here the cartesian product provides the tensor product and unit provides the tensor unit.[8]

Example 6.9 (Endofunctors and natural transformations)**.** Given a category $\mathbb{C}$ the monoidal category $[\mathbb{C}, \mathbb{C}]_\circ$ can be formed. The objects of this category are the endofunctors on $\mathbb{C}$ and the morphisms are the natural transformations between them. The tensor product is provided by composition of functors and the tensor unit is the identity endofunctor on $\mathbb{C}$.[9] Note that in the resulting monoidal category the associator and unitors are all strict, i.e., they are identities.

6.1.4 Lax monoidal functors

A *lax monoidal functor*[10] [Bén63; Mac69] is a functor that maps between monoidal categories instead of non-monoidal categories. This means, in addition to preserving the categorical structure, it also preserves the monoidal structures on the objects across the mapping.

[6] Agda proof: `Theory.Category.Monoidal.Examples.Unit`
[7] Agda proof: `Theory.Category.Monoidal.Examples.Monoid`
[8] Agda proof: `Theory.Category.Monoidal.Examples.SetCat`
[9] Agda proof: `Theory.Category.Monoidal.Examples.FunctorWithComposition`
[10] Agda formalisation: `Theory.Functor.Monoidal`

Just as the basis of a monoidal category is a category, a lax monoidal functor between the monoidal categories $\mathbb{C}$ and $\mathbb{D}$ is based on a functor $F : \mathbb{C} \longrightarrow \mathbb{D}$ between these categories. The tensor units are connected through a morphism from the tensor unit $1_{\mathbb{D}}$ of $\mathbb{D}$ to the mapping of the tensor unit $F(1_{\mathbb{C}})$ from $\mathbb{C}$. To relate the tensor products, a natural transformation maps the tensor product $F(a) \otimes_{\mathbb{D}} F(b)$ in $\mathbb{D}$ to the mapped tensor product $F(a \otimes_{\mathbb{C}} b)$ for all $a, b \in \mathsf{Obj}_{\mathbb{C}}$. The laws ensure that these mappings preserve associativity and the left and right identity.

Definition 6.10 (Lax monoidal functor)**.** Let $(\mathbb{C}, \otimes_{\mathbb{C}}, 1_{\mathbb{C}})$ and $(\mathbb{D}, \otimes_{\mathbb{D}}, 1_{\mathbb{D}})$ be two monoidal categories. A *lax monoidal functor* between them consists of

- a functor $F : \mathbb{C} \longrightarrow \mathbb{D}$,
- a morphism $\eta : 1_{\mathbb{D}} \to F(1_{\mathbb{C}})$, and
- a natural transformation

$$\mu_{a,b} : F(a) \otimes_{\mathbb{D}} F(b) \xrightarrow{\cdot} F(a \otimes_{\mathbb{C}} b)$$

  for all $a, b \in \mathsf{Obj}_{\mathbb{C}}$,

such that the diagrams for *associativity*

$$\begin{array}{ccc}
(F(a) \otimes_{\mathbb{D}} F(b)) \otimes_{\mathbb{D}} F(c) & \xrightarrow{\alpha^{\mathbb{D}}_{F(a),F(b),F(c)}} & F(a) \otimes_{\mathbb{D}} (F(b) \otimes_{\mathbb{D}} F(c)) \\
\downarrow{\scriptstyle \mu_{a,b} \otimes_{\mathbb{D}} \mathsf{id}_{F(c)}} & & \downarrow{\scriptstyle \mathsf{id}_{F(a)} \otimes_{\mathbb{D}} \mu_{b,c}} \\
F(a \otimes_{\mathbb{C}} b) \otimes_{\mathbb{D}} F(c) & & F(a) \otimes_{\mathbb{D}} F(b \otimes_{\mathbb{C}} c) \\
\downarrow{\scriptstyle \mu_{a \otimes_{\mathbb{C}} b, c}} & & \downarrow{\scriptstyle \mu_{a, b \otimes_{\mathbb{C}} c}} \\
F((a \otimes_{\mathbb{C}} b) \otimes_{\mathbb{C}} c) & \xrightarrow[F(\alpha^{\mathbb{C}}_{a,b,c})]{} & F(a \otimes_{\mathbb{C}} (b \otimes_{\mathbb{C}} c))
\end{array}$$

left unitality

$$\begin{array}{ccc}
1_{\mathbb{D}} \otimes_{\mathbb{D}} F(a) & \xrightarrow{\eta \otimes_{\mathbb{D}} \mathsf{id}_a} & F(1_{\mathbb{C}}) \otimes_{\mathbb{D}} F(a) \\
\downarrow{\scriptstyle \lambda^{\mathbb{D}}_{F(a)}} & & \downarrow{\scriptstyle \mu_{1_{\mathbb{C}}, a}} \\
F(a) & \xleftarrow[F(\lambda^{\mathbb{C}}_a)]{} & F(1_{\mathbb{C}} \otimes_{\mathbb{C}} a)
\end{array}$$

and *right unitality*

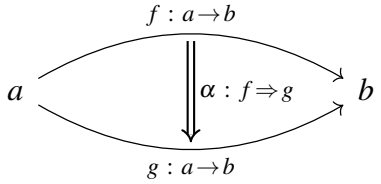
$$\begin{array}{ccc}
F(a) \otimes_{\mathbb{D}} 1_{\mathbb{D}} & \xrightarrow{\mathsf{id}_a \otimes_{\mathbb{D}} \eta} & F(a) \otimes_{\mathbb{D}} F(1_{\mathbb{C}}) \\
\downarrow{\scriptstyle \rho^{\mathbb{D}}_{F(a)}} & & \downarrow{\scriptstyle \mu_{a, 1_{\mathbb{C}}}} \\
F(a) & \xleftarrow[F(\rho^{\mathbb{C}}_a)]{} & F(a \otimes_{\mathbb{C}} 1_{\mathbb{C}})
\end{array}$$

commute for all $a, b, c \in \mathsf{Obj}_{\mathbb{C}}$.

That the monoidal functor is "lax" just indicates that its morphism η and natural transformations $\mu_{-,-}$ are not isomorphisms. My work only requires the notion of a lax monoidal functor, but it should be mentioned that there are other variations such as oplax, strong or strict monoidal functors.

6.1.5 Strict 2-categories

Strict 2-categories [Bén65; Bén67; Mac69] are a generalisations of categories. A strict 2-category[11] $\mathbb{C}$ has a collection of objects, called *0-cells*, just like a category, but instead of a collection of morphisms there is a category for every two objects, the *homomorphism category* $\mathbb{C}(-,-)$. Thus, in a 2-category the morphisms, called *1-cells*, are provided by the objects of the homomorphism categories and the morphisms of the homomorphism categories represent morphisms between morphisms, called *2-cells*:



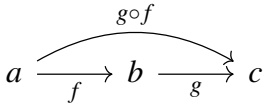
a, b, c	object or 0-cell
f, g, h	morphism or 1-cell
α, β	2-cell

The homomorphism categories only provide identity 2-cells. Therefore, the identity 1-cells 1_a need to be provided separately for all $a \in \mathsf{Obj}_{\mathbb{C}}$.

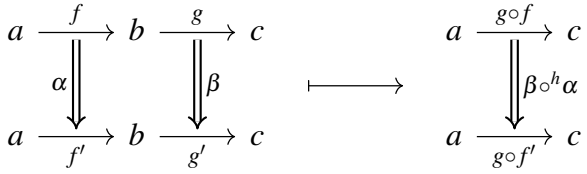
The composition of homomorphism categories is given by the *composition functor*

$$\mathsf{Comp}_{a,b,c} : \mathbb{C}(b,c) \times \mathbb{C}(a,b) \longrightarrow \mathbb{C}(a,c).$$

The object mapping of Comp composes 1-cells (morphisms) just as in a category:



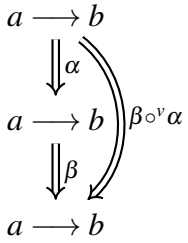
Notice that there are two different ways to compose 2-cells. The first is the *horizontal composition* provided by the morphism mapping of Comp:



The second is the *vertical composition* provided in each of the homomorphism

[11] Agda formalisation: `Theory.TwoCategory.Definition`

categories:



The laws of strict 2-categories ensure that horizontal composition and composition between 1-cells abide associativity, left identity and right identity. Vertical composition fulfills all these laws, because it originates from the homomorphism categories.

Definition 6.11 (Strict 2-category)**.** A *strict 2-category* $\mathbb{C}$ consists of

- a collection $\mathsf{Obj}_{\mathbb{C}}$ of objects,
- for each pair of objects $a,b \in \mathsf{Obj}_{\mathbb{C}}$ a category $\mathbb{C}(a,b)$ (*homomorphism category*),
- for each object $a \in \mathsf{Obj}_{\mathbb{C}}$ a object $1_a \in \mathsf{Obj}_{\mathbb{C}(a,a)}$ (*unit*) and
- for each triple of object $a,b,c \in \mathsf{Obj}_{\mathbb{C}}$ a functor

$$\mathsf{Comp}_{a,b,c} : \mathbb{C}(b,c) \times \mathbb{C}(a,b) \longrightarrow \mathbb{C}(a,c)$$

  called the *composition functor*.

The elements in $\mathsf{Obj}_{\mathbb{C}}$ are called *0-cells*. For any $a,b \in \mathsf{Obj}_{\mathbb{C}}$ the morphisms in $\mathsf{Obj}_{\mathbb{C}(a,b)}$ are called *1-cells* and the elements of $\mathsf{Hom}_{\mathbb{C}(a,b)}(-,-)$ are called *2-cells*. The object-level mapping of Comp provides composition on 1-cells (denoted as $\circ$), the morphism-level mapping of Comp provides *horizontal composition* on 2-cells (denoted as $\circ^h$) and the composition provided by homomorphism categories on 2-cells is *vertical composition* (denoted as $\circ^v$).

The 1-cell and horizontal composition provided by the composition functor are required to satisfy the following laws:

- For all $a,b,c,d \in \mathsf{Obj}_{\mathbb{C}}$, $h \in \mathsf{Obj}_{\mathbb{C}(c,d)}$, $g \in \mathsf{Obj}_{\mathbb{C}(b,c)}$ and $f \in \mathsf{Obj}_{\mathbb{C}(a,b)}$ the *associativity* of 1-cell composition

$$(h \circ g) \circ f = h \circ (g \circ f),$$

- for all $a,b,c,d \in \mathsf{Obj}_{\mathbb{C}}$, $f,f' \in \mathsf{Obj}_{\mathbb{C}(a,b)}$, $g,g' \in \mathsf{Obj}_{\mathbb{C}(b,c)}$, $h,h' \in \mathsf{Obj}_{\mathbb{C}(c,d)}$, $\alpha \in \mathsf{Hom}_{\mathbb{C}(c,d)}(h,h')$, $\beta \in \mathsf{Hom}_{\mathbb{C}(b,c)}(g,g')$ and $\gamma \in \mathsf{Hom}_{\mathbb{C}(a,b)}(f,f')$ the *associativity* of horizontal composition

$$(\alpha \circ^h \beta) \circ^h \gamma = \alpha \circ^h (\beta \circ^h \gamma),$$

- for all $a, b \in \mathsf{Obj}_{\mathbb{C}}$ and $f \in \mathsf{Obj}_{\mathbb{C}(a,b)}$ the *left* and *right identity* of 1-cell composition
$$1_b \circ f = f = f \circ 1_a \quad \text{and}$$
- for all $a, b \in \mathsf{Obj}_{\mathbb{C}}$, $f, g \in \mathsf{Obj}_{\mathbb{C}(a,b)}$ and $\alpha \in \mathsf{Hom}_{\mathbb{C}(a,b)}(f, g)$ the *left* and *right identity* of horizontal composition
$$\mathsf{id}_{1_b} \circ^h \alpha = \alpha = \alpha \circ^h \mathsf{id}_{1_a}.$$

Example 6.12 (Unit)**.** A trivial example is the unit 2-category **1** with exactly one 0-cell, one 1-cell and one 2-cell.[12]

Example 6.13 (2-Category of categories)**.** The canonical example is Cat the strict 2-category of (small) categories, functors and natural transformations, which also provides the intuition behind most of the above notation.[13]

Example 6.14 (Discrete 2-category)**.** Another example is the *discrete* strict 2-category formed by a category $\mathbb{C}$. Here the 0-cells are the objects of $\mathbb{C}$ and the 1-cells are the morphisms of $\mathbb{C}$. The only 2-cells are the identity 2-cells for each morphism. This category is denoted $\mathsf{Disc}_{\mathbb{C}}$.[14]

Example 6.15 (Monoid 2-category)**.** Given a monoid $(M, \diamond, e)$ a strict 2-category Mon^2_M with exactly one 0-cell can be defined. The 1-cells are the elements of the carrier M (endomorphisms) and the 2-cells are discrete.[15]

6.1.6 Lax 2-functors

A *lax 2-functor*[16] [Bén65; Bén67; Mac69] between strict 2-categories is generalised in a similar way as categories are generalised to 2-categories. Like a functor it has an object or 0-cell mapping. The mapping of morphisms becomes a functor between the homomorphism categories of the two involved 2-categories.

A functor has laws to ensure identity and composition is preserved across the mapping. When generalising these laws, they may stay equalities or become 2-cell isomorphisms. Since these 2-functors are *lax* the laws reduce to 2-cells without an inverse. Thus, a lax 2-functor requires the existence of a 2-cell that maps 1-cell identities and preserves 1-cell composition instead of having the classical laws of a functor.

To make sure that a 2-functor still behaves as expected it needs to make sure that it preserves associativity, left identity and right identity on 1-cells. Therefore, a 2-functor also has a set of coherence laws.

[12] Agda proof: `Theory.TwoCategory.Examples.Unit`
[13] Agda proof: `Theory.TwoCategory.Examples.Functor`
[14] Agda proof: `Theory.TwoCategory.Examples.DiscreteHomCat`
[15] Agda proof: `Theory.TwoCategory.Examples.Monoid`
[16] Agda formalisation: `Theory.TwoFunctor.Definition`

Definition 6.16 (Lax 2-functor)**.** A *lax 2-functor* $F : \mathbb{C} \longrightarrow \mathbb{D}$ from a strict 2-category $\mathbb{C}$ to a strict 2-category $\mathbb{D}$ consists of

- a 0-cell mapping $F : \mathsf{Obj}_{\mathbb{C}} \to \mathsf{Obj}_{\mathbb{D}}$,
- for each homomorphism category $\mathbb{C}(a,b)$ a functor $F_{a,b} : \mathbb{C}(a,b) \longrightarrow \mathbb{D}(F(a),F(b))$ which provides the 1- and 2-cell mappings,
- for each object $a \in \mathsf{Obj}_{\mathbb{C}}$, a 2-cell $\eta_a : 1_{F(a)} \Rightarrow F_{a,a}(1_a)$ in $\mathbb{D}$, and
- for each triple of objects $a,b,c \in \mathsf{Obj}_{\mathbb{C}}$ a 2-cell

$$\mu_{a,b,c}(f,g) : F_{b,c}(g) \circ F_{a,b}(f) \Rightarrow F_{a,c}(g \circ f)$$

that is natural in the 1-cells $f \in \mathsf{Obj}_{\mathbb{C}(a,b)}$ and $g \in \mathsf{Obj}_{\mathbb{C}(b,c)}$

such that associativity, left unitality and right unitality for horizontal composition are preserved; this means the following diagrams need to commute

$$\begin{array}{ccc}
F_{c,d}(h) \circ_{\mathbb{D}} (F_{b,c}(g) \circ_{\mathbb{D}} F_{a,b}(f)) & \xRightarrow{\mathsf{id}_{(F_{c,d}(h) \circ_{\mathbb{D}} F_{b,c}(g) \circ_{\mathbb{D}} F_{a,b}(f))}} & (F_{c,d}(h) \circ_{\mathbb{D}} F_{b,c}(g)) \circ_{\mathbb{D}} F_{a,b}(f) \\
{\scriptstyle \mathsf{id}_{F_{c,d}(h)} \circ^h_{\mathbb{D}} \mu_{a,b,c}(f,\,g)}\Downarrow & & \Downarrow{\scriptstyle \mu_{b,c,d}(g,\,h) \circ^h_{\mathbb{D}} \mathsf{id}_{F_{a,b}(f)}} \\
F_{c,d}(h) \circ_{\mathbb{D}} F_{a,c}(g \circ_{\mathbb{C}} f) & & F_{b,d}(h \circ_{\mathbb{C}} g) \circ_{\mathbb{D}} F_{a,d}(f) \\
{\scriptstyle \mu_{a,c,d}((g \circ_{\mathbb{C}} f),\,h)}\Downarrow & & \Downarrow{\scriptstyle \mu_{a,b,d}(f,\,(h \circ_{\mathbb{C}} g))} \\
F_{a,d}(h \circ_{\mathbb{C}} (g \circ_{\mathbb{C}} f)) & \xRightarrow[\mathsf{id}_{F_{a,d}(h \circ_{\mathbb{C}} g \circ_{\mathbb{C}} f)}]{} & F_{a,d}((h \circ_{\mathbb{C}} g) \circ_{\mathbb{C}} f)
\end{array}$$

$$\begin{array}{ccc}
1_{F(a)} \circ_{\mathbb{D}} F_{a,b}(f) & \xRightarrow{\eta_x \circ^h_{\mathbb{D}} \mathsf{id}_{F_{a,b}(f)}} & F_{b,b}(1_b) \circ_{\mathbb{D}} F_{a,b}(f) \\
{\scriptstyle \mathsf{id}_{F_{a,b}(f)}}\Downarrow & & \Downarrow{\scriptstyle \mu_{a,b,b}(f,1_b)} \\
F_{a,b}(f) & \xLeftarrow[F_{a,b}(\mathsf{id}_f)]{} & F_{a,b}(1_b \circ_{\mathbb{C}} f)
\end{array}$$

$$\begin{array}{ccc}
F_{a,b}(f) \circ_{\mathbb{D}} 1_{F(a)} & \xRightarrow{\mathsf{id}_{F_{a,b}(f)} \circ^h_{\mathbb{D}} \eta_x} & F_{a,b}(f) \circ_{\mathbb{D}} F_{a,a}(1_a) \\
{\scriptstyle \mathsf{id}_{F_{a,b}(f)}}\Downarrow & & \Downarrow{\scriptstyle \mu_{a,a,b}(1_a,f)} \\
F_{a,b}(f) & \xLeftarrow[F_{a,b}(\mathsf{id}_f)]{} & F_{a,b}(f \circ_{\mathbb{C}} 1_a)
\end{array}$$

for all $a,b,c,d \in \mathsf{Obj}_{\mathbb{C}}$, $f \in \mathsf{Obj}_{\mathbb{C}(a,b)}$, $g \in \mathsf{Obj}_{\mathbb{C}(b,c)}$ and $h \in \mathsf{Obj}_{\mathbb{C}(c,d)}$.

Example 6.17 (Lax monoidal functors)**.** Let $\mathbb{C}$ be a category and $(M,\diamond,e)$ a monoid. A lax monoidal functor $F : \mathsf{Mon}_M \longrightarrow [\mathbb{C},\mathbb{C}]_\circ$ is also a lax 2-functor $\mathsf{Mon}^2_M \longrightarrow \mathsf{Cat}$ that maps the single 0-cells in Mon^2_M to $\mathbb{C}$. In fact, these kinds of lax monoidal functors and lax 2-functor are in one-to-one correspondence with each other.[17]

[17] Agda proof: `Theory.TwoFunctor.Properties.IsomorphicLaxMonoidalFunctor`

6.1.7 Monads and Kleisli triples

A monad[18] [Mac69; Awo06] consists of an endofunctor $F : \mathbb{C} \longrightarrow \mathbb{C}$ in some category $\mathbb{C}$ together with two natural transformations $\eta : \mathsf{Id}_{\mathbb{C}} \stackrel{\cdot}{\longrightarrow} F$ (`return`) and $\mu : F \circ F \stackrel{\cdot}{\longrightarrow} F$ (`join`). In addition, there are coherence laws to ensure that η is the identity of μ and that μ is associative.

Definition 6.18 (Monad)**.** Let $\mathbb{C}$ be a category. A *monad* consists of

- a functor $F : \mathbb{C} \longrightarrow \mathbb{C}$,
- a natural transformation $\eta : \mathsf{Id}_{\mathbb{C}} \stackrel{\cdot}{\longrightarrow} F$, and
- a natural transformation $\mu : F \circ F \stackrel{\cdot}{\longrightarrow} F$

such that the following diagrams for associativity, left identity and right identity commute

$$\begin{array}{ccc} F(F(F(c))) & \xrightarrow{F(\mu_c)} & F(F(c)) \\ {\scriptstyle \mu_{F(c)}}\downarrow & & \downarrow{\scriptstyle \mu_c} \\ F(F(c)) & \xrightarrow[\mu_c]{} & F(c) \end{array} \qquad \begin{array}{ccc} F(c) & \xrightarrow{F(\eta_c)} & F(F(c)) \\ {\scriptstyle \eta_{F(c)}}\downarrow & \searrow & \downarrow{\scriptstyle \mu_c} \\ F(F(c)) & \xrightarrow[\mu_c]{} & F(c) \end{array}$$

for all $c \in \mathsf{Obj}_{\mathbb{C}}$.

Note that $\mathsf{Id}_{\mathbb{C}}$ is the identity functor on $\mathbb{C}$ and the composition in $F \circ F$ is the composition of functors.

There is an alternative, but equivalent[19], notion of monad in category theory, called a *Kleisli triple*[20] [Mog91].

Definition 6.19 (Kleisli triple)**.** Let $\mathbb{C}$ be a category. A *Kleisli triple* consists of

- an object mapping $F : \mathsf{Obj}_{\mathbb{C}} \to \mathsf{Obj}_{\mathbb{C}}$,
- for each object $a \in \mathsf{Obj}_{\mathbb{C}}$ a morphism $\eta_a \in \mathsf{Hom}_{\mathbb{C}}(a, F(a))$, and
- for all $a, b \in \mathsf{Obj}_{\mathbb{C}}$ and $k \in \mathsf{Hom}_{\mathbb{C}}(a, F(b))$ a morphism $k^* \in \mathsf{Hom}_{\mathbb{C}}(F(a), F(b))$, called the *Kleisli extension*,

such that

- the *right unit* law
$$k = k^* \circ_{\mathbb{C}} \eta_a,$$
- the *left unit* law
$$\eta_a^* = \mathsf{id}_{\mathbb{C}},$$

[18] Agda formalisation: `Theory.Monad.Definition`
[19] Agda proof: `Theory.Monad.Kleisli`
[20] Agda formalisation: `Theory.Monad.Kleisli`

- and the *associativity* law

$$(l^* \circ_{\mathbb{C}} k)^* = l^* \circ_{\mathbb{C}} k^*$$

hold for all $a, b, c \in \mathsf{Obj}_{\mathbb{C}}$, $k \in \mathsf{Hom}_{\mathbb{C}}(a, F(b))$ and $l \in \mathsf{Hom}_{\mathbb{C}}(b, F(c))$.

Kleisli triples are defined in terms of the familiar bind and return operations from Haskell. The return operation can directly be translated into the η morphisms. The Kleisli extension, which is the essentially the bind operation, swaps the arguments when compared to Haskell.

$$\begin{aligned} (>>=) &: F(a) \to (a \to F(b)) \to F(b) \\ (-)^* &: (a \to F(b)) \to F(a) \to F(b) \end{aligned}$$

Note that in Haskell the function and morphism arrows coincide. The three laws associated with Kleisli triples are in one-to-one correspondence with the standard monad laws from Haskell.

Interestingly, the definition of Kleisli triples neither states that F forms a functor nor that η and μ are natural. However, the laws imply that F forms a functor and that the associated morphisms are indeed natural given that the morphism mapping of F is defined as

$$F(f) \stackrel{\mathsf{def}}{=} (\eta_b \circ f)^*$$

for all $f \in \mathsf{Hom}_{\mathbb{C}}(a, b)$.

6.1.8 Relative monads

Relative monads[21] [ACU10] generalise monads. The underlying endofunctor is generalised to an arbitrary functor, i.e., allowing monads between different categories.

As discussed in the previous section, a monad in a category $\mathbb{C}$ is usually defined in terms of an endofunctor $F : \mathbb{C} \longrightarrow \mathbb{C}$ and a natural transformation $\mu : F \circ F \to F$ called join (see Definition 6.18). Due to the composition of F with itself in μ, this definition does not allow generalising F to a non-endofunctor.

As a result, relative monads generalise Kleisli triples (Definition 6.19) instead of standard categorical monads. The Kleisli triple allows replacing the mapping F with a mapping between different categories $F : \mathsf{Obj}_{\mathbb{C}} \to \mathsf{Obj}_{\mathbb{D}}$. However, the Kleisli-extension

$$(-)^* : (a \to F(b)) \mapsto (F(a) \to F(b))$$

maps a morphism $a \to F(b)$ and hence now requires a way to lift the object $a \in \mathsf{Obj}_{\mathbb{C}}$ into the codomain $\mathsf{Obj}_{\mathbb{D}}$ to remain a valid morphism within $\mathbb{D}$. Therefore,

[21] Agda formalisation: `Theory.Monad.Relative`

relative monads introduce an additional functor $J : \mathbb{C} \longrightarrow \mathbb{D}$ that only serves the purpose of transferring objects into F's codomain. Thus, leading to the following Kleisli-extension and return morphism:

$$(-)^* : (J(a) \to F(b)) \mapsto (F(a) \to F(b))$$
$$\eta_a : J(a) \to F(a)$$

The laws remain the same as for non-relative Kleisli triples.

Definition 6.20 (Relative monad)**.** Let $\mathbb{C}$ and $\mathbb{D}$ be categories. A *relative monad* consists of

- a functor $J : \mathbb{C} \longrightarrow \mathbb{D}$,
- an object mapping $F : \mathsf{Obj}_{\mathbb{C}} \to \mathsf{Obj}_{\mathbb{D}}$,
- for all $a \in \mathsf{Obj}_{\mathbb{C}}$ a morphism $\eta_a \in \mathsf{Hom}_{\mathbb{D}}(J(a), F(a))$, and
- for all $a, b \in \mathsf{Obj}_{\mathbb{C}}$ and $k \in \mathsf{Hom}_{\mathbb{D}}(J(a), F(b))$ a morphism

$$k^* \in \mathsf{Hom}_{\mathbb{D}}(F(a), F(b))$$

such that

- the *right unit* law
$$k = k^* \circ_{\mathbb{D}} \eta_a,$$
- the *left unit* law
$$\eta_a^* = \mathsf{id}_{\mathbb{D}} \in \mathsf{Hom}_{\mathbb{D}}(F(a), F(a)),$$
- and the *associativity* law
$$(l^* \circ_{\mathbb{D}} k)^* = l^* \circ_{\mathbb{D}} k^*$$

hold for all $a, b, c \in \mathsf{Obj}_{\mathbb{C}}$, $k \in \mathsf{Hom}_{\mathbb{D}}(J(a), F(b))$ and $l \in \mathsf{Hom}_{\mathbb{D}}(J(b), F(c))$.

Note that, although the full definition only requires an object mapping $F : \mathsf{Obj}_{\mathbb{C}} \to \mathsf{Obj}_{\mathbb{D}}$, analogous to a Kleisli triple, a canonical functor can be defined based on F through the following morphism mapping:

$$F(f) \stackrel{\mathsf{def}}{=} (\eta_b \circ_{\mathbb{D}} J(f))^*$$

6.1.9 Categorical representation of constraints

Constraints can be modelled categorically through *concrete categories*:

Definition 6.21 (Concrete category)**.** Let $\mathbb{O}$ be a category. $\mathbb{O}$ is *concrete* iff it has an associated functor $J : \mathbb{O} \longrightarrow \mathsf{Set}$ and J is faithful (Definition 6.22).

If the associated functor J maps to some category $\mathbb{C}$ instead of Set then $\mathbb{O}$ is called *concrete on* $\mathbb{C}$.

Definition 6.22 (Faithful functor)**.** A functor is *faithful* iff its morphism mapping is injective.

The faithfulness of a functor does not imply that the object mapping is injective as well.

If there are constraints on the result type `a` of a functor, applicative, or monadic type `F a` these have to be modelled in the source category $\mathbb{O}$ of the underlying functors. Thus `F` becomes $\texttt{F} : \mathbb{O} \longrightarrow \mathsf{Set}$. The category $\mathbb{O}$ is required to be concrete such that the underlying unconstrained type `a` in Set can be recovered through the associated functor J.

This approach to model constraints can be exemplified by applying it to the definition of a constrained functor for the type `Set`. As explained in Section 3.1.4, `Set` provides a representation of finite sets in Haskell. `Set` is prevented from instantiating a `Functor` instance, because its `map` function imposes an `Ord` constraint, due to its internal representation with balanced binary trees.

```
map :: Ord b => (a -> b) -> Set a -> Set b
```

I have formalised an implementation of `Set` in Agda to give evidence that this suggested categorical model of constrained functors, applicatives, and monads is correct. The formalisation is not an exact reimplementation of `Set` in Agda, but rather an alternate implementation. It uses ordered lists instead of balanced binary trees to remove complexity from the formalisation. I do not deem this difference in implementation a problem, because the implementation with ordered lists is semantically equivalent to the implementation of `Set` in Haskell.

Another difference of the formalisation is that it requires every result type to be ordered not just those that require it due to implementation. Notice that, in the above `map` an ordering is only required for `b`, but not `a`. The morphism mapping resulting from the formalisation would have the following form instead:

```
map :: (Ord a , Ord b) => (a -> b) -> Set a -> Set b
```

I argue that this would be the morally correct way of implementing `Set` in Haskell, because only empty or singleton sets can be constructed if the element type does not have an ordering. Ideally `Set` would enforce the ordering on its elements within the `Set` data type itself, but standard Haskell lacks the ability to do this.

In the formalisation the constrained source category $\mathbb{O}$ has dependent pairs of sets (or types) and their instances of `Ord` as objects. The morphisms are total functions between these sets.

$$\mathsf{Obj}_{\mathbb{O}_{\texttt{Ord}}} \stackrel{\mathsf{def}}{=} \{\ (a, \texttt{Ord}\ a) \mid \forall\ a \in \mathsf{Obj}_{\mathsf{Set}} \text{ that have an instance of } \texttt{Ord}\ \}$$
$$\mathsf{Hom}_{\mathbb{O}_{\texttt{Ord}}}(a,b) \stackrel{\mathsf{def}}{=} \mathsf{Hom}_{\mathsf{Set}}(\mathsf{proj}_1(a), \mathsf{proj}_1(b))$$

The identity morphism and composition from Set are also reused. The associated functor $J_{\mathbb{O}_{\texttt{Ord}}}$ is simply the projection into Set

$$J_{\mathbb{O}_{\texttt{Ord}}} : \begin{cases} \mathbb{O}_{\texttt{Ord}} \longrightarrow \mathsf{Set} \\ a \mapsto \mathsf{proj}_1(a) \\ f \mapsto f \end{cases}$$

which is clearly faithful and therefore $\mathbb{O}_{\texttt{Ord}}$ is concrete. The formalisation of `Set` indeed forms a categorical functor from $\mathbb{O}_{\texttt{Ord}}$ to Set[22] as expected. This functor provides the following morphism mapping

$$\forall\ (a,\ \texttt{Ord}\ a),\ (b,\ \texttt{Ord}\ b) \in \mathsf{Obj}_{\mathbb{O}_{\texttt{Ord}}}.\ (a \to b) \mapsto (\texttt{Set}\ a \to \texttt{Set}\ b)$$

which translates to the mapping function presented above.

To create a `Functor` instance that exactly matches the `map` function provided in the Haskell implementation, a different constraint category could be used where the morphisms are dependent pairs of the form $(f : a \to b,\ \texttt{Ord}\ b)$ and the objects are simply sets.

This technique can be applied to any kind of constraint that can be captured by the `FunctorCts` associated constraints of the constrained variant of the `Functor` class. That the constraints abide the category laws and allow for proper composition only has to be ensured if different constraints apply to `a` and `b`, respectively.

6.2 Monadic notions

This section discusses how the different monadic notions can be modelled categorically. Table 6.1 provides an overview and guide of the categorical models for the monadic notions and highlights their common structure. Note that the formalisation of Haskell functors in Set and Set-based functors in category theory are equivalent[23] to each other. Graded and indexed monads have a whole family of underlying functors instead of just one.

[22] Agda proof: `Theory.Haskell.Constrained.Examples.SetFunctor`
[23] Agda proof: `Theory.Functor.Properties.IsomorphicHaskellFunctor`

Monadic notion	Standard	Graded $\forall i \in M.$	Indexed $\forall i, j \in I.$	Constrained
Underlying functors	$F : \mathbb{C} \longrightarrow \mathbb{C}$	$F_i : \mathbb{C} \longrightarrow \mathbb{C}$	$F_{i,j} : \mathbb{C} \longrightarrow \mathbb{C}$	$F : \mathbb{O} \longrightarrow \mathbb{C}$
Basic model	Monad	—	—	Relative monad
Lax mon. functor	$\mathbf{1} \longrightarrow [\mathbb{C}, \mathbb{C}]_\circ$ $\bullet \mapsto F$ $\mathsf{id}_\bullet \mapsto \mathsf{id}_F$	$\mathsf{Mon}_M \longrightarrow [\mathbb{C}, \mathbb{C}]_\circ$ $i \mapsto F_i$ $\mathsf{id}_i \mapsto \mathsf{id}_{F_i}$	—	– ? –
Lax 2-functor	$\mathbf{1} \longrightarrow \mathsf{Cat}$ $\bullet \mapsto \mathbb{C}$ $\bullet \mapsto F$ $\mathsf{id}_\bullet \mapsto \mathsf{id}_F$	$\mathsf{Mon}_M^2 \longrightarrow \mathsf{Cat}$ $\bullet \mapsto \mathbb{C}$ $i \mapsto F_i$ $\mathsf{id}_i \mapsto \mathsf{id}_{F_i}$	$\mathbb{I} \longrightarrow \mathsf{Cat}$ $i \mapsto \mathbb{C}$ $(i, j) \mapsto F_{i,j}$ $\mathsf{id}_{(i,j)} \mapsto \mathsf{id}_{F_{i,j}}$	– ? –

M – The carrier of a monoid.

$\mathbb{C}$ – The underlying category of the monadic notion.

$\mathbb{O}$ – The category providing the constrained version of $\mathbb{C}$. $\mathbb{O}$ is concrete on $\mathbb{C}$.

I – The set of possible indices.

$\mathbb{I}$ – The strict 2-category with I as 0-cells, codiscrete 1-cells, and discrete 2-cells.

Table 6.1: Overview of categorical formalisations for monadic notions.

6.2.1 Standard monads

Monads, as they are encoded in Haskell, closely resemble Kleisli triples (Section 6.1.7) and are equivalent to standard categorical monads[24] from Set to Set. For brevity the complete definitions of Kleisli triple (Definition 6.19) and monad (Definition 6.18) are not repeated here.

It has to be noted that the natural transformations of a monad $\eta : \mathsf{Id}_{\mathbb{C}} \xrightarrow{\cdot} F$ and $\mu : F \circ F \xrightarrow{\cdot} F$ exactly match the building blocks of a lax monoidal functor. Indeed monads in a category $\mathbb{C}$ are in one-to-one correspondence[25] with lax monoidal functors from the unit category $\mathbf{1}$ to $[\mathbb{C},\mathbb{C}]_{\circ}$.

Proposition 6.23 (Monads $\leftrightarrow$ Lax Monoidal Functors)**.** Let $\mathbb{C}$ be a category:

$$(\text{Monads on } \mathbb{C}) \leftrightarrow (\text{Lax monoidal functors from } \mathbf{1} \text{ to } [\mathbb{C},\mathbb{C}]_{\circ})$$

The object level mapping selects the functor F in $[\mathbb{C},\mathbb{C}]_{\circ}$ and the morphism level mapping maps to the identity natural transformation on F. The η and $\mu_{a,b}$ transformations of the lax monoidal functor are in mapped onto the η and μ transformations of the monad.

6.2.2 Graded monads

Graded monads (Section 3.1.3) do not have a standard categorical model to refer to, but the categorical definition of standard monads from Definition 6.18 can be generalised[26] in a straightforward way.

Definition 6.24 (Graded monad)**.** Let $\mathbb{C}$ be a category and $(M,\diamond,e)$ be a monoid. A *graded monad* consists of

- a family of functors $F_i : \mathbb{C} \longrightarrow \mathbb{C}$ for all $i \in M$,
- a natural transformation $\eta : \mathsf{id}_{\mathbb{C}} \xrightarrow{\cdot} F_e$, and
- a family of natural transformations $\mu^{i,j} : F_i \circ F_j \xrightarrow{\cdot} F_{i \diamond j}$ for all $i,j \in M$

such that the following diagrams for associativity, left identity and right identity commute

$$\begin{array}{ccc} F_i\big(F_j(F_k(c))\big) & \xrightarrow{F_i\left(\mu_c^{j,k}\right)} & F_i\big(F_{j\diamond k}(c)\big) \\ {\scriptstyle \mu^{i,j}_{F_k(c)}}\downarrow & & \downarrow{\scriptstyle \mu_c^{i,j\diamond k}} \\ F_{i\diamond j}(F_k(c)) & \xrightarrow[\mu_c^{i\diamond j,k}]{} & F_{i\diamond j\diamond k}(c) \end{array} \qquad \begin{array}{ccc} F_i(c) & \xrightarrow{F_i(\eta_c)} & F_i(F_e(c)) \\ {\scriptstyle \eta_{F_i}(c)}\downarrow & \searrow\!\searrow & \downarrow{\scriptstyle \mu_c^{i,e}} \\ F_e(F_i(c)) & \xrightarrow[\mu_c^{e,i}]{} & F_i(c) \end{array}$$

for all $i,j,k \in M$ and $c \in \mathsf{Obj}_{\mathbb{C}}$.

[24] Agda proof: `Theory.Monad.Properties.IsomorphicHaskellMonad`
[25] Agda proof: `Theory.Functor.Monoidal.Properties.IsomorphicMonad`
[26] Agda formalisation: `Theory.Haskell.Parameterized.Graded.Monad`

This generalisation is equivalent[27] to the Set-based (and Haskell inspired) definition of graded monads[28]. Just as with standard monads, there is a one-to-one correspondence[29] with lax monoidal functors [Kat14; Gab+16].

Proposition 6.25 (Graded monads $\leftrightarrow$ Lax monoidal functors)**.** Let $\mathbb{C}$ be a category and M be a monoid:

$$\text{Graded monads on } \mathbb{C} \text{ with } M \text{ as monoid}$$
$$\leftrightarrow$$
$$\text{Lax monoidal functors from } \mathsf{Mon}_M \text{ to } [\mathbb{C},\mathbb{C}]_\circ$$

The unit category $\mathbf{1}$ is exchanged with the monoidal category Mon_M based on the monoid of effects M that the graded monad is using. This process yields a lax monoidal functor of the form $\mathsf{Mon}_M \longrightarrow [\mathbb{C},\mathbb{C}]_\circ$. On the object level each element of M is mapped to the functor with that element as index and the identity morphisms are mapped to the identity transformation on the corresponding functor.

6.2.3 Indexed monads

Indexed monads (Section 3.1.2) do not have a standard categorical model either, but, as with graded monads, Definition 6.18 can be generalised[30] to obtain a suitable definition.

Definition 6.26 (Parameterised monad)**.** Let $\mathbb{C}$ and $\mathbb{I}$ be categories. $\mathbb{I}$ is the category of indices. A *parameterised monad* consists of

- a family of functors $F_f : \mathbb{C} \longrightarrow \mathbb{C}$ for all $i, j \in \mathsf{Obj}_\mathbb{I}$ and $f \in \mathsf{Hom}_\mathbb{I}(i,j)$,
- a family of natural transformations $\eta^i : \mathsf{id}_\mathbb{C} \dot{\longrightarrow} F_{\mathsf{id}_i}$ for all $i \in \mathsf{Obj}_\mathbb{I}$, and
- a family of natural transformations $\mu^{f,g} : F_g \circ F_f \dot{\longrightarrow} F_{g \circ_\mathbb{I} f}$ for all $i,j,k \in \mathsf{Obj}_\mathbb{I}$, $f \in \mathsf{Hom}_\mathbb{I}(i,j)$ and $g \in \mathsf{Hom}_\mathbb{I}(j,k)$

such that the following diagrams for associativity, left identity and right identity commute

$$\begin{array}{ccc} F_h\big(F_g\big(F_f(c)\big)\big) & \xrightarrow{F_h\left(\mu_c^{f,g}\right)} & F_h\big(F_{g\circ_\mathbb{I} f}(c)\big) \\ {\scriptstyle \mu^{g,h}_{F_f(c)}}\downarrow & & \downarrow{\scriptstyle \mu_c^{g\circ_\mathbb{I} f,h}} \\ F_{h\circ_\mathbb{I} g}\big(F_f(c)\big) & \xrightarrow[\mu_c^{f,h\circ_\mathbb{I} g}]{} & F_{h\circ_\mathbb{I} g\circ_\mathbb{I} f}(c) \end{array} \qquad \begin{array}{ccc} F_f(c) & \xrightarrow{F_f\left(\eta_c^i\right)} & F_f(F_{\mathsf{id}_i}(c)) \\ {\scriptstyle \eta^j_{F_f}(c)}\downarrow & \searrow\!\searrow & \downarrow{\scriptstyle \mu_c^{\mathsf{id}_i,f}} \\ F_{\mathsf{id}_j}\big(F_f(c)\big) & \xrightarrow[\mu_c^{f,\mathsf{id}_i}]{} & F_f(c) \end{array}$$

for all $i,j,k,l \in \mathsf{Obj}_\mathbb{I}$, $f \in \mathsf{Hom}_\mathbb{I}(i,j)$, $g \in \mathsf{Hom}_\mathbb{I}(j,k)$ and $h \in \mathsf{Hom}_\mathbb{I}(k,l)$.

[27] Agda proof: `Theory.Haskell.Parameterized.Graded.Monad.Properties.IsomorphicHaskellGradedMonad`

[28] Agda formalisation: `Haskell.Parameterized.Graded.Monad`

[29] Agda proof: `Theory.Functor.Monoidal.Properties.IsomorphicGradedMonad`

[30] Agda formalisation: `Theory.Haskell.Parameterized.Indexed.Monad`

Note that the indices are provided by the morphisms of a category instead of a simple set. Also notice that this definition has a structure similar to that of a graded monad. Indeed, if Mon_M (for any monoid M) is used as the category of indices both definitions are in one-to-one correspondence[31] with each other.

Proposition 6.27 (Graded monads $\leftrightarrow$ Parameterised monads)**.** Let $\mathbb{C}$ be a category and M be a monoid:

$$\text{Graded monad on } \mathbb{C} \text{ with monoid } M$$
$$\leftrightarrow$$
$$\text{Parameterised monad on } \mathbb{C} \text{ with index category } \mathsf{Mon}_M$$

The new definition is called a parameterised monad instead of indexed monad, because it captures standard, graded and indexed monads alike.

As before this definition is equivalent[32] to the Set-based (and Haskell inspired) definition[33] of indexed monads. This definition is *not* in one-to-one correspondence with a lax monoidal functor, because the indices form a general category rather than having a monoidal structure. To find a matching categorical notion the more general structure of a lax 2-functor is required. The category of possible indices $\mathbb{I}$ needs to be expanded to a 2-category with discrete 2-cells. It then forms the strict 2-category of $\mathsf{Obj}_{\mathbb{I}}$ 0-cells, $\mathsf{Hom}_{\mathbb{I}}(-,-)$ 1-cells, and discrete 2-cells. The corresponding lax 2-functor of a parameterised monad maps 0-cells to the underlying category $\mathbb{C}$ of the parameterised monad. Each 1-cells selects the corresponding functor from the family of underlying functors. The 2-cells are mapped onto the corresponding identity transformations of those functors. Just as with the lax monoidal functors in the previous paragraphs, the η_a and $\mu_{a,b,c}$ transformations of the lax 2-functor are in exact correspondence with the return and join operation of the parameterised monad[34].

Proposition 6.28 (Parameterised monads $\leftrightarrow$ Lax 2-functors)**.** Let $\mathbb{C}$ and $\mathbb{I}$ be categories:

$$\text{Parameterised monads on } \mathbb{C} \text{ with index category } \mathbb{I}$$
$$\leftrightarrow$$
$$\text{Lax 2-functors from } \mathsf{Disc}_{\mathbb{I}} \text{ to } \mathsf{Cat} \text{ that map all 0-cells to } \mathbb{C}$$

Note that lax 2-functors capture graded and standard monads by composing Proposition 6.27 with Proposition 6.28 and also via the connection between lax monoidal functors and lax 2-functors presented in Example 6.17.

[31] Agda proof: `Theory.Haskell.Parameterized.Indexed.Monad.Properties.IsomorphicGradedMonad`

[32] Agda proof: `Theory.Haskell.Parameterized.Indexed.Monad.Properties.IsomorphicHaskellIndexedMonad`

[33] Agda formalisation: `Haskell.Parameterized.Indexed.Monad`

[34] Agda proof: `Theory.TwoFunctor.Properties.IsomorphicIndexedMonad`

I am aware that Atkey [Atk09] gave a different categorical model for parameterised monads. However, the above definition of parameterised monad is slightly more general and connects with lax 2-functors (next section) naturally. Atkey's work is discussed in Section 7.1.

6.2.4 Unified categorical representation of parameterised monads

A lax monoidal functor can always be "lifted" into a lax 2-functor, i.e., for any category $\mathbb{C}$ and monoid M, if the monoidal functor goes from Mon_M to $[\mathbb{C},\mathbb{C}]_\circ$ then it is equivalent[35] to a lax 2-functors from Mon^2_M to Cat where the 0-cell mapping is constant to $\mathbb{C}$ (Example 6.17). Since the lax monoidal functors of standard[36] and graded[37] monads fulfil these conditions (Proposition 6.23 and 6.25) they are in one-to-one correspondence with the lax 2-functors of the corresponding structure.

Due to this correspondence there is a common categorical representation of standard, graded and indexed monads as lax 2-functors that use a constant mapping as their 0-cell mapping.

Note, that the generalisation of the categorical definition of standard monads to parameterised monads provides a categorical model that is independent of Set and that captures standard, graded, and indexed monads more precisely and intuitively than lax 2-functors. Although these generalisations are not standard categorical notions I deem them useful, especially for functional programmers that have limited familiarity with category theory.

6.2.5 Constrained monads

Relative monads can be used to model constrained monads in the same way as described for functors in Section 6.1.9. A given category of constraints $\mathbb{O}$ that is concrete on $\mathbb{C}$ can use the associated functor $J_{\mathbb{O}}$ as the functor J from the definition of relative monads (6.20). Thus, a constrained monad can be defined as a relative monad from $\mathbb{O}$ to the underlying category $\mathbb{C}$. Hence, given an object mapping F the morphisms η_a and the Kleisli extension $(-)^*$ still need to be defined.

The `Set` example from Section 6.1.9 provides a good example for this process. In the `Set` example the underlying category was Set. The concrete category $\mathbb{O}_{\mathtt{Ord}}$ is the source category of the relative monad and its associated functor $J_{\mathbb{O}_{\mathtt{Ord}}}$

[35] Agda proof: `Theory.TwoFunctor.Properties.IsomorphicLaxMonoidalFunctor`
[36] Agda proof: `Theory.TwoFunctor.Properties.IsomorphicMonad`
[37] Agda proof: `Theory.TwoFunctor.Properties.IsomorphicGradedMonad`

provides J. This results in the following morphisms for η_a

$$\begin{aligned}&\forall\ (a,\ \texttt{Ord}\ a) \in \mathsf{Obj}_{\mathbb{O}_{\texttt{Ord}}}.\ J_{\mathbb{O}_{\texttt{Ord}}}(a,\ \texttt{Ord}\ a) \to F(a,\ \texttt{Ord}\ a)\\ &\qquad\qquad =\\ &\forall\ (a,\ \texttt{Ord}\ a) \in \mathsf{Obj}_{\mathbb{O}_{\texttt{Ord}}}.\ a \to F(a,\ \texttt{Ord}\ a)\end{aligned}$$

and the following mappings for the Kleisli extension

$$\begin{aligned}&\forall\ (a,\ \texttt{Ord}\ a), (b,\ \texttt{Ord}\ b) \in \mathsf{Obj}_{\mathbb{O}_{\texttt{Ord}}}.\\ &\quad (J_{\mathbb{O}_{\texttt{Ord}}}(a,\ \texttt{Ord}\ a) \to F(b,\ \texttt{Ord}\ b)) \ \mapsto\ (F(a,\ \texttt{Ord}\ a) \to F(b,\ \texttt{Ord}\ b))\\ &\qquad\qquad =\\ &\forall\ (a,\ \texttt{Ord}\ a), (b,\ \texttt{Ord}\ b) \in \mathsf{Obj}_{\mathbb{O}_{\texttt{Ord}}}.\\ &\quad (a \to F(b,\ \texttt{Ord}\ b)) \ \mapsto\ (F(a,\ \texttt{Ord}\ a) \to F(b,\ \texttt{Ord}\ b)).\end{aligned}$$

which reflect a constrained return and bind operation. The formalisation of `Set` indeed forms a relative monad in this way[38].

Note that the Haskell representation of constrained monads has separate constraints for the return and bind operation, i.e., `ReturnCts` and `BindCts`. This is due to the necessary split of the monad type class into two type classes. When instantiating a constrained monad instance the programmer needs to make sure that `BindCts` and `ReturnsCts` are consistent according to the definition of a relative monad.

Unfortunately, I have not yet found a pre-existing categorical structure that captures both lax 2-functors and relative monads at the same time. As pointed out by Altenkirch, Chapman, and Uustalu [ACU10] there is no obvious way to form a join operation, i.e., a transformation $F \circ F \dot{\longrightarrow} F$, as would be required by the lax 2-functors that are equivalent to the parameterised notions. Their paper does show that relative monads in $[\mathbb{J},\mathbb{C}]$ (the category of functors from $\mathbb{J}$ to $\mathbb{C}$) can be transformed into lax monoidal functors in $[\mathbb{C},\mathbb{C}]$, if the left Kan extension $[\mathbb{J},\mathbb{C}] \longrightarrow [\mathbb{C},\mathbb{C}]$ exists. Unfortunately, this transformation is not applicable to the relative monads that I construct to model constrained monads, because the associated functor J is just faithful but not fully faithful as is required to apply the transformation. Future work could explore if and under which conditions the relative monads presented here can be transformed.

Even though I could not find a standard categorical notion that captures lax 2-functors and relative monads, the same technique of generalisation that was used to define parameterised monads can be applied to relative monads. This leads to the notion of a parameterised relative monad.

Definition 6.29 (Parameterised relative monad)**.** Let $\mathbb{C}$, $\mathbb{D}$ and $\mathbb{I}$ be categories. $\mathbb{I}$ is the category of indices. A *parameterised relative monad* consists of

- a functor $J : \mathbb{C} \longrightarrow \mathbb{D}$,

[38] Agda proof: `Theory.Haskell.Constrained.Examples.SetMonad`

- an object mapping $F_f : \mathsf{Obj}_{\mathbb{C}} \to \mathsf{Obj}_{\mathbb{D}}$ for all $i, j \in \mathsf{Obj}_{\mathbb{I}}$ and $f \in \mathsf{Hom}_{\mathbb{I}}(i, j)$,
- for all $a \in \mathsf{Obj}_{\mathbb{C}}$ and $i \in \mathsf{Obj}_{\mathbb{I}}$ a morphism

$$\eta_a^i \in \mathsf{Hom}_{\mathbb{D}}(J(a), F_{\mathsf{id}_i}(a)), \quad \text{and}$$

- for all $a, b \in \mathsf{Obj}_{\mathbb{C}}$, $i, j, k \in \mathsf{Obj}_{\mathbb{I}}$, $f \in \mathsf{Hom}_{\mathbb{I}}(i, j)$, $g \in \mathsf{Hom}_{\mathbb{I}}(j, k)$ and $k \in \mathsf{Hom}_{\mathbb{D}}\big(J(a), F_f(b)\big)$ a morphism

$$k^*_{f,g} \in \mathsf{Hom}_{\mathbb{D}}\big(F_g(a), F_{g \circ_{\mathbb{I}} f}(b)\big)$$

that satisfy the following laws:

- for all $a, b \in \mathsf{Obj}_{\mathbb{C}}$, $i, j \in \mathsf{Obj}_{\mathbb{I}}$, $f \in \mathsf{Hom}_{\mathbb{I}}(i, j)$ and $k \in \mathsf{Hom}_{\mathbb{D}}\big(J(a), F_f(b)\big)$ the *right unit* law

$$k = k^*_{f,\mathsf{id}_j} \circ_{\mathbb{D}} \eta_a^j,$$

- for all $a \in \mathsf{Obj}_{\mathbb{C}}$, $i, j \in \mathsf{Obj}_{\mathbb{I}}$ and $f \in \mathsf{Hom}_{\mathbb{I}}(i, j)$ the *left unit* law

$$\big(\eta_a^i\big)^*_{\mathsf{id}_i, f} = \mathsf{id}_{F_f(a)} \in \mathsf{Hom}_{\mathbb{D}}\big(F_f(a), F_f(a)\big), \quad \text{and}$$

- for all $a, b, c \in \mathsf{Obj}_{\mathbb{C}}$, $s, t, u, v \in \mathsf{Obj}_{\mathbb{I}}$, $f \in \mathsf{Hom}_{\mathbb{I}}(s, t)$, $g \in \mathsf{Hom}_{\mathbb{I}}(t, u)$, $h \in \mathsf{Hom}_{\mathbb{I}}(u, v)$, $k \in \mathsf{Hom}_{\mathbb{D}}(J(a), F_g(b))$ and $l \in \mathsf{Hom}_{\mathbb{D}}\big(J(b), F_f(c)\big)$ the *associativity* law

$$(l^*_{f,g} \circ_{\mathbb{D}} k)^*_{g \circ_{\mathbb{I}} f, h} = l^*_{f, h \circ_{\mathbb{I}} g} \circ_{\mathbb{D}} k^*_{g,h}.$$

As expected the notion of parameterised relative monads is in one-to-one correspondence[39] with a parameterised monad if $\mathbb{C}$ and $\mathbb{D}$ are the same.

Proposition 6.30 (Parameterised monads $\leftrightarrow$ Parameterised relative monads)**.** Let $\mathbb{C}$ and $\mathbb{I}$ be categories:

$$\begin{gathered}\text{Parameterised monads on } \mathbb{C} \text{ with index category } \mathbb{I} \\ \leftrightarrow \\ \text{Parameterised relative monads between } \mathbb{C} \text{ and } \mathbb{C} \\ \text{with index category } \mathbb{I} \text{ and } J = \mathsf{Id}_{\mathbb{C}}\end{gathered}$$

If $\mathbb{I}$ is the unit category the definition of parameterised relative monads is in one-to-one correspondence[40] with that of relative monads.

[39] Agda proof: `Theory.Haskell.Parameterized.Relative.Monad.Properties.IsomorphicIndexedMonad`

[40] Agda proof: `Theory.Haskell.Parameterized.Relative.Monad.Properties.IsomorphicRelativeMonad`

Applicative notion	Standard	Graded $\forall i \in M.$	Indexed $\forall i, j.$	Constrained
Underlying functors	$F : \mathbb{C} \longrightarrow \mathbb{C}$	$F_i : \mathbb{C} \longrightarrow \mathbb{C}$	$F_{i,j} : \mathbb{C} \longrightarrow \mathbb{C}$	$F : \mathbb{O} \longrightarrow \mathbb{C}$
Lax mon. functor	$\mathbb{C} \longrightarrow \mathbb{C}$	$\mathsf{Mon}_M \times \mathbb{C} \longrightarrow \mathbb{C}$	—	$\mathbb{O} \longrightarrow \mathbb{C}$
Lax mon. functor (*conjectured*)	$\mathbf{1} \longrightarrow [\mathbb{C}, \mathbb{C}]_{\text{Day}}$ $\bullet \mapsto F$ $\mathsf{id}_\bullet \mapsto \mathsf{id}_F$	$\mathsf{Mon}_M \longrightarrow [\mathbb{C}, \mathbb{C}]_{\text{Day}}$ $i \mapsto F_i$ $\mathsf{id}_i \mapsto \mathsf{id}_{F_i}$	—	$\mathbf{1} \longrightarrow [\mathbb{O}, \mathbb{C}]_{\text{Day}}$ $\bullet \mapsto F$ $\mathsf{id}_\bullet \mapsto \mathsf{id}_F$

M – The carrier of a monoid.

$\mathbb{C}$ – The underlying category of the applicative notions.

$\mathbb{O}$ – The category providing the constrained version of $\mathbb{C}$. $\mathbb{O}$ is concrete on $\mathbb{C}$.

Table 6.2: Overview of categorical formalisations for applicative notions.

Proposition 6.31 (Relative monads $\leftrightarrow$ Parameterised relative monads)**.** Let $\mathbb{C}$ and $\mathbb{D}$ be categories. Let $T : \mathsf{Obj}_{\mathbb{C}} \to \mathsf{Obj}_{\mathbb{D}}$ be an object mapping and let $J : \mathbb{C} \longrightarrow \mathbb{D}$ be a functor.

Relative monads from $\mathbb{C}$ to $\mathbb{D}$ with object mapping T and associated functor J

$\leftrightarrow$

Parameterised relative monads from $\mathbb{C}$ to $\mathbb{D}$

with index category $\mathbf{1}$, object mapping T and associated functor J

Due to their relationship with parameterised monads, parameterised relative monads are in one-to-one correspondence[41] with certain lax 2-functors if $\mathbb{C}$ and $\mathbb{D}$ are equal and the 0-cell mapping of the lax 2-functor is constant (Proposition 6.28 and 6.30). Hence, parameterised relative monads provide a custom categorical notion that is able to provide a model for all of the monadic notions presented in Section 3.1 (given their Set-based representation).

Unpublished work by Orchard and Mycroft [OM12] discusses using relative monads as a categorical model for constrained monads in a similar manner as was presented here.

6.3 Applicative notions

As can be seen in Table 6.2, all generalisations of applicatives are based on the same underlying functors as the previously presented monadic notions.

[41] Agda proof: `Theory.TwoFunctor.Properties.IsomorphicParameterizedRelativeMonad`

6.3.1 Standard applicatives

Standard applicatives in Set are equivalent[42] to lax monoidal functors [MP08]. The morphism η of a the lax monoidal functor $F : \mathsf{Set} \longrightarrow \mathsf{Set}$ that is represented by a standard applicative `F` is given by

$$\eta : \begin{cases} \top \to F(\top) \\ \texttt{()} \mapsto \texttt{pure ()} \end{cases}$$

and the natural transformation $\mu_{a,b}$ is given by

$$\mu_{a,b} : \begin{cases} F(a) \times F(b) \to F(a \times b) \\ (\texttt{u}, \texttt{v}) \mapsto \texttt{fmap (\x y -> (x,y)) u <*> v.} \end{cases}$$

In contrast to monads, applicatives do not require their underlying functors to be endofunctors. This is key to model graded and constrained applicatives.

6.3.2 Graded applicatives

Graded applicatives (Section 3.2.3) can be represented by "adding in" the monoid $(M, \diamond, e)$ of effects that they are parameterised over. This is achieved by using the product monoidal category of Mon_M and Set as the source category of the monoidal functor:

$$\mathsf{Mon}_M \times \mathsf{Set} \longrightarrow \mathsf{Set}$$

This monoidal functor provides the required operations

$$\begin{aligned} \eta &: (e, \top) \to F(e, \top) \\ \mu_{a,b} &: F(i,a) \times F(j,b) \stackrel{\cdot}{\longrightarrow} F(i \diamond j, a \times b) \end{aligned}$$

to form a graded applicative and is indeed equivalent[43] to one:

```
pure ::  a -> F (Unit F) a
pure a = fmap (\() -> a) (η ())

(<*>) ::  F i (a -> b) -> F j a -> F (Comp F i j) b
ff <*> fa = fmap (\(f , x) -> f x) (μ_{a,b} (ff , fa))
```

Although not necessary the definition of a lax monoidal functor can be generalised in a similar manner as was done for graded monads:

Definition 6.32 (Graded lax monoidal functor)**.** Let $(\mathbb{C}, 1_{\mathbb{C}}, \otimes_{\mathbb{C}})$ and $(\mathbb{D}, 1_{\mathbb{D}}, \otimes_{\mathbb{D}})$ be two monoidal categories. Let $(M, \diamond, e)$ be a monoid. A *graded lax monoidal functor* between $\mathbb{C}$ and $\mathbb{D}$ consists of

[42] Agda proof: `Theory.Functor.Monoidal.Properties.IsomorphicHaskellApplicative`
[43] Agda proof: `Theory.Functor.Monoidal.Properties.IsomorphicGradedApplicative`

- a family of functors $F_i : \mathbb{C} \longrightarrow \mathbb{D}$ for all $i \in M$,
- a morphism $\eta : 1_{\mathbb{D}} \to F_e(1_{\mathbb{C}})$, and
- a family of natural transformations

$$\mu^{i,j}_{a,b} : F_i(a) \otimes_{\mathbb{D}} F_j(b) \xrightarrow{\cdot} F_{i \diamond j}(a \otimes_{\mathbb{C}} b)$$

  for all $i, j \in M$ and $a, b \in \mathsf{Obj}_{\mathbb{C}}$,

such that the diagram for *associativity*

$$\begin{array}{ccc}
(F_i(a) \otimes_{\mathbb{D}} F_j(b)) \otimes_{\mathbb{D}} F_k(c) & \xrightarrow{\alpha^{\mathbb{D}}_{F_i(a),F_j(b),F_k(c)}} & F_i(a) \otimes_{\mathbb{D}} (F_j(b) \otimes_{\mathbb{D}} F_k(c)) \\
\downarrow{\scriptstyle \mu^{i,j}_{a,b} \otimes_{\mathbb{D}} \mathsf{id}_{F_k(c)}} & & \downarrow{\scriptstyle \mathsf{id}_{F_i(a)} \otimes_{\mathbb{D}} \mu^{j,k}_{b,c}} \\
F_{i \diamond j}(a \otimes_{\mathbb{C}} b) \otimes_{\mathbb{D}} F_k(c) & & F_i(a) \otimes_{\mathbb{D}} F_{j \diamond k}(b \otimes_{\mathbb{C}} c) \\
\downarrow{\scriptstyle \mu^{i \diamond j, k}_{a \otimes_{\mathbb{C}} b, c}} & & \downarrow{\scriptstyle \mu^{i, j \diamond k}_{a, b \otimes_{\mathbb{C}} c}} \\
F_{(i \diamond j) \diamond k}((a \otimes_{\mathbb{C}} b) \otimes_{\mathbb{C}} c) & \xrightarrow[F_{i \diamond j \diamond k}(\alpha^{\mathbb{C}}_{a,b,c})]{} & F_{i \diamond (j \diamond k)}(a \otimes_{\mathbb{C}} (b \otimes_{\mathbb{C}} c))
\end{array}$$

and the diagrams for *left* and *right unitality*

$$\begin{array}{ccc}
1_{\mathbb{D}} \otimes_{\mathbb{D}} F_i(a) & \xrightarrow{\eta \otimes_{\mathbb{D}} \mathsf{id}_a} & F_e(1_{\mathbb{C}}) \otimes_{\mathbb{D}} F_i(a) \\
\downarrow{\scriptstyle \lambda^{\mathbb{D}}_{F_i(a)}} & & \downarrow{\scriptstyle \mu^{e,i}_{1_{\mathbb{C}},a}} \\
F_i(a) & \xleftarrow[F_i(\lambda^{\mathbb{C}}_a)]{i \overset{\mathsf{def}}{=} e \diamond i} & F_{e \diamond i}(1_{\mathbb{C}} \otimes_{\mathbb{C}} a)
\end{array}$$

$$\begin{array}{ccc}
F_i(a) \otimes_{\mathbb{D}} 1_{\mathbb{D}} & \xrightarrow{\mathsf{id}_a \otimes_{\mathbb{D}} \eta} & F_i(a) \otimes_{\mathbb{D}} F_e(1_{\mathbb{C}}) \\
\downarrow{\scriptstyle \rho^{\mathbb{D}}_{F_i(a)}} & & \downarrow{\scriptstyle \mu^{i,e}_{a,1_{\mathbb{C}}}} \\
F_i(a) & \xleftarrow[F_i(\rho^{\mathbb{C}}_a)]{i \overset{\mathsf{def}}{=} i \diamond e} & F_{i \diamond e}(a \otimes_{\mathbb{C}} 1_{\mathbb{C}})
\end{array}$$

commute for all $i, j, k \in M$ and $a, b, c \in \mathsf{Obj}_{\mathbb{C}}$.

This definition is in one-to-one correspondence[44] with any lax monoidal functor of the form $\mathsf{Mon}_M \times \mathbb{C} \longrightarrow \mathbb{D}$.

Proposition 6.33 (Lax monoidal functors $\leftrightarrow$ Graded lax monoidal functors)**.** Let $\mathbb{C}$ and $\mathbb{D}$ be monoidal categories and let M be a monoid:

Lax monoidal functors from $\mathsf{Mon}_M \times \mathbb{C}$ to $\mathbb{D}$

$\leftrightarrow$

Graded lax monoidal functors from $\mathbb{C}$ to $\mathbb{D}$ with monoid M

[44] Agda proof: `Theory.Haskell.Parameterized.Graded.LaxMonoidalFunctor.Properties.IsomorphicLaxMonoidalFunctor`

6.3.3 Constrained applicatives

For constrained applicatives (Section 3.2.4) a concrete category that models the constraints is used as source category for the lax monoidal functor. This works analogous to the modelling of constrained functors and monads described in the previous sections. The difference is that the concrete category is now also required to be monoidal. This means, that the constraints need to obey the laws that are expected for a monoidal category.

Again `Set` provides a good example for this. In the context of the `Set` example the category $\mathbb{O}_{\texttt{Ord}}$ is the source of the lax monoidal functor. As a result of the functor being monoidal, a tensor product has to exist for the constraints:

$$\otimes : \begin{cases} \mathbb{O}_{\texttt{Ord}} \times \mathbb{O}_{\texttt{Ord}} \longrightarrow \mathbb{O}_{\texttt{Ord}} \\ ((a,\ \texttt{Ord}\ a),\ (b,\ \texttt{Ord}\ b)) \mapsto (a \times b,\ \texttt{Ord}\ (a \times b)) \\ (f : a \to b,\ g : c \to d) \mapsto ((\lambda(x,\ y).\ (f(x),\ g(y))) : a \times c \to b \times d) \end{cases}$$

In addition, a tensor unit is required, i.e., $(\top,\ \texttt{Ord}\ \top)$ is required to be in $\mathsf{Obj}_{\mathbb{O}_{\texttt{Ord}}}$. Finally, it also needs to be ensured that the associator, left unitor and right unitor can be defined and obey triangle and pentagon identity. The formalisation of `Set` forms a lax monoidal functor from $\mathbb{O}_{\texttt{Ord}}$ to Set as expected[45].

Given the lax monoidal functor that represents a standard, graded or constrained applicative I *conjecture* that each representation is equivalent to a lax monoidal functor from an appropriate indexing category to the monoidal functor category that uses day convolution as tensor product. This would give the applicative notions a lax monoidal structure similar to that of the monadic notions.

I only conjecture the equivalence between the two different lax monoidal representations, because the formalisation of day convolution requires more advanced support for quotient types and equalities in Agda.

6.3.4 Indexed applicatives

Unfortunately, just as their monadic counterpart, indexed applicatives (Section 3.2.2) do not match the structure of a lax monoidal functor. I suspect that there is a more abstract structure similar to that of a lax 2-functor that captures all of the applicative notions discussed previously.

As a first approximation of this more general structure the definition of a lax monoidal functors can be generalised in a similar manner as the definitions of monad was generalised to parameterised monad:

Definition 6.34 (Parameterised lax monoidal functor)**.** Let $(\mathbb{C}, 1_{\mathbb{C}}, \otimes_{\mathbb{C}})$ and $(\mathbb{D}, 1_{\mathbb{D}}, \otimes_{\mathbb{D}})$ be two monoidal categories. Let $\mathbb{I}$ be a category. A *parameterised lax monoidal functor* between $\mathbb{C}$ and $\mathbb{D}$ consists of

- a family of functors $F_f : \mathbb{C} \longrightarrow \mathbb{D}$ for every $f \in \mathsf{Hom}_{\mathbb{I}}(i, j)$,

[45] Agda proof: `Theory.Haskell.Constrained.Examples.SetApplicative`

- a morphism $\eta : 1_{\mathbb{D}} \to F_{\mathsf{id}_i}(1_{\mathbb{C}})$ for every $i \in \mathsf{Obj}_{\mathbb{I}}$, and
- a family of natural transformations

$$\mu^{f,g}_{a,b} : F_f(a) \otimes_{\mathbb{D}} F_g(b) \xrightarrow{\cdot} F_{g \circ_{\mathbb{I}} f}(a \otimes_{\mathbb{C}} b)$$

for all $i, j, k \in \mathsf{Obj}_{\mathbb{I}}$, $f \in \mathsf{Hom}_{\mathbb{I}}(i, j)$, $g \in \mathsf{Hom}_{\mathbb{I}}(j, k)$ and $a, b \in \mathsf{Obj}_{\mathbb{C}}$,

such that the diagram for *associativity*

$$\begin{array}{ccc}
(F_f(a) \otimes_{\mathbb{D}} F_g(b)) \otimes_{\mathbb{D}} F_h(c) & \xrightarrow{\alpha^{\mathbb{D}}_{F_f(a),F_g(b),F_k(c)}} & F_i(a) \otimes_{\mathbb{D}} (F_j(b) \otimes_{\mathbb{D}} F_k(c)) \\
\downarrow{\scriptstyle \mu^{f,g}_{a,b} \otimes_{\mathbb{D}} \mathsf{id}_{F_k(c)}} & & \downarrow{\scriptstyle \mathsf{id}_{F_i(a)} \otimes_{\mathbb{D}} \mu^{g,h}_{b,c}} \\
F_{g \circ f}(a \otimes_{\mathbb{C}} b) \otimes_{\mathbb{D}} F_k(c) & & F_f(a) \otimes_{\mathbb{D}} F_{h \circ g}(b \otimes_{\mathbb{C}} c) \\
\downarrow{\scriptstyle \mu^{g \circ f,h}_{a \otimes_{\mathbb{C}} b,c}} & & \downarrow{\scriptstyle \mu^{f,h \circ g}_{a,b \otimes_{\mathbb{C}} c}} \\
F_{h \circ (g \circ f)}((a \otimes_{\mathbb{C}} b) \otimes_{\mathbb{C}} c) & \xrightarrow[F_{h \circ g \circ f}(\alpha^{\mathbb{C}}_{a,b,c})]{} & F_{(h \circ g) \circ f}(a \otimes_{\mathbb{C}} (b \otimes_{\mathbb{C}} c))
\end{array}$$

and the diagrams for *left* and *right unitality*

$$\begin{array}{ccc}
1_{\mathbb{D}} \otimes_{\mathbb{D}} F_f(a) & \xrightarrow{\eta \otimes_{\mathbb{D}} \mathsf{id}_a} & F_{\mathsf{id}_i}(1_{\mathbb{C}}) \otimes_{\mathbb{D}} F_f(a) \\
\downarrow{\scriptstyle \lambda^{\mathbb{D}}_{F_f(a)}} & & \downarrow{\scriptstyle \mu^{\mathsf{id}_i,f}_{1_{\mathbb{C}},a}} \\
F_f(a) & \xleftarrow[F_f(\lambda^{\mathbb{C}}_a)]{f \overset{\mathsf{def}}{=} f \circ \mathsf{id}_i} & F_{f \circ \mathsf{id}_i}(1_{\mathbb{C}} \otimes_{\mathbb{C}} a)
\end{array}$$

$$\begin{array}{ccc}
F_f(a) \otimes_{\mathbb{D}} 1_{\mathbb{D}} & \xrightarrow{\mathsf{id}_a \otimes_{\mathbb{D}} \eta} & F_f(a) \otimes_{\mathbb{D}} F_{\mathsf{id}_j}(1_{\mathbb{C}}) \\
\downarrow{\scriptstyle \rho^{\mathbb{D}}_{F_f(a)}} & & \downarrow{\scriptstyle \mu^{f,\mathsf{id}_j}_{a,1_{\mathbb{C}}}} \\
F_f(a) & \xleftarrow[F_f(\rho^{\mathbb{C}}_a)]{f \overset{\mathsf{def}}{=} \mathsf{id}_j \circ f} & F_{\mathsf{id}_j \circ f}(a \otimes_{\mathbb{C}} 1_{\mathbb{C}})
\end{array}$$

commute for all $i, j, k, l \in \mathsf{Obj}_{\mathbb{I}}$, $f \in \mathsf{Hom}_{\mathbb{I}}(i, j)$, $g \in \mathsf{Hom}_{\mathbb{I}}(j, k)$, $h \in \mathsf{Hom}_{\mathbb{I}}(k, l)$ and $a, b, c \in \mathsf{Obj}_{\mathbb{C}}$.

This structure is equivalent[46] to the Set-based definition of indexed applicatives if a codiscrete category is used as the index category $\mathbb{I}$. It is also in one-to-one correspondence[47] with standard lax monoidal functors if $\mathbb{I}$ is the unit category.

[46] Agda proof: `Theory.Haskell.Parameterized.Indexed.LaxMonoidalFunctor.Properties.IsomorphicHaskellIndexedApplicative`

[47] Agda proof: `Theory.Haskell.Parameterized.Indexed.LaxMonoidalFunctor.Properties.IsomorphicLaxMonoidalFunctor`

Proposition 6.35 (Lax monoidal functors $\leftrightarrow$ Parameterised lax monoidal functors)**.** Let $\mathbb{C}$ and $\mathbb{D}$ be monoidal categories:

$$\text{Lax monoidal functors from } \mathbb{C} \text{ to } \mathbb{D}$$
$$\leftrightarrow$$
$$\text{Parameterised lax monoidal functors from } \mathbb{C} \text{ to } \mathbb{D} \text{ with index category } \mathbf{1}$$

The definition is also in one-to-one correspondence[48] with the definition of graded lax monoidal functors if $\mathbb{I}$ is the category Mon_M formed by a monoid M.

Proposition 6.36 (Graded lax monoidal functors $\leftrightarrow$ Parameterised lax monoidal functors)**.** Let $\mathbb{C}$ and $\mathbb{D}$ be monoidal categories and let M be a monoid:

$$\text{Graded lax monoidal functors from } \mathbb{C} \text{ to } \mathbb{D} \text{ with monoid } M$$
$$\leftrightarrow$$
$$\text{Parameterised lax monoidal functors from } \mathbb{C} \text{ to } \mathbb{D} \text{ with index category } \mathsf{Mon}_M$$

Thus, the definition of parameterised lax monoidal functor captures all of the presented applicative notions.

Exploring if there is a more mature pre-existing categorical notion that captures standard, constrained, graded, and indexed applicatives at the same time remains future work.

6.4 Conclusion of categorical models

In conclusion, the categorical models presented in this section have made a considerable step towards a unified categorical model for different generalisations of monads and applicatives. The notion of *parameterised relative monad* (Definition 6.29) captures the standard, graded, indexed, and constrained variations of the monadic notions and the *parameterised lax monoidal functor* (Definition 6.34) captures the respective variations of the applicative notions.

A result of this work is a hierarchy of monadic and applicative notions. Figure 6.1 contains the hierarchy of models for the monadic notions and Figure 6.2 contains the hierarchy of models for the applicative notions. All of the one-to-one correspondences proven by the propositions in the previous sections provide a bijection between certain instances of different notions. Each of these bijections provides an injection that is presented as an arrow in the hierarchies of Figure 6.1 and 6.2.

Even though an overarching pre-existing categorical model is still missing for either notion (as indicated by the question marks in the diagrams), the presented work contributes a considerable step towards a unified theory of these notions.

[48] Agda proof: `Theory.Haskell.Parameterized.Indexed.LaxMonoidalFunctor.Properties.IsomorphicGradedLaxMonoidalFunctor`

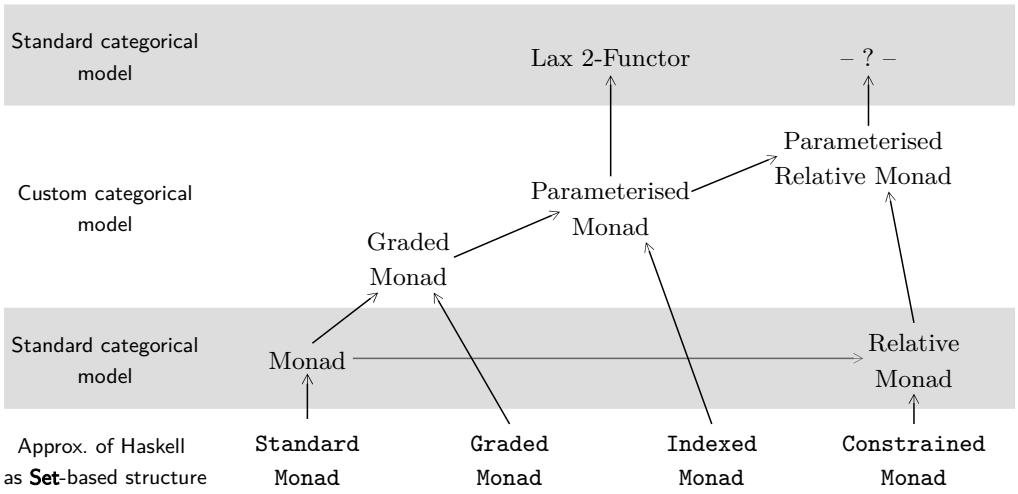


Figure 6.1: Hierarchy of categorical models for monadic notions.

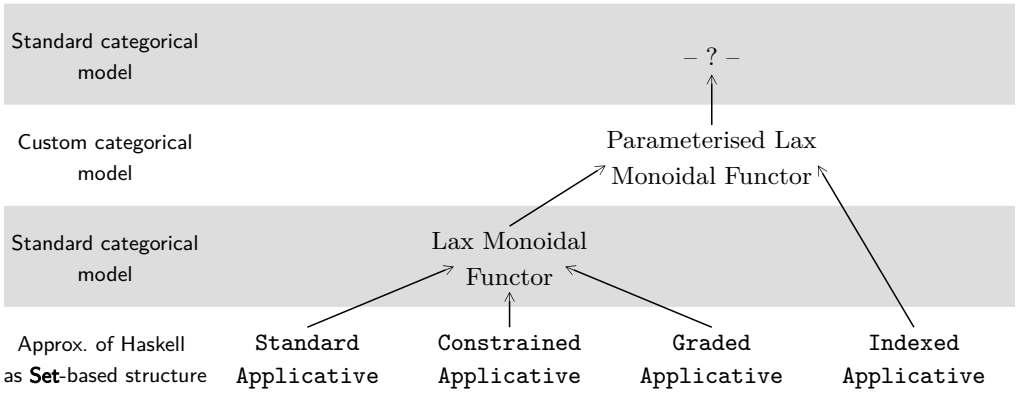


Figure 6.2: Hierarchy of categorical models for applicative notions.

First, I have given an overview of the different notions and put them into context with each other. I am not aware of any previous work that has provided a detailed categorical context in this form.

Second, I have formalised all of the involved monadic, applicative, and categorical notions in the proof assistant Agda. This formalisation provides hard evidence of the discussed relationships between the different notions.

Chapter 7

Related work

There is a substantial amount of work related to the generalisation of monads and applicatives. In this section I focus on the most closely related work.

First, Section 7.1 discuss Atkey's alternative categorical model of parameterised monads. Then Section 7.2 presents Kmett's original encoding of indexed monads in Haskell which provides the basic idea of the encoding I use for polymonads and supermonads. Afterwards, Section 7.3 briefly introduces a variety of possible generalisations of functors, applicatives, and monads that are not covered by my work. Section 7.4 discusses possible generalisation of arrows to give an outlook for possible future work in that area. A short discussion of work related to polymonads is given in Section 7.5. Afterwards, Section 7.6 shortly discusses work on unified categorical abstractions for applicatives, monads, and arrows. Finally, I close with the short discussion of an alternate approach to implement polymonads or supermonads in Section 7.7.

7.1 Atkey's parameterised notions of computation

Atkey [Atk09] also wanted to give a unified categorical model for the different kinds of parameterised monads. Though his work has a mathematical orientation, his motivation to give a unified theory for standard, graded and indexed monads is aligned with mine.

He suggests a way of modelling parameterised monads categorically through a functor

$$T : \mathbb{I}^{\mathsf{op}} \times \mathbb{I} \times \mathbb{C} \longrightarrow \mathbb{C}$$

together with natural transformations for the return and bind operations. Atkey takes an approach different from the one I present in Definition 6.26. The functor T that Atkey uses to model indexed monads can also be expressed[1] as a functor of the following shape:

$$\mathbb{I}^{\mathsf{op}} \times \mathbb{I} \longrightarrow [\mathbb{C}, \mathbb{C}]$$

[1] Agda proof: `Theory.Functor.Properties.Curry`

Hence, the associated family of functors for the indexed monad is determined by the pairs of indices. In contrast, my definition of a parameterised monad uses the morphisms of the index category to identify the associated family of functors. Therefore, Definition 6.26 is slightly more general than Atkey's definition, because there is no necessity to define an associated functor for every pair of indices and there may be more than one functor per pair of indices due the functors being associated with the morphisms. Both definitions are equivalent[2] if the index category $\mathbb{I}$ of Atkey's parameterised monad is discrete and the index category of my parameterised monad is the codiscrete counterpart of $\mathbb{I}$.

In summary, Atkey provides an alternative categorical model for parameterised monads. His model is less general than the lax 2-functors and parameterised monads that I propose in Definition 6.26. Despite being less general it appeals by being less abstract and sufficient for practical purposes in the context of functional programming. Overall my work provides a different additional way to model parameterised monads. However, Atkey's work does not consider constrained monads.

7.2 Kmett's approach to indexed monads in Haskell

In 2007 Kmett[3,4] was experimenting with the encoding of different monadic generalisations in Haskell. In particular he experimented with an encoding of indexed monads that provides the essential idea for the encoding of bind operations in the polymonad and supermonad implementation. Kmett's goal was to allow the use of indexed monads in Haskell without rigidly enforcing two indices in the type constructor. Due to the hindered type inference resulting from his encoding he gave up on it in favour of the encoding presented in Section 3.1.2 which is provided by the `indexed`[5] package.

As explained in Section 5.1.3, Kmett's work includes a functional dependency on the `Bind` type class and a specialised return operation. Both the functional dependency and the additional return operation are introduced to support type inference.

```
class Bind m n p | m n -> p where
  (>>=) :: m a -> (a -> n b) -> p b

class Return m where
  returnM :: a -> m a
```

[2] Agda proof: `Theory.Haskell.Parameterized.Indexed.Monad.Properties.IsomorphicAtkeyParameterizedMonad`

[3] Hackage: *monad-param* - `http://hackage.haskell.org/package/monad-param`

[4] *Parameterized Monads in Haskell (13. July 2007)* - `http://comonad.com/reader/2007/parameterized-monads-in-haskell/`

[5] Hackage: *indexed* - `http://hackage.haskell.org/package/indexed`

```
return :: a -> Identity a
return a = Identity a
```

To examine how well type inference performs in the context of Kmett's encoding, I applied his approach to the first case study in Section 5.4.1. The case study revealed many cases where manual type annotations and a correct choice of the return operation were still necessary to resolve ambiguous types, which resulted in tedious work.

A functional dependency is not sufficient or general enough to allow constraint solving and type inference to work properly in the case of polymonads or supermonads. Hence, the functional dependency is not part of either encoding and custom plugins are provided to resolve the insufficient constraint solving for the type classes that encode polymonads and supermonads.

However, an advantage of Kmett's approach is that it allows bind operations that can "implicitly" lift from one monad into another. For example:

```
instance Bind Maybe [] [] where
  -- (>>=) :: Maybe a -> (a -> [b]) -> [b]
  Just a >>= f = f a
  Nothing >>= _ = []
```

Note that a "lifting" from `Maybe` to list has been "embedded" into the bind operation.

Polymonads can express such "combinations" of monads, although it is not obvious which other bind operations may be required by the polymonad as a result of introducing such a "lifting" bind operation. Further investigation of polymonads may reveal a good way of providing this kind of flexibility in the future.

On the other hand, supermonads do not allow mixing different base constructors. Hence, Kmett's approach is more flexible than supermonads as there is no requirement for a single base constructor. This leads to the question of why supermonads and superapplicatives do not allow these lifting instances?

Implicit lifting may be seen as either convenient or confusing. It may even be unintentional depending on the circumstances. For example, it is not always clear when a lift should happen. If a chain of several bind operations uses the `Maybe` monad for the first computation and the list monad for the last computation, when should the lift into the list monad occur? Does the lifting happen as early as possible or as late as possible? There is no obviously correct answer to this question and arguments can be made for either strategy.

The decision when to lift may also influence the performance and the runtime behaviour of the resulting program. For example, if a bind operation from `STM` (software transactional memory) [Har+05] to `IO` is provided, the lifting strategy determines which operations take place within the same atomic `STM` computation. Depending on the circumstances, this can influence the semantics of a parallel program, which can lead to deadlocks or other undesirable behaviour.

What if the lifting decides the instance of a class that is used? In either case the lifting can influence the runtime behaviour.

There are no obviously correct answers to these questions. Hence, I decided to not allow lifting for supermonad bind operations and require the users of supermonads to express lifting from one notion to another explicitly.

However, even if there were no concerns about the semantics of implicit lifting for supermonads, it would still have to be disallowed, because the supermonad solving algorithm is based on the assumption that all `Bind` and `Applicative` instance arguments are partial applications of the same base constructor.

In addition, the categorical model also supports the decision to only allow instances with a single base constructor, because each of the different monadic and applicative notions modelled only ever contains a single base constructor.

Kmett did not present any laws or a theory for his approach, though I assume he intended a generalised version of the standard monads laws just as those that the categorical models in Section 6 and Figure 3.2 in Section 3.1 provide.

7.3 Alternate generalisations of functors, monads and applicatives

The following subsections present a variety of other generalisations of functors, applicatives, and monads aside of those presented in Section 3. The integration of these other generalisations with supermonads or polymonads remains future work. Sometimes it is unclear how and if the generalisation can be integrated. In other cases possibilities to integrate them are available for the existing encodings, but these possibilities still have outstanding issues or require further maturing.

When giving categorical explanations in the following subsections I use the category Set as an approximation for the category that Haskell resides in.

7.3.1 McBride's Kleisli arrows of outrageous fortune

In unpublished work, McBride [McB11] presents a generalisation of monads different from any of the generalisations discussed in Section 3. In his generalisation he proposes a bind and return operation with the following type signature:

$$(\texttt{>>=}) :: \mathsf{m}\ \alpha\ i \to (\forall\ j.\ \alpha\ j \to \mathsf{m}\ \beta\ j) \to \mathsf{m}\ \beta\ i$$
$$\texttt{return} :: \alpha\ i \to \mathsf{m}\ \alpha\ i$$

He exemplifies the use cases of his generalisation by using it to encode indexed monads and statically typing the opened or closed state of a file handle. Both examples can be modelled using the range of monadic notions from Section 3. Hence, there is no obvious advantage to his encoding when compared with polymonads or supermonads.

7.3.2 Functor parameterised data structures

Another generalisation is given in the packages `rank2classes`[6] and `conkin`[7]. The motivation and goal behind these packages is to provide abstractions that allow working with data structures and records that are parameterised by a functor. For example:

```
data Person f = Person
  { name :: f String
  , age :: f Int }
```

With the functor as an argument to `Person`, the data structure becomes more versatile. Depending on the functor given as argument the data structure can be used to represent people in different contexts. For example, `Identity` as argument represents a person that can be saved in or retrieved from a database, whereas `Maybe` may represent a partially initialised person or the difference between two people that only contains those values that differ.

To support data types that are structured in this way the packages provide a generalised `Functor` and `Applicative` class.

```
newtype Nat f g a = Nat (f a -> g a)

class Functor (p :: (* -> *) -> *) where
  fmap :: (forall a. Nat f g a) -> p f -> p g

class Applicative (p :: (* -> *) -> *) where
  (<*>) :: p (Nat f g) -> p f -> p g
  pure :: (forall a. f a) -> p f
```

The functor mapping and the ap operation now map natural transformations to allow mapping between the functors. The pure operation initialises values of these data structures with a default value.

Thus, the functors above represent functors that map from the category of endofunctors on Set, also referred to as $[\mathsf{Set}, \mathsf{Set}]$, into Set:

$$[\mathsf{Set}, \mathsf{Set}] \longrightarrow \mathsf{Set}$$

Notice that the kind signature reflects this categorical structure: `(* -> *) -> *`.

My encodings of polymonads and supermonads do not capture this generalisation of functors and applicatives for two reasons. First, the type constructors involved with my encodings have the kind signature `* -> *` instead of `(* -> *) -> *`. Second, my encodings are unable to replace the mapped functions with the `Nat` data type.

[6] Hackage: *rank2classes* - `http://hackage.haskell.org/package/rank2classes`
[7] Hackage: *conkin* - `http://hackage.haskell.org/package/conkin`

The first shortcoming can be remedied by using a polymorphic kind variable instead of star, i.e., `k -> *`, in the kind of the type constructors. The second shortcoming can be addressed by adding an associated type family that allows specifying which type the mapping arrows have.

```
class Functor (f :: k -> *) where
  type ArrF f (a :: k) (b :: k) :: *
  fmap :: (ArrF f a b) -> f a -> f b

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

Unfortunately, the additional type synonym cannot be shared between the functor and applicative class, because they are required to have different kinds.

7.3.3 Different source categories for functors

The functors encoded in Haskell usually are endofunctors on Set. However, as demonstrated by the work in the previous section the source category can be exchanged with other categories.

Another example where the source category is exchanged are invertible syntax descriptions [RO10]. To ensure that mappings can be inverted these syntax descriptions require the mapped arrows to be partial isomorphisms instead of simple functions.

```
newtype Iso a b = Iso (a -> Maybe b) (b -> Maybe a)

class IsoFunctor f where
  fmap :: Iso a b -> f a -> f b
```

Here the source category Set is replaced with the category of types and partial isomorphism between them.

This generalisation can be adapted in a similar manner as described in the previous section by introducing an associated type synonym.

7.3.4 Higher-order functors

The work of Johann and Ghani [JG07] demonstrates how functors can be generalised to higher-order functors. Their work on GADTs [JG08] uses higher-order functors as a tool to work with GADTs and model their semantics. Further, Swierstra [Swi08] notices that some of the data types involved with his presentation of free monads in “Data types à la carte” actually form higher-order functors as well.

Categorically higher-order functors have the following form:

$$[\mathsf{Set}, \mathsf{Set}] \longrightarrow [\mathsf{Set}, \mathsf{Set}]$$

Hence, higher-order functors map functors to functors and natural transformations to natural transformations. Johann and Ghani [JG07] encode higher-order functors as follows:

```
newtype Nat f g = Nat (forall a. f a -> g a)

class HFunctor (f :: (* -> *) -> (* -> *))  where
  ffmap :: Functor g => (a -> b) -> f g a -> f g b
  hfmap :: Nat g h -> Nat (f g) (f h)
```

The `ffmap` function provides the functor-to-functor mapping and the `hfmap` provides the natural transformation mapping.

There is no obvious way to integrate higher-order functors with the existing encoding of functors in the polymonad or supermonad implementations.

7.3.5 Another approach to constrained functors and monads

There have been previous discussions[8] and encodings of constrained functors and monads to allow types such as `Set` to form a functor and monad.

The `rmonad` (restricted monads) package[9] provides such an encoding of constrained functors and monads. The restricted monads apply the same constraints to every result type involved with the type of a monadic or functor operation. For example, in the `rmonad` package `fmap` has the type signature

```
fmap :: (FunctorCts f a, FunctorCts f b)
     => (a -> b) -> f a -> f b
```

whereas for supermonads it has the following type signature:

```
fmap :: (FunctorCts f a b)
     => (a -> b) -> f a -> f b
```

The `rmonad` approach has the advantage that polymorphic functions involving constrained notions are less cluttered with constraints. However, the supermonad approach is more flexible as constraints can distinguish between the two result types `a` and `b`.

Notice that the approach taken by the `rmonad` package is fully compatible with the categorical background I discuss to model constrained functors, applicatives, and monads in Section 6.1.9, 6.2.5 and 6.3.3. The `rmonad` approach restricts constraints to only occur on the objects of the constrained source categories.

[8] *Haskell Programming: Attractive Types* - `http://okmij.org/ftp/Haskell/types.html#restricted-datatypes`

[9] Hackage: *rmonad* - `http://hackage.haskell.org/package/rmonad`

7.4 Generalisations of arrows

As has been hinted in the introduction, arrows are another popular notion of computation. Multiple generalisations have been discussed for arrows, but in the context of arrows there are also more complications and choices that arise.

7.4.1 Adding indices or constraints

One possible way to generalise arrows is to introduce indices similar to those of graded or indexed monads. Nilsson and Nielsen [NN14] require this kind of generalisation to properly type their arrow framework for Bayesian inference. However, I have not come across other use cases for this generalisation yet. Hence, there is no clear indication as to how these indices should behave across the different arrow operations in general. Even the above paper only provides the indexing structure on a subset of the arrow operations.

Another possible generalisation of arrows could introduce constraints similar to constrained monads and applicatives. However, I have not come across an implementation or use case of this generalisation yet.

7.4.2 Abstracting the involved notion of product and function

The arrow operations are fundamentally connected to products and functions:

```
class Category => Arrow a where
  arr :: (b -> c) -> a b c

  first  :: a b c -> a (b, d) (c, d)
  second :: a b c -> a (d, b) (d, c)

  (***) :: a b c -> a b' c' -> a (b, b') (c, c')
  (&&&) :: a b c -> a b  c' -> a b       (c, c')
```

The `arr` operation allows embedding arbitrary Haskell functions into an arrow and all of the other operations produce arrows that involve products. In many possible application domains this inherent use of functions and products poses a problem.

For example, in bidirectional programming [Ali+05; JHH09] embedding a function is problematic, because such a function does not provide or necessarily have an inverse.

Another example is describing synchronous circuits [Pat01] with arrows. Again allowing to embed an arbitrary function into an arrow is to powerful, because an arbitrary function cannot necessarily be translated into a finite circuit especially if general recursion is involved. In addition, depending on the representation of the circuits, e.g., deep or shallow embedding, it may also be advantageous to use a different type to replace the built-in use of products.

A first attempt to remedy these shortcomings may be to get rid of the `arr` function, but that is not possible, because it is essential to the translation of the `proc`-notation [Pat01], i.e., the syntactic support for arrows in Haskell.

Joseph [Jos14] explores a generalisation of arrows that allows to exchange the product type and does not require the `arr` function. The generalised arrow type class is defined as follows:

```
class Category g => GArrow g (**) u where
  ga_first  :: g x y -> g (x ** z) (y ** z)
  ga_second :: g x y -> g (z ** x) (z ** y)

  ga_cancell :: g (u ** x) x
  ga_cancelr :: g (x ** u) x

  ga_uncancell :: g x (u ** x)
  ga_uncancelr :: g x (x ** u)

  ga_assoc   :: g ((x ** y) ** z) (x ** (y ** z))
  ga_unassoc :: g (x ** (y ** z)) ((x ** y) ** z)
```

He also developed a branch of GHC that supports `proc`-Notation for his generalised arrows. Unfortunately, his branch no longer works with modern versions of GHC.

7.4.3 Problems when generalising arrows

The major problem when generalising arrows in Haskell is that the `proc`-notation is not as flexible as the `do`-notation. As mentioned in the previous section the `proc`-notation inherently relies on the `arr` function. In addition, it is insufficiently documented which types are permitted for the functions in the `Arrow` class when enabling `RebindableSyntax`: The documentation states that "the details are in flux" and that the replacing operations have to "match the prelude type closely" (GHC User's Guide[10], Section 9.3.15).

7.5 Polymonads

Swamy et al. [Swa+11] presented a way to automatically insert bind and return operations into functional programs. The aim of their work was to provide a way of writing implicitly monadic programs and to also cover the integration of morphisms between different monads in those programs. Their work provides insight in how monadic generalisations may be integrated into other languages in

[10] *Glasgow Haskell Compiler User's Guide (8.2.1)* - `http://downloads.haskell.org/~ghc/8.2.1/docs/html/users_guide`

the future. However, their work does not solve the problems described during the discussion of implicit lifting in the context of Kmett's approach in Section 7.2.

Building on the work of Swamy et al. [Swa+11], polymonads are introduced in the work by Guts et al. [Gut+12] and Hicks et al. [Hic+14]. Their work focuses on the formal introduction of polymonads, the coherence of type inference for polymonads and a prototypical implementation of polymonads in a small toy language. Polymonads provide a unified notion that captures standard, graded and indexed monads. Unfortunately, their implementation requires the involved polymonads to be principal in order for type inference to work properly. As a result, the parameterised monads require their indices to be phantom. It is also unclear how and if constrained monads are supported by the polymonad theory.

Section 4 introduces polymonads and discusses their integration into Haskell. A full discussion of the advantages and disadvantages of the polymonad approach and how it compares to supermonads is given in the overall conclusions (Section 8.1).

7.6 Unifying applicatives, monads and arrows

My work is concerned with the unification of monadic and applicative generalisations, respectively. It is not concerned with the connections *between* applicatives, monads, and arrows.

Work by Rivas and Jaskelioff [RJ17] shows that applicatives, monads, and arrows can all be seen as monoids in monoidal categories. They demonstrate that useful results can be obtained even when working at a high level of abstraction. It remains future work to explore how and if this abstract connection between the three notions generalises to the generalisations of applicatives, monads, and arrows.

7.7 Alternatives to GHC type checker plugins

My integration of polymonads and supermonads in Haskell uses the GHC type checker plugins[11] to restore type inference. However, there are other possible approaches to implement the special support required by polymonads and supermonads.

A possible alternate approach are constraint handling rules. Jones [Jon95] presented custom improvements which provide a system to aid constraint solving by associating patterns of constraints containing open type variables with equations involving those type variables. Stuckey and Sulzmann [SS05] developed a theory of constraint handling rules that applies custom improvements to functional languages. They also developed a prototype language with constraint

[11] *Glasgow Haskell Compiler User's Guide (8.2.1), Section 11.3.4* - `https://downloads.haskell.org/~ghc/8.2.1/docs/html/users_guide/extending_ghc.html#typechecker-plugins`

handling rules called Chameleon [SSW04]. Unfortunately, their implementation is not available publicly anymore. Although there is no implementation of constraint handling rules for GHC work on them has not ceased and other authors such as Serrano and Hage [SH17] are still working on implementations that may one day be integrated with GHC or other Haskell compilers. Therefore, to my knowledge, GHC plugins are the most practical way of implementing and integrating polymonads and supermonads into GHC.

However, it is important to note that the idea of the polymonad or supermonad approach is not deeply connected to GHC's type checker plugins or any particular implementation. Polymonads and supermonads can be implemented in other languages or Haskell implementations in different ways or may even become a core part of another language. The current implementation is a just first step to allow the use and experimentation with polymonads and supermonads in Haskell.

Chapter 8

Conclusions

I have presented a variety of different generalised monadic notions that are already in use in many Haskell programs today. Based on these notions I developed corresponding generalised notions of applicatives that, to my knowledge, have not been explored before.

My work on polymonads and supermonads enables programmers to use and experiment with the generalised notions in a uniform manner, fostering code reuse through a common standard library and removing the need for tedious annotations when working with a variety of notions at the same time.

I have also explored and found categorical models that subsume all of the different monadic and applicative generalisations I discuss.

8.1 Polymonads vs supermonads

Building on the previous work from Kmett[1], Guts et al. [Gut+12] and Hicks et al. [Hic+14] I developed a Haskell language extension for polymonads. The polymonad approach has the downside that it only supports standard, graded, and indexed monads with phantom indices. Hence, I proceeded to develop supermonads and superapplicative as a language extension for Haskell. Supermonads and superapplicatives support all of the monadic and applicative notions I presented in Section 3. Both extensions of the language use GHC's type checker plugin mechanism to achieve their goals.

Polymonads [Hic+14] are similar to supermonads in that their encoding of bind operations in Haskell are alike. Though supermonads and polymonads appear similar, there are several key differences:

- Polymonads do not have specific return operations. They encode their return operations through a special bind operation.
- There is not necessarily a common base constructor for a given polymonad.

[1] *Parameterized Monads in Haskell (13. July 2007)* - `http://comonad.com/reader/2007/parameterized-monads-in-haskell/`

- All polymonads also have to contain a distinguished identity type constructor.
- A polymonad can be the union of several different polymonads and it is not immediately clear which bind operation belongs to which original polymonad.

Whether one of the notions subsumes the other, and what the exact relationship between supermonads and polymonads is, remains future work. Though both notions are probably incomparable, because there are generalisations that polymonads can express and supermonads cannot and the same is also true the other way around.

The laws of polymonads are more complex than the laws of the categorical models for supermonads and superapplicatives and they do not as obviously relate to the standard monad laws either. Though the standard monad laws can be derived[2] from the polymonad laws.

To guarantee the existence of a unique solution to a set of polymonad constraints a polymonad has to be principal. This property essentially ensures that there always exists a best solution for any given ambiguous type constructor. Due to the requirement that polymonads need to be principal for solving they only support *phantom* indices for graded and indexed monads. This is a major disadvantage compared to supermonads, because non-phantom indices allow for many interesting examples and applications.

The polymonad theory also does not offer obvious support for constraints on result types and thus does not support constrained monads. The feasibility of integrating such constraints into the polymonad theory is an open question. Although supermonads support constraints on result types, their support comes with practical implications. Due to these implications, I offer a separate prelude that allows programmers to choose whether they want to deal with these implications or not when working with supermonads.

One advantage of polymonads over supermonads is that they allow more than one base constructor to be used. This opens a design space for monadic notions different from the ones discussed, including the implicit lifting bind operations mentioned in the comparison with Kmett's approach (Section 7.2). Hence, polymonads are more general then supermonads. The definition of polymonads allows for a wider range of monadic notions then the supermonad approach does. What limits the practical applicability of polymonads is mainly the requirement of principality.

8.2 Categorical models

The exploration of categorical models for the different generalised notions shows that there are abstract structures that connect many of the presented notions.

[2] Agda proof: `Haskell.Monad.Polymonad`

Namely, lax 2-functors provide a common categorical model for all of the parameterised monadic notions and lax monoidal functors model all of the applicative notions except indexed applicatives. In addition, I proposed the custom definitions of *parameterised relative monads* (Definition 6.29) and *parameterised lax monoidal functors* (Definition 6.34) to capture all of the monadic and applicative notions, respectively.

Although I did not find a pre-existing categorical notion that subsumes the proposed definitions, my work has made great progress towards a categorical model that connects all of the different notions discussed.

The terms supermonad and superapplicative refer to the general approach and technique to represent a unified encoding and implementation of the different monadic and applicative notions in a language. Hence, the supermonad approach though implemented in Haskell may be applicable to other languages in the future. The supported generalisations should be characterised by the categorical notions presented in Section 6. Although the proposed categorical models characterise the supported notions, the plugin cannot enforce their laws or their proper implementation, just as GHC cannot ensure that instances of the standard `Monad` class are correct.

8.3 Future work

There are numerous lines of future work.

For polymonads, the support of non-phantom indices and constrained monads in practice is an important topic for future work. Extending polymonads to support applicatives also provides an interesting line of work.

The supermonad approach would benefit from a better handling of constraints on result types when writing polymorphic code. Future work on supermonads may also further explore the laws given in [BN16] and extends them to superapplicatives as discussed in Section 5.1.4.

As discussed in Section 7.4, exploring the different generalisations of arrows, finding a unified notion for them, and extending the Haskell language support for generalisations of arrows provides a rich pool of future work.

Aside of applicatives, monads, and arrows there are other notions of computation, e.g., comonads, that should also be considered for generalisation.

The Haskell standard library[3] has a rich hierarchy of type classes for monads and applicatives, e.g., `MonadPlus`, `Alternative`, `MonadFix`, and `Traversable`. In the context of polymonads and supermonads all of these useful type classes require adaptation and integration with the monadic and applicative generalisations.

I present categorical models for all of the notions in Section 3 and I propose to use these models to specify which instantiations of supermonads are valid. However, a proper operational or denotational semantics of the polymonad and

[3]Hackage: *base* - `http://hackage.haskell.org/package/base`

supermonad plugin may be useful especially to analyse how the language extensions interact with Haskell's type system and constraint solving mechanism [Vyt+11].

Bibliography

[ACU10] Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. "Monads Need Not Be Endofunctors". In: *Foundations of Software Science and Computational Structures*. Ed. by Luke Ong. Vol. 6014. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 297–311. ISBN: 978-3-642-12031-2. DOI: 10.1007/978-3-642-12032-9_21.

[Ada93] Stephen Adams. "Functional Pearls Efficient sets — a balancing act". In: *Journal of Functional Programming* 3 (04 1993), pp. 553–561. ISSN: 1469-7653. DOI: 10.1017/S0956796800000885.

[Ali+05] Artem Alimarine, Sjaak Smetsers, Arjen van Weelden, Marko van Eekelen, and Rinus Plasmeijer. "There and Back Again: Arrows for Invertible Programming". In: *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*. Haskell '05. Tallinn, Estonia: ACM, 2005, pp. 86–97. ISBN: 1-59593-071-X. DOI: 10.1145/1088348.1088357.

[Atk09] Robert Atkey. "Parameterised notions of computation". In: *Journal of Functional Programming* 19 (3-4 2009), pp. 335–376. ISSN: 1469-7653. DOI: 10.1017/S095679680900728X.

[Awo06] Steve Awodey. *Category theory*. Oxford University Press, 2006. ISBN: 9780199237180.

[Bén63] Jean Bénabou. "Categories avec multiplication". French. In: *C. R. Acad. Sci., Paris* 256 (1963), pp. 1887–1890. ISSN: 0001-4036. URL: http://gallica.bnf.fr/ark:/12148/bpt6k3208j/f1965.item.r=Benabou.

[Bén65] Jean Bénabou. "Categories rélatives". French. In: *C. R. Acad. Sci., Paris* 260 (1965), pp. 3824–3827. ISSN: 0001-4036. URL: http://gallica.bnf.fr/ark:/12148/bpt6k4019v/f37.item.r=benabou.

[Bén67] Jean Bénabou. "Introduction to bicategories". In: *Reports of the Midwest Category Seminar*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1967, pp. 1–77. ISBN: 978-3-540-35545-8. DOI: 10.1007/BFb0074299.

[BG14] Jan Bracker and Andy Gill. "Practical Aspects of Declarative Languages: 16th International Symposium, PADL 2014, San Diego, CA, USA, January 20-21, 2014. Proceedings". In: ed. by Matthew Flatt and Hai-Feng Guo. Cham: Springer International Publishing, 2014. Chap. Sunroof: A Monadic DSL for Generating JavaScript, pp. 65–80. ISBN: 978-3-319-04132-2. DOI: 10.1007/978-3-319-04132-2_5.

[BN15] Jan Bracker and Henrik Nilsson. "Polymonad Programming in Haskell". In: *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages*. IFL '15. Koblenz, Germany: ACM, 2015, 3:1–3:12. ISBN: 978-1-4503-4273-5. DOI: 10.1145/2897336.2897340.

[BN16] Jan Bracker and Henrik Nilsson. "Supermonads: One Notion to Bind Them All". In: *Proceedings of the 9th International Symposium on Haskell*. Haskell 2016. Nara, Japan: ACM, 2016, pp. 158–169. ISBN: 978-1-4503-4434-0. DOI: 10.1145/2976002.2976012.

[BN18] Jan Bracker and Henrik Nilsson. "Supermonads and superapplicatives". Submitted to the Journal of Functional Programming. 2018.

[Bot+17] Gert-Jan Bottu, Georgios Karachalias, Tom Schrijvers, Bruno C. d. S. Oliveira, and Philip Wadler. "Quantified Class Constraints". In: *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*. Haskell 2017. Oxford, UK: ACM, 2017, pp. 148–161. ISBN: 978-1-4503-5182-9. DOI: 10.1145/3122955.3122967.

[CKJ05] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. "Associated Type Synonyms". In: *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*. ICFP '05. Tallinn, Estonia: ACM, 2005, pp. 241–253. ISBN: 1-59593-064-7. DOI: 10.1145/1086365.1086397.

[Dia15] Iavor S. Diatchki. "Improving Haskell Types with SMT". In: *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell*. Haskell 2015. Vancouver, BC, Canada: ACM, 2015, pp. 1–10. ISBN: 978-1-4503-3808-0. DOI: 10.1145/2804302.2804307.

[Dijnd] Edsger W. Dijkstra. "Een algorithme ter voorkoming van de dodelijke omarming". circulated privately. n.d. URL: http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD108.PDF.

[DP11] Dominique Devriese and Frank Piessens. "Information Flow Enforcement in Monadic Libraries". In: *Proceedings of the 7th ACM SIGPLAN Workshop on Types in Language Design and Implementation*. TLDI '11. Austin, Texas, USA: ACM, 2011, pp. 59–72. ISBN: 978-1-4503-0484-9. DOI: 10.1145/1929553.1929564.

[Gab+16] Marco Gaboardi, Shin-ya Katsumata, Dominic Orchard, Flavien Breuvart, and Tarmo Uustalu. "Combining Effects and Coeffects via Grading". In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ICFP 2016. Nara, Japan: ACM, 2016, pp. 476–489. ISBN: 978-1-4503-4219-3. DOI: 10.1145/2951913.2951939.

[Gio+11] George Giorgidze, Torsten Grust, Tom Schreiber, and Jeroen Weijers. "Haskell Boards the Ferry". In: *Implementation and Application of Functional Languages: 22nd International Symposium, IFL 2010, Alphen aan den Rijn, The Netherlands, September 1-3, 2010, Revised Selected Papers*. Ed. by Jurriaan Hage and Marco T. Morazán. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1–18. ISBN: 978-3-642-24276-2. DOI: 10.1007/978-3-642-24276-2_1.

[Gun15] Adam Gundry. "A Typechecker Plugin for Units of Measure: Domain-specific Constraint Solving in GHC Haskell". In: *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell*. Haskell 2015. Vancouver, BC, Canada: ACM, 2015, pp. 11–22. ISBN: 978-1-4503-3808-0. DOI: 10.1145/2804302.2804305.

[Gut+12] Nataliya Guts, Michael Hicks, Nikhil Swamy, Daan Leijen, and Gavin Bierman. *Polymonads*. Tech. rep. Extended version of POPL'13 submission. University of Maryland Department of Computer Science, 2012. URL: http://www.cs.umd.edu/~mwh/papers/polymonadsTR.pdf.

[Har+05] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. "Composable Memory Transactions". In: *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '05. Chicago, IL, USA: ACM, 2005, pp. 48–60. ISBN: 1-59593-080-9. DOI: 10.1145/1065944.1065952.

[Hic+14] Michael Hicks, Gavin Bierman, Nataliya Guts, Daan Leijen, and Nikhil Swamy. "Polymonadic Programming". In: *Electronic Proceedings in Theoretical Computer Science* 153 (2014), pp. 79–99. DOI: 10.4204/EPTCS.153.7.

[Hoa69] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Communications of the ACM* 12.10 (1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259.

[Hug00] John Hughes. "Generalising monads to arrows". In: *Science of Computer Programming* 37.1–3 (2000), pp. 67–111. ISSN: 0167-6423. DOI: 10.1016/S0167-6423(99)00023-4.

[Hug99] John Hughes. "Restricted data types in Haskell". In: *Haskell Workshop*. Vol. 99. 1999.

[JG07] Patricia Johann and Neil Ghani. "Initial Algebra Semantics Is Enough!" In: *Typed Lambda Calculi and Applications*. Ed. by Simona Ronchi Della Rocca. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 207–222. ISBN: 978-3-540-73228-0.

[JG08] Patricia Johann and Neil Ghani. "Foundations for Structured Programming with GADTs". In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '08. San Francisco, California, USA: ACM, 2008, pp. 297–308. ISBN: 978-1-59593-689-9. DOI: 10.1145/1328438.1328475.

[JHH09] Bart Jacobs, Chris Heunen, and Ichiro Hasuo. "Categorical semantics for arrows". In: *Journal of Functional Programming* 19.3-4 (July 2009), pp. 403–438. DOI: 10.1017/S0956796809007308.

[Jon95] Mark P. Jones. "Simplifying and Improving Qualified Types". In: *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*. FPCA '95. La Jolla, California, USA: ACM, 1995, pp. 160–169. ISBN: 0-89791-719-7. DOI: 10.1145/224164.224198.

[Jos14] Adam Megacz Joseph. "Generalized Arrows". PhD thesis. EECS Department, University of California, Berkeley, 2014. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-130.html.

[Kat14] Shin-ya Katsumata. "Parametric Effect Monads and Semantics of Effect Systems". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. San Diego, California, USA: ACM, 2014, pp. 633–645. ISBN: 978-1-4503-2544-8. DOI: 10.1145/2535838.2535846.

[Mac69] Saunders Mac Lane. *Categories for the working mathematician*. Vol. 5. Springer, 1969. ISBN: 978-1-4757-4721-8.

[Mar+14] Simon Marlow, Louis Brandy, Jonathan Coens, and Jon Purdy. "There is No Fork: An Abstraction for Efficient, Concurrent, and Concise Data Access". In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP '14. Gothenburg, Sweden: ACM, 2014, pp. 325–337. ISBN: 978-1-4503-2873-9. DOI: 10.1145/2628136.2628144.

[Mar+16] Simon Marlow, Simon Peyton Jones, Edward Kmett, and Andrey Mokhov. "Desugaring Haskell's Do-notation into Applicative Operations". In: *Proceedings of the 9th International Symposium on Haskell*. Haskell 2016. Nara, Japan: ACM, 2016, pp. 92–104. ISBN: 978-1-4503-4434-0. DOI: 10.1145/2976002.2976007.

[McB11] Conor McBride. "Functional pearl: Kleisli arrows of outrageous fortune". Unpublished. 2011. URL: https://personal.cis.strath.ac.uk/conor.mcbride/Kleisli.pdf.

[Mog88] Eugenio Moggi. *Computational Lambda-Calculus and Monads*. IEEE Computer Society Press, 1988, pp. 14–23. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.80.3104&rep=rep1&type=pdf.

[Mog91] Eugenio Moggi. "Notions of computation and monads". In: *Information and Computation* 93.1 (1991). Selections from 1989 IEEE Symposium on Logic in Computer Science, pp. 55–92. ISSN: 0890-5401. DOI: 10.1016/0890-5401(91)90052-4.

[MP08] Conor McBride and Ross Paterson. "Applicative programming with effects". In: *Journal of Functional Programming* 18 (01 2008), pp. 1–13. ISSN: 1469-7653. DOI: 10.1017/S0956796807006326.

[NN14] Henrik Nilsson and Thomas A. Nielsen. "Declarative Modelling for Bayesian Inference by Shallow Embedding". In: *Proceedings of the 6th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. EOOLT '14. Berlin, Germany: ACM, 2014, pp. 39–42. ISBN: 978-1-4503-2953-8. DOI: 10.1145/2666202.2666208.

[Nor07] Ulf Norell. "Towards a practical programming language based on dependent type theory". PhD thesis. Chalmers University of Technology, 2007. ISBN: 978-91-7291-996-9. URL: http://www.cse.chalmers.se/~ulfn/papers/thesis.pdf.

[OM12] Dominic Orchard and Alan Mycroft. "Categorical Programming for Data Types with Restricted Parametricity". Draft submitted and rejected by TFP 2012 post-proceedings. 2012. URL: https://www.cl.cam.ac.uk/~dao29/drafts/tfp-structures-orchard12.pdf.

[OP14] Dominic Orchard and Tomas Petricek. "Embedding Effect Systems in Haskell". In: *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*. Haskell '14. Gothenburg, Sweden: ACM, 2014, pp. 13–24. ISBN: 978-1-4503-3041-1. DOI: 10.1145/2633357.2633368.

[PAS12] Anders Persson, Emil Axelsson, and Josef Svenningsson. "Generic Monadic Constructs for Embedded Languages". In: *Implementation and Application of Functional Languages: 23rd International Symposium, IFL 2011, Lawrence, KS, USA, October 3-5, 2011, Revised Selected Papers*. Ed. by Andy Gill and Jurriaan Hage. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 85–99. ISBN: 978-3-642-34407-7. DOI: 10.1007/978-3-642-34407-7_6.

[Pat01] Ross Paterson. "A New Notation for Arrows". In: *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*. ICFP '01. Florence, Italy: ACM, 2001, pp. 229–240. ISBN: 1-58113-415-0. DOI: 10.1145/507635.507664.

[Pie91] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. Cambridge, MA, USA: MIT Press, 1991. ISBN: 0-262-66071-7.

[PT08] Riccardo Pucella and Jesse A. Tov. "Haskell Session Types with (Almost) No Class". In: *Proceedings of the First ACM SIGPLAN Symposium on Haskell*. Haskell '08. Victoria, BC, Canada: ACM, 2008, pp. 25–36. ISBN: 978-1-60558-064-7. DOI: 10.1145/1411286.1411290.

[RJ17] Exequiel Rivas and Mauro Jaskelioff. "Notions of computation as monoids". In: *Journal of Functional Programming* 27 (Oct. 2017). DOI: 10.1017/S0956796817000132.

[RO10] Tillmann Rendel and Klaus Ostermann. "Invertible Syntax Descriptions: Unifying Parsing and Pretty Printing". In: *Proceedings of the Third ACM Haskell Symposium on Haskell*. Haskell '10. Baltimore, Maryland, USA: ACM, 2010, pp. 1–12. ISBN: 978-1-4503-0252-4. DOI: 10.1145/1863523.1863525.

[Sch+08] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. "Type Checking with Open Type Functions". In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. ICFP '08. Victoria, BC, Canada: ACM, 2008, pp. 51–62. ISBN: 978-1-59593-919-7. DOI: 10.1145/1411204.1411215.

[Scu+13] Neil Sculthorpe, Jan Bracker, George Giorgidze, and Andy Gill. "The Constrained-monad Problem". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. Boston, Massachusetts, USA: ACM, 2013, pp. 287–298. ISBN: 978-1-4503-2326-0. DOI: 10.1145/2500365.2500602.

[SD96] S. Doaitse Swierstra and Luc Duponcheel. "Deterministic, error-correcting combinator parsers". In: *Advanced Functional Programming: Second International School Olympia, WA, USA, August 26–30, 1996 Tutorial Text*. Ed. by John Launchbury, Erik Meijer, and Tim Sheard. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 184–207. ISBN: 978-3-540-70639-7. DOI: 10.1007/3-540-61628-4_7.

[SH17] Alejandro Serrano and Jurriaan Hage. "Constraint handling rules with binders, patterns and generic quantification". In: *Theory and Practice of Logic Programming* 17.5-6 (2017), pp. 992–1009. DOI: 10.1017/S1471068417000230.

[SS05] Peter J. Stuckey and Martin Sulzmann. "A Theory of Overloading". In: *ACM Transactions on Programming Languages and Systems* 27.6 (2005), pp. 1216–1269. ISSN: 0164-0925. DOI: 10.1145/1108970.1108974.

[SSW04] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. "The Chameleon System". In: *First Workshop on Constraint Handling Rules: Selected papers*. Ed. by Thom Frühwirth and Marc Meister. University of Ulm. 2004, pp. 13–32. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.5.7319.

[Swa+11] Nikhil Swamy, Nataliya Guts, Daan Leijen, and Michael Hicks. *Lightweight Monadic Programming in ML*. Tech. rep. MSR-TR-2011-39. Microsoft Research, 2011. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=147003.

[Swi08] Wouter Swierstra. "Data types à la carte". In: *Journal of Functional Programming* 18.4 (2008), pp. 423–436. DOI: 10.1017/S0956796808006758.

[VAS06] Juliana Vizzotto, Thorsten Altenkirch, and Amr Sabry. "Structuring quantum effects: superoperators as arrows". In: *Mathematical Structures in Computer Science* 16 (03 June 2006), pp. 453–468. ISSN: 1469-8072. DOI: 10.1017/S0960129506005287.

[Vyt+11] Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. "OutsideIn(X) Modular type inference with local assumptions". In: *Journal of Functional Programming* 21 (Special Issue 4-5 Sept. 2011), pp. 333–412. ISSN: 1469-7653. DOI: 10.1017/S0956796811000098.

[Wad92] Philip Wadler. "The Essence of Functional Programming". In: *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '92. Albuquerque, New Mexico, USA: ACM, 1992, pp. 1–14. ISBN: 0-89791-453-8. DOI: 10.1145/143165.143169.

[Wad94] Philip Wadler. "Monads and composable continuations". In: *LISP and Symbolic Computation* 7.1 (1994), pp. 39–55. ISSN: 0892-4635. DOI: 10.1007/BF01019944.

[Wad98] Philip Wadler. "The Marriage of Effects and Monads". In: *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*. ICFP '98. Baltimore, Maryland, USA: ACM, 1998, pp. 63–74. ISBN: 1-58113-024-4. DOI: 10.1145/289423.289429.

Appendix A

How to read Agda

The following sections provide a short introduction to Agda. The introduction assumes that the reader is familiar with Haskell. The aim of the introduction is to allow the reader to understand the Agda proofs referenced in this thesis. Hence, this introduction explains how to read Agda code and not how to write it. The introduction only focuses on features of Agda that are used by the referenced proofs and therefore does not cover all of the language.

A.1 General remarks

Agda is dependently typed and therefore can use variables introduced as part of a type signature within that type signature. Hence, the order of declarations is relevant and important.

```
concatVec : {ℓ : Level} {A : Set ℓ}
          → (n m : ℕ) → Vec A n → Vec A m → Vec A (n + m)
```

In the above example the value of the argument ℓ is used to specify the type of `a` and that type `a` together with the natural numbers `n` and `m` are used as arguments for the vector type `Vec`.

Notice that Agda uses colon (`:`) instead of a double colon (`::`) to separate identifiers from their type. In addition, Agda embraces the use of unicode and allows it in identifiers.

Furthermore, notice the different brackets used for the introduction of ℓ and `A`. The curly brackets (`{}`) introduce implicit arguments whereas the standard brackets (`()`) introduce normal arguments. When instantiating arguments of a function or module an implicit argument is optional. The implicit arguments can be given explicitly, but this is unnecessary if their values can be deduced from the subsequent arguments. Hence, `concatVec` could be partially applied in different ways:

```
concatVec n m v₁ v₂
concatVec {ℓ} {A} n m v₁ v₂
concatVec {A = SomeType} n m v₁ v₂
```

The first application does not specify the implicit arguments and trusts that Agda is capable of deducing their value automatically from the other arguments. The second application explicitly specifies both implicit arguments. The third application only specifies the arguments `A` explicitly. The value of `A` has to be given in named notation, because the first implicit argument ℓ has not been given explicitly. When declaring a function the curly braces are also necessary to pattern match on implicit arguments.

Every type in Agda forms a set (as in set theory). Therefore, the type of a type is the set of all types: `Set` ℓ. This relationship can be seen in the second implicit argument: `A : Set` ℓ. To avoid the inconsistencies that would result if `Set` ℓ were a member of itself, Agda provides a hierarchy of sets:

```
Set zero : Set (suc zero) : ... : Set ℓ : Set (suc ℓ) : ...
```

Simple types, such as `Bool`, are in `Set zero`, whereas more complex types, e.g., types that contain other types, reside in higher-level sets. The indices of the set hierarchy, are encoded through `zero` and `suc` constructors (peano numbers) and are provided by the built-in type `Level`.

Agda has no strict rules about the capitalisation of names for types or constructors. Since types and values can be used at both type and value level a forced syntactic separation is not necessary and the user is not forced to use uppercase and lowercase in a specific way.

A.2 Modules

An Agda module is similar in appearance to a Haskell module. The module name corresponds to the folder and file name just as in Haskell. Within the file the module is declared as follows:

```
module My.Module where
```

In contrast to Haskell an Agda module can have arguments which are listed behind the module's name.

```
module My.Module {ℓ : Level} (T : Set ℓ) where
```

In this example there are two arguments. The first argument ℓ is of type `Level` and the second arguments `T` is of type `Set` ℓ. Module arguments introduce variables that are in scope for all declarations within the module. The arguments can be instantiated when importing the module elsewhere or if not instantiated during import supplied as additional arguments to all declarations imported from the module.

A module can also contain submodules which are declared in the same way as a top-level module.

A.3 Imports

Imports of other modules can be written anywhere within a module, before and after the module declaration, but also locally within a `where`-clause. The imported declarations are then visible in subsequent declarations within the same scope as the import. An import of all declarations in a module can be given as follows:

```
open import Data.Maybe
```

Which declarations are imported can be controlled with the `using` and `hiding` modifiers, respectively. Naming conflicts can be resolved with the `renaming` modifier.

```
open import Data.Maybe
    using ( Maybe ; nothing )
    renaming ( just to Just )
open import Data.Bool hiding ( true )
```

In the first import above, `using` signifies that only `Maybe`, `nothing`, and `Just` from the module `Data.Maybe` are brought into scope, where `Just` is an alias for the hidden `just`. In the second import, `hiding` indicates that `true` is the only declaration from `Data.Bool` that is not brought into scope. The `hiding` modifier can be combined with `renaming` just as the `using` modifier. Lists of identifiers and renamings are separated by a semicolon (`;`).

To import the contents of a submodules, the parent module has to be imported as described above. The submodule can then be opened as follows:

```
open SubModule
```

Note that the `import` keyword is not necessary and full qualification of the submodule is not necessary either. The `using`, `hiding` and `renaming` modifiers can be used with submodule imports as well.

As mentioned in Section A.1 arguments of a module can be supplied during import. For example, the arguments for the module from Section A.2

```
module My.Module {ℓ : Level} (T : Set ℓ) where
```

can be given as follows:

```
open import My.Module Maybe
```

Note that, the implicit argument was not given in this case. The application of module arguments is analogues to the application of a function to arguments and can also be partial.

A.4 Functions and operators

Just as in Haskell pattern matching is used to declare functions.

```
concatVec : {ℓ : Level} {a : Set ℓ}
          → (n m : ℕ) → Vec a n → Vec a m → Vec a (n + m)
concatVec .zero    m []       v₂ = v₂
concatVec (suc n) m (x :: v₁) v₂ = x :: concatVec n m v₁ v₂
```

The `concatVec` function pattern matches the first vector. As a result the matched constructors of the vector determine the pattern of the first length, i.e., for the empty vector it has to be `zero` and for the non-empty vector it has to be `suc n` for some `n`. The dot (`.`) in front of `zero` signifies that it is not a pattern to be matched, but a expression determined by another argument and the type signature. In this case the dot pattern is not essential, but there are cases, e.g., if two arguments are proven to be equal, where dot patterns are necessary, because otherwise it would be necessary to match on the same variable twice, which is not allowed.

To pattern match arbitrary expressions that are not arguments Agda provides the `with` construct as opposed to `case` in Haskell.

```
findVec : {ℓ : Level} {a : Set ℓ} {n : ℕ}
        → (a → Bool) → Vec a n → Maybe a
findVec p [] = nothing
findVec p (x :: v) with p x
findVec p (x :: v) | false = findVec p v
findVec p (x :: v) | true  = just x
```

The cases resulting from `with` are separated from the other arguments by a pipe (`|`). It is also possible to replace everything in front of the pipe with an ellipsis (`...`).

Just as in Haskell underscore (_) can be used as a wildcard pattern if an argument is irrelevant for the function.

Agda also allows the definition of operators. Any identifier that contains an underscore (_) becomes an operator. The number of underscores determines the arity of the operator. For example, if-then-else can be defined as a ternary operator instead of having to be built into the language:

```
if_then_else_ : {ℓ : Level} {a : Set ℓ}
              → Bool → a → a → a
if false then t else e = t
if true  then t else e = e
```

Reading the Agda manual[1] is advised for details on the associativity and precedence rules.

[1] *Agda's documentation* - `https://agda.readthedocs.io/en/v2.5.3/`

A.5 Data types and records

Algebraic data types are declared using the GADT style. The constructors are listed with function-like type signatures. For example, the data type `Maybe` can be declared with the following declaration:

```
data Maybe {ℓ : Level} (a : Set ℓ) : Set (suc ℓ) where
  nothing : Maybe a
  just : a → Maybe a
```

The type signatures for constructors are essentially written the same way as those for functions, but they have to deliver the type they are constructors of as a result and there are some restrictions to prohibit encodings of possibly non-terminating recursion (the type needs to be strictly positive).

Record syntax can be used to declare types that only have a single constructor. For example, a record for monads could be declared as follows:

```
record Monad {ℓ : Level} (M : Set ℓ → Set ℓ) : Set (lsuc ℓ) where
  constructor
    monad
  field
    return : {a : Set ℓ} → a → M a
    _>>=_ : {a b : Set ℓ} → M a → (a → M b) → M b

  join : {a : Set ℓ} → M (M a) → M a
  join mma = mma >>= (λ x → x)
```

The keyword `data` is exchanged with `record`. A constructor name for pattern matching can be given in the `constructor` block, which is optional. Everything listed within the `field` block represents an additional field or argument of the record. The names given to fields can be used to access the fields of a record. Definitions that are solely based on the fields of a record can be given inside the `record` block. Note that there can be several field blocks to allow the local import of other modules and the definition of supporting functions between different fields.

When working with a record its fields and associated definitions usually have to be qualified, e.g., `Monad.return m` has to be written to access the `return` field of some `m : Monad M`. To avoid the repeated qualification it is possible to open records:

```
open Monad
```

If the record being worked with is clear, it is also possible to open that specific record. For example, a specific monad `m : Monad M` can be opened with the following line:

```
open Monad m
```

This import then binds `return` to the `return` field of `m`. Hence, the programmer can now write `return` instead of `Monad.return m`.

A.6 Proofs of equality

In Agda proofs of equality are typically expressed with propositional equality. The module `Relation.Binary.PropositionalEquality` provides the type

```
data _≡_ {ℓ : Level} {A : Set ℓ} (x : A) : A → Set a where
  refl : x ≡ x
```

Propositional equality encodes the equality of two values on the type level. This equality is proven if `refl` can be given as evidence for the type-level equation. The module also exports all of the laws that are important to prove equalities:

```
sym : {ℓ : Level} {A : Set ℓ} {a b : A}
    → a ≡ b → b ≡ a

trans : {ℓ : Level} {A : Set ℓ} {a b c : A}
      → a ≡ b → b ≡ c → a ≡ c

subst : {ℓ₁ ℓ₂ : Level} {A : Set ℓ₁}
      → (f : A → Set ℓ₂) → {x y : A}
      → x ≡ y → f x → f y

cong : {ℓ₁ ℓ₂ : Level} {A : Set ℓ₁} {B : Set ℓ₂}
     → (f : A → B) → {x y : A}
     → x ≡ y → f x ≡ f y
```

The following code demonstrates how the associativity of addition among natural numbers can be proven using propositional equality:

```
data ℕ : Set zero where
  z : ℕ
  s : ℕ → ℕ

_+_ : ℕ → ℕ → ℕ
z + m = m
s n + m = s (n + m)

associative : (i j k : ℕ) → (i + j) + k ≡ i + (j + k)
associative z j k = refl
associative (s i) j k = cong (λ X → s X) (associative i j k)
```

The example first defines natural numbers (`ℕ`) in terms of zero (`z`) and successor (`s`) and then defines the addition operator (`_+_`). Finally, the function `associativity` proves associativity of addition. The function pattern matches on the first argument. In the zero case the equality becomes trivial and can be proven with `refl` directly, because the left- and right-hand side of the equality both partially evaluate to `j + k`. In the successor case both sides can be partially evaluated as follows:

```
s ((i + j) + k) ≡ s (i + (j + k))
```

Hence, congruence (`cong`) can be used to apply the law of associativity recursively inside of the successor constructor.

For proofs with additional, more complicated steps the standard library also provides a special notation to make equality proofs more readable. To use this notation the user has to open the submodule `≡-Reasoning`. Once the submodule is open it allows to write the `associativity` proofs as follows:

```
open ≡-Reasoning

associative : (i j k : ℕ) → (i + j) + k ≡ i + (j + k)
associative z j k = refl
associative (s i) j k = begin
  s ((i + j) + k)
    ≡⟨ cong (λ X → s X) (associative i j k) ⟩
  s (i + (j + k)) ■
```

A.7 Universal and existential quantification

Universally quantified propositions are simply functions with arguments. The argument of a function provides the universally quantified values. For example, the `associative` function from the previous section

```
associative : (i j k : ℕ) → (i + j) + k ≡ i + (j + k)
```

encodes

$$\forall\, i,j,k \in \mathbb{N}.\ (i+j)+k = i+(j+k).$$

There is also special syntax to highlight universal quantification. The for-all operator ($\forall$) can be used to introduce implicit variables without giving their type, given it can be derived from context.

```
replicate : ∀ {ℓ} {n} {A : Set ℓ} → (a : A) → Vec A n
replicate {n = zero}  a = []
replicate {n = suc n} a = a :: (replicate {n = n} a)
```

Existential quantification can be expressed through dependent products. Dependent products are products where the type of the second entry can depend on the value of the first entry. For example, the definition of `filter` on `Vec` requires the existence of another, yet to be determined, length for the resulting vector.

```
filter : {ℓ : Level} {n : ℕ} {A : Set ℓ}
       → (A → Bool) → Vec A n → Σ ℕ (λ m → Vec A m)

filter : {ℓ : Level} {n : ℕ} {A : Set ℓ}
       → (A → Bool) → Vec A n → ∃ λ (m : ℕ) → Vec A m
```

The type of dependent products is Σ. The second type signature above shows the alternative syntax using ∃ to suggest that the dependent product is used in the spirit of an existential quantification. The constructor for dependent products is `_,_`.

Appendix B

Overview of Agda formalisations and proofs

All of the Agda formalisations and proofs can be found in a publicly available Git repository[1]. This appendix gives an overview of which modules contain which definitions and proofs.

B.1 Formalisations

B.1.1 Formalisations of Haskell structures

Alternative ..... `Haskell.Alternative`
Applicative ..... `Haskell.Applicative`
Functor ..... `Haskell.Functor`
Graded applicative ..... `Haskell.Parameterized.Graded.Applicative`
Graded monad ..... `Haskell.Parameterized.Graded.Monad`
Indexed applicative ..... `Haskell.Parameterized.Indexed.Applicative`
Indexed monad ..... `Haskell.Parameterized.Indexed.Monad`
Monad ..... `Haskell.Monad`

B.1.2 Formalisations of polymonads

Polymonad (4.1) ..... `Polymonad.Definition`
Principal polymonad (4.8) ..... `Polymonad.Principal`
Unionable polymonad (4.10) ..... `Polymonad.Unionable`

[1] GitHub: *jbracker/polymonad-proofs* - `https://github.com/jbracker/polymonad-proofs`

B.1.3 Formalisations of category theory

Atkey's parameterised monad . . . . . `Theory.Monad.Atkey`
Bicategory . . . . . `Theory.TwoCategory.Bicategory`
Category . . . . . `Theory.Category.Definition`
Closed category . . . . . `Theory.Category.Closed`
Closed functor . . . . . `Theory.Functor.Closed`
Codiscrete category (6.3) . . . . . `Theory.Category.Examples.Codiscrete`
Concrete category (6.21) . . . . . `Theory.Category.Concrete`
Dependent product of categories . . . . . `Theory.Category.Dependent`
Discrete category (6.2) . . . . . `Theory.Category.Examples.Discrete`
Function extensionality . . . . . `Extensionality`
Functor . . . . . `Theory.Functor.Definition`
Lax 2-functor on strict 2-categories (6.16) . . . . . `Theory.TwoFunctor.Definition`
Lax 2-functor with constant 0-cell mapping . . . . . `Theory.TwoFunctor.ConstZeroCell`
Lax closed functor . . . . . `Theory.Functor.Closed`
Lax monoidal functor (6.10) . . . . . `Theory.Functor.Monoidal`
Graded lax monoidal functor (6.32) . . . . . `Theory.Haskell.Parameterized.Graded.LaxMonoidalFunctor`
Graded monad (6.24) . . . . `Theory.Haskell.Parameterized.Graded.Monad`
Kleisli triple (6.19) . . . . . `Theory.Monad.Kleisli`
Monad (6.18) . . . . . `Theory.Monad.Definition`
Monoid . . . . . `Theory.Monoid`
Monoidal category (6.5) . . . . . `Theory.Category.Monoidal`
Monoidal functor . . . . . `Theory.Functor.Monoidal`
Natural isomorphism (6.4) . . . . . `Theory.Natural.Isomorphism`
Natural transformation . . . . . `Theory.Natural.Transformation`
Parameterised monad (6.26) . . . . . `Theory.Haskell.Parameterized.Indexed.Monad`
Parameterised lax monoidal functor (6.34) . . . . . `Theory.Haskell.Parameterized.Indexed.LaxMonoidalFunctor`
Parameterised relative monad (6.29) . . . . . `Theory.Haskell.Parameterized.Relative.Monad`
Product of monoidal categories . . . . . `Theory.Category.Monoidal.Product`
Relative monad (6.20) . . . . . `Theory.Monad.Relative`
Strict 2-category (6.11) . . . . . `Theory.TwoCategory.Definition`
Subcategory . . . . . `Theory.Category.Subcategory`

B.2 Proofs

The listed proofs show how the different formalised structures are related to each other. Most of the proofs either show a one-to-one correspondence or an injection from one structure to the other. I use the following symbols to indicate which is the case:

- $A \leftrightarrow B$ – Proof that certain instances of A are in one-to-one correspondence with certain instances of B.
- $A \hookrightarrow B$ – Proof that there is an injection certain instances of A into B.

B.2.1 Proofs involving Haskell structures

- Atkey's parameterised monad $\leftrightarrow$ Haskell indexed monad:
  `Theory.Monad.Atkey.Properties.IsomorphicIndexedMonad`
- Functor $\leftrightarrow$ Haskell functor:
  `Theory.Functor.Properties.IsomorphicHaskellFunctor`
- Graded lax monoidal functor $\leftrightarrow$ Haskell graded applicative:
  `Theory.Haskell.Parameterized.Graded.LaxMonoidalFunctor.`
  `Properties.IsomorphicHaskellGradedApplicative`
- Graded monad $\leftrightarrow$ Haskell graded monad:
  `Theory.Haskell.Parameterized.Graded.Monad.`
  `Properties.IsomorphicHaskellGradedMonad`
- Haskell indexed monad $\hookrightarrow$ Haskell functor:
  `Haskell.Parameterized.Indexed.Functor`
- Haskell indexed monad $\hookrightarrow$ Haskell indexed applicative:
  `Haskell.Parameterized.Indexed.Applicative.FromIndexedMonad`
- Haskell indexed monad $\hookrightarrow$ Polymonad:
  `Haskell.Parameterized.Indexed.Polymonad`
- Haskell indexed monad $\hookrightarrow$ Unionable polymonad:
  `Haskell.Parameterized.Indexed.Unionable`
- Haskell indexed monad with phantom indices $\hookrightarrow$ Polymonad:
  `Haskell.Parameterized.Indexed.PhantomMonad`
- Haskell indexed monad with phantom indices $\hookrightarrow$ Principal polymonad:
  `Haskell.Parameterized.Indexed.PhantomMonad`
- Haskell indexed monad with phantom indices $\hookrightarrow$ Unionable polymonad:
  `Haskell.Parameterized.Indexed.PhantomMonad`

- Haskell graded monad ↪ Haskell functor:
  `Haskell.Parameterized.Graded.Functor`
- Haskell graded monad ↪ Haskell graded applicative:
  `Haskell.Parameterized.Graded.Applicative.FromGradedMonad`
- Haskell graded monad ↪ Polymonad:
  `Haskell.Parameterized.Graded.Polymonad`
- Haskell monad ↪ Polymonad:
  `Haskell.Monad.Polymonad`
- Haskell monad ↪ Principal polymonad:
  `Haskell.Monad.Principal`
- Lax monoidal functor ↔ Haskell applicative:
  `Theory.Functor.Monoidal.Properties.`
  `IsomorphicHaskellApplicative`
- Lax monoidal functor ↔ Haskell graded applicative:
  `Theory.Functor.Monoidal.Properties.`
  `IsomorphicGradedApplicative`
- Lax monoidal functor ↔ Haskell graded monad:
  `Theory.Functor.Monoidal.Properties.IsomorphicGradedMonad`
- Monad ↔ Haskell monad:
  `Theory.Monad.Properties.IsomorphicHaskellMonad`
- Parameterised lax monoidal functor ↔ Haskell Indexed Applicative:
  `Theory.Haskell.Parameterized.Indexed.LaxMonoidalFunctor.`
  `Properties.IsomorphicHaskellIndexedApplicative`
- Parameterised monad ↔ Haskell indexed monad:
  `Theory.Haskell.Parameterized.Indexed.Monad.`
  `Properties.IsomorphicHaskellIndexedMonad`
- Polymonad ↪ Haskell monad:
  `Haskell.Monad.Polymonad`

B.2.2 Proofs involving polymonads

- Bind operation are unique:
  `Polymonad.UniqueBinds`
- Haskell indexed monad ↪ Polymonad:
  `Haskell.Parameterized.Indexed.Polymonad`

- Haskell indexed monad ↪ Unionable polymonad:
  `Haskell.Parameterized.Indexed.Unionable`
- Haskell indexed monad with phantom indices ↪ Polymonad:
  `Haskell.Parameterized.Indexed.PhantomMonad`
- Haskell indexed monad with phantom indices ↪ Principal polymonad:
  `Haskell.Parameterized.Indexed.PhantomMonad`
- Haskell indexed monad with phantom indices ↪ Unionable polymonad:
  `Haskell.Parameterized.Indexed.PhantomMonad`
- Haskell graded monad ↪ Polymonad:
  `Haskell.Parameterized.Graded.Polymonad`
- Haskell monad ↪ Polymonad:
  `Haskell.Monad.Polymonad`
- Haskell monad ↪ Principal polymonad:
  `Haskell.Monad.Principal`
- Haskell monad ↪ Unionable polymonad:
  `Haskell.Monad.Unionable`
- Polymonad ↪ Haskell monad:
  `Haskell.Monad.Polymonad`
- Union of polymonads ↪ Polymonad:
  `Polymonad.Union`
- Union of polymonads ↪ Principal polymonad:
  `Polymonad.Union.Principal`
- Union of polymonads ↪ Unionable polymonad:
  `Polymonad.Union.Unionable`

B.2.3 Proofs within category theory

- Atkey's parameterised monad ↪ Lax 2-Functor:
  `Theory.TwoFunctor.Properties.FromAtkeyParameterizedMonad`
- Curried functor:
  `Theory.Functor.Properties.Curry`
- Functor ↪ Lax 2-functor:
  `Theory.TwoFunctor.Properties.FromFunctor`
- Graded monad ↔ Lax 2-functor:
  `Theory.TwoFunctor.Properties.IsomorphicGradedMonad`

- Parameterised lax monoidal functor ↔ Graded lax monoidal functor (6.36):
  `Theory.Haskell.Parameterized.Indexed.LaxMonoidalFunctor.Properties.IsomorphicGradedLaxMonoidalFunctor`
- Parameterised monad ↔ Lax 2-functor (6.28):
  `Theory.TwoFunctor.Properties.IsomorphicIndexedMonad`
- Parameterised monad ↔ Graded monad (6.27):
  `Theory.Haskell.Parameterized.Indexed.Monad.Properties.IsomorphicGradedMonad`
- Parameterised monad ↔ Atkey's parameterised monad:
  `Theory.Haskell.Parameterized.Indexed.Monad.Properties.IsomorphicAtkeyParameterizedMonad`
- Parameterised relative monad ↔ Parameterised monad (6.30):
  `Theory.Haskell.Parameterized.Relative.Monad.Properties.IsomorphicIndexedMonad`
- Parameterised relative monad ↔ Relative monad (6.31):
  `Theory.Haskell.Parameterized.Relative.Monad.Properties.IsomorphicRelativeMonad`
- Lax monoidal functor ↔ Graded lax monoidal functor (6.33):
  `Theory.Haskell.Parameterized.Graded.LaxMonoidalFunctor.Properties.IsomorphicLaxMonoidalFunctor`
- Lax monoidal functor ↔ Parameterised lax monoidal functor (6.35):
  `Theory.Haskell.Parameterized.Indexed.LaxMonoidalFunctor.Properties.IsomorphicLaxMonoidalFunctor`
- Lax monoidal functor ↔ Graded monad (6.25):
  `Theory.Functor.Monoidal.Properties.IsomorphicGradedMonad`
- Lax monoidal functor ↔ Monad (6.23):
  `Theory.Functor.Monoidal.Properties.IsomorphicMonad`
- Lax monoidal functor ↔ Lax 2-functor (6.17):
  `Theory.TwoFunctor.Properties.IsomorphicLaxMonoidalFunctor`
- Monad ↔ Lax 2-functor:
  `Theory.TwoFunctor.Properties.IsomorphicMonad`
- Monad ↔ Kleisli triple:
  `Theory.Monad.Kleisli`

List of Figures

3.1 Types of the bind and return operations for different monadic notions. 13
3.2 Laws of different monadic notions. 13
3.3 Types of the operators of different generalisations of applicatives. 20
3.4 Laws of different generalisations of applicatives. 20

4.1 Control flow within the diamond and associativity law for polymonads. 25
4.2 Example of a module header that enables the use of polymonads. 38

5.1 Graph produced by the separation algorithm from the example. . 63
5.2 Example of a module header that enables the use of supermonads and superapplicatives. 66

6.1 Hierarchy of categorical models for monadic notions. 98
6.2 Hierarchy of categorical models for applicative notions. 98

List of Tables

6.1 Overview of categorical formalisations for monadic notions. . . 85
6.2 Overview of categorical formalisations for applicative notions. . 92

List of definitions, propositions, and examples

4.1 Definition (Polymonad) . . . 24
4.2 Definition (Polymonad basis) . . . 25
4.3 Example (Identity polymonad) . . . 27
4.4 Example (Polymonad formed by a standard monad) . . . 27
4.5 Example (Polymonad formed by an indexed monad) . . . 28
4.6 Example (Polymonad formed by a graded monad) . . . 30
4.7 Proposition (Uniqueness of bind operations) . . . 31
4.8 Definition (Principal polymonad) . . . 32
4.9 Definition (Unionable polymonad) . . . 33
4.10 Proposition (Union of polymonads) . . . 33
4.11 Assumption . . . 34
4.12 Assumption . . . 34
4.13 Proposition (Polymonad union preserves principality) . . . 35

6.1 Definition (One-to-One Correspondence, $\leftrightarrow$) . . . 72
6.2 Definition (Discrete category) . . . 72
6.3 Definition (Codiscrete category) . . . 72
6.4 Definition (Natural isomorphism) . . . 73
6.5 Definition (Monoidal category) . . . 73
6.6 Example (Unit) . . . 74
6.7 Example (Monoid) . . . 74
6.8 Example (Set) . . . 74
6.9 Example (Endofunctors and natural transformations) . . . 74
6.10 Definition (Lax monoidal functor) . . . 75
6.11 Definition (Strict 2-category) . . . 77
6.12 Example (Unit) . . . 78
6.13 Example (2-Category of categories) . . . 78
6.14 Example (Discrete 2-category) . . . 78
6.15 Example (Monoid 2-category) . . . 78
6.16 Definition (Lax 2-functor) . . . 79
6.17 Example (Lax monoidal functors) . . . 79
6.18 Definition (Monad) . . . 80

6.19 Definition (Kleisli triple) 80
6.20 Definition (Relative monad) 82
6.21 Definition (Concrete category) 83
6.22 Definition (Faithful functor) 83
6.23 Proposition (Monads ↔ Lax Monoidal Functors) 86
6.24 Definition (Graded monad) 86
6.25 Proposition (Graded monads ↔ Lax monoidal functors) 87
6.26 Definition (Parameterised monad) 87
6.27 Proposition (Graded monads ↔ Parameterised monads) 88
6.28 Proposition (Parameterised monads ↔ Lax 2-functors) 88
6.29 Definition (Parameterised relative monad) 90
6.30 Proposition (Parameterised monads ↔ Parameterised relative monads) 91
6.31 Proposition (Relative monads ↔ Parameterised relative monads) 92
6.32 Definition (Graded lax monoidal functor) 93
6.33 Proposition (Lax monoidal functors ↔ Graded lax monoidal functors) 94
6.34 Definition (Parameterised lax monoidal functor) 95
6.35 Proposition (Lax monoidal functors ↔ Parameterised lax monoidal functors) 97
6.36 Proposition (Graded lax monoidal functors ↔ Parameterised lax monoidal functors) 97