Under consideration for publication in J. Functional Programming

Supermonads and superapplicatives

JAN BRACKER and HENRIK NILSSON
University of Nottingham,
School of Computer Science,
Functional Programming Laboratory,
Nottingham NG8 1BB, UK
(*e-mail:* {jzb, nhn}@cs.nott.ac.uk)

Abstract

Monads, applicative functors, and related notions of computation have been instrumental both for the theoretical study of effects and for practical programming with effects in a pure setting. Some languages, notably Haskell, even provide dedicated support for programming with such notions. Over time, practical needs as well as theoretical investigations have given rise to a number of generalisations. Examples include constrained and indexed monads. These generalisations may seem modest and obvious, but there are significant differences in terms of the categorical foundations. Further, language support is lacking, leading to mutually incompatible implementations with negative consequences for practical use. This raises the question exactly how these generalisations are related, and if there are unifying notions that can serve as a common foundation for implementation. In earlier work, as a partial answer, we introduced *supermonads*, a unifying notion for several popular generalisations of monads, along with language support for Haskell in the form of libraries and a type-checker extension. Building on that, this paper aims to provide a comprehensive answer by also introducing *superapplicatives*, extending the language support accordingly, and providing a detailed study of the categorical foundations for the various notions, culminating in unifying notions that show that supermonads and superapplicatives rest on sound theoretical foundations. The paper further provides examples and case studies illustrating the use of supermonads and superapplicatives in practice.

1 Introduction

A number of different notions of computation have been introduced to study effects and to structure effectful programming in a pure setting, notably *monads* (Moggi, 1988; Moggi, 1991; Wadler, 1992), *arrows* (Hughes, 2000), and *applicative functors* or simply *applicatives* (McBride & Paterson, 2008). Monads in particular are an integral part of Haskell and enjoy dedicated syntactic support in the form of the `do`-notation. Applicatives and arrows are also widely used, and extensions of the `do`-notation have been introduced in support (Paterson, 2001; Marlow *et al.*, 2016) confirming their growing importance.

Standard monads, arrows and applicatives cover a lot of ground. Nevertheless, a range of generalisations have been proposed, driven by both theoretical investigations and practical needs, generally in a quest for improved static correctness guarantees by adding type indices or constraints. Examples include session-types (Pucella & Tov, 2008), effect systems (Orchard & Petricek, 2014; McBride, 2011), and information flow control (Devriese &

Piessens, 2011). The practical development of these generalisations have generally taken place in the context of Haskell with GHC extensions. Indeed, some of the generalisations have been enabled by, or become more useful thanks to, recent type system extensions.

Unfortunately, as presently realised, these generalisations do not interoperate smoothly leading to a number of usability problems. In Haskell, computational notions such as monads or applicatives are each embodied by a type class allowing the programmer to use specific instances through a common API, including the `do`-notation. The generalised notions, however, beacuse of the type indices and constraints, require new classes, separate from the standard ones, and as a consequence also separate versions of standard library functions. This makes it harder both to read and write code and hampers code reuse. Further, if more than one variation of a computational notion is used within a module, it often becomes necessary to write the code in a way that makes it manifest which variation is used where, adding significant clutter.

These shortcomings are arguably Haskell-specific, but they point at a deeper problem: there are significant differences between the categorical foundations of the different generalisations. Consequently it is hardly surprising that direct realisations of the various variations, however done, are quite disparate, leading to issues like what was discussed above. Thus, while the programming language context for the present work is Haskell, the problem is general. It is therefore of interest to look for a single notion that captures as many of the monadic generalisations as possible in a uniform manner, along with a concrete realisation that fits with existing monadic support in the language in question. And for the same reasons, it is of interest to look for a unifying notion for applicatives.

In prior work (Bracker & Nilsson, 2015), we successfully integrated *polymonads* (Hicks *et al.*, 2014) into Haskell in an effort to mitigate the above problems. Polymonads are very general and have the benefit of being compositional. In its current form, however, the polymonad theory does not allow non-phantom indices or constraints on result types. Both of these restrictions exclude important use cases. Further, the polymonad laws are complex and less intuitive than the standard monad laws, creating an additional hurdle for end users.

More recently we explored a different approach we call *supermonads* (Bracker & Nilsson, 2016). Supermonads provide a unified representation covering a broad range of generalised monads in Haskell. The prototype implementation consists of a GHC type-checker plugin to assist with constraint solving and supporting class definitions and libraries. While supermonads are less general than polymonads in some respects, they do not have the specific limitations regarding indices and constraints discussed above, and they do cover all use cases that we found through a comprehensive literature survey and a survey of readily available Haskell libraries (Section 3). Moreover, the supermonad laws are straightforward generalisations of the standard monad laws.

The present article takes the work on supermonads further in two ways: we show that the approach also works for applicative functors by introducing *superapplicatives*, and we provide theoretical foundations for both supermonads and superapplicatives. To make this article reasonably self-contained, the material on supermonads draws from our earlier work. The specific contributions of this article in more detail are:

- Superapplicatives: a generalisation of applicative functors mirroring supermonads.
- Extensions of the supermonad plugin and libraries to support superapplicatives.

- Categorical models for a range of monadic and applicative notions of computation, with full formalisation of models and theorems in the proof assistant Agda.
- Work towards a unified categorical model to provide semantics and formal structure to supeapplicatives and supermonads.

The article also provides detailed examples and case studies as motivation and for illustration.

The rest of the article is structured as follows. Section 2 provides technical preliminaries, while sections 3 and 4 introduce and discuss various generalisations of monads and applicatives. Supermonads and superapplicatives are then introduced as unifying notions in Section 5, with Section 6 illustrating with a range of examples. Section 7 studies the categorical foundations of the various generalisations and their relationships, culminating in unifying categorical notions that provide a theoretical foundation for superapplicatives and supermonads. The implementation of the plugin for the GHC type checker, which helps resolving the constraints associated with superapplicatives and supermonads, is discussed in Section 8, while Section 9 evaluates our approach as presently implemented through a couple of case studies. Related work is discussed in Section 10 and, finally, Section 11 concludes and outlines future work.

2 Preliminaries

2.1 Nomenclature

Let us first fix the terminology for the generalised monadic notions under consideration as the terminology is somewhat varying both in our previous work (Bracker & Nilsson, 2015; Bracker & Nilsson, 2016) and the literature more broadly. We opt for what we consider to be consistent and well-established terms.

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

We will use the term *parametrised monad* collectively for standard, graded, and indexed monads.

The term “applicative functor” can be very unwieldy. Therefore, we will use the phrase “applicative” or “applicatives” synonymously with it from this point onward.

2.2 Haskell extensions

The Haskell code discussed and presented in this work requires several extensions of the language to compile. Therefore, we use the Glasgow Haskell Compiler[1] (GHC). The plugin

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

mechanism and language extensions used to implement supermonads or superapplicatives are only available in GHC.

The following list contains the most important language extensions our library and examples require. Each extension gives reference to the section of the GHC user's guide[2] that covers it.

`RebindableSyntax`**:** Allows syntactic sugar, such as the `do`-notation, to be translated in terms of user-specified functions, such as custom bind and sequence operations (Section 9.3.15).

`MultiParamTypeClasses`**:** Allows defining type classes with more than one argument (Section 9.8.1.1).

`TypeFamilies`**:** Facilitates type-level programming by effectively providing type-level functions. We require this extension to use associated type synonyms (Chakravarty *et al.*, 2005) for defining instance specific type-level information (Section 9.9).

`DataKinds`**:** Allows data types to be used on the type/kind level instead of just the value/type level (Section 9.10).

`PolyKinds`**:** Allows kinds that are polymorphic in the used kind and generalises kind inference to support polymorphic kinds (Section 9.11).

`ConstraintKinds`**:** Allows types to contain constraints by introducing a kind for constraints[3]. Together with associated type synonyms, this allows instance specific constraints (Section 9.14.5).

`KindSignatures`**:** Implied by `TypeFamilies`. Adds syntactic support for specifying the kind of a type; e.g., in the head of type class declarations or in the arguments of associated type synonyms (Section 9.15.4).

3 Monadic notions

To create a unified representation and theory of different generalisations of monads we need to decide which notions we aim to support. In this section we will introduce the three popular generalisations of monads that we aim to support alongside with standard monads: indexed, graded and constrained monads. Each notions will be introduced as it is usually represented and formulated in Haskell.

When we talk about monadic notions, we will often work with n-ary type constructors K and their arguments $a_1, \ldots, a_n$. We will refer to K as the *base constructor* and $a_1, \ldots, a_{n-1}$ as the *indices* of K. The *result type* of our monadic computation is a_n.

An overview of all the different operations and laws from the notions we will present is given in Figure 1 and Figure 2 respectively. We provide these tables as a reference point to directly compare different monadic notions with each other. All parts of the table will be discussed and explained in the following sections.

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

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

Monad	**Type constructor**	$\mathsf{bind} : \forall\ \alpha\ \beta.$	$\mathsf{return} : \forall\ \alpha.$
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.	$\diamond$	- Operation of the monoid E.
E	- Kind of the elements of a monoid.	ε	- Neutral element of the monoid E.

Fig. 1. Types of the operators of different generalisations of monads.

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

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

Fig. 2. Laws of different generalisations of monads.

3.1 Standard monads

To highlight the differences of the generalisations, we first reintroduce standard monads (Moggi, 1988). 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
```

Instances of this class are expected to obey the monad laws (Figure 3) with suitable a, f, g and m. As indicated by the superclass, each standard monad has an underlying functor

$$
\begin{array}{rcll}
\texttt{return}\, a \texttt{ >>= } f & = & f\ a & \text{(Left identity)} \\
m \texttt{ >>= return} & = & m & \text{(Right identity)} \\
(m \texttt{ >>= } f) \texttt{ >>= } g & = & m \texttt{ >>= } (\lambda x \rightarrow f\ x \texttt{ >>= } g) & \text{(Associativity)}
\end{array}
$$

Fig. 3. The monad laws.

that provides the `fmap` function to map the result type of a monadic computation with a function:

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

Section 5.7 will explain why we do not mention the recently added[4] `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 (Moggi, 1991; Wadler, 1992) as well as embedded domain specific languages (EDSLs) (Bracker & Gill, 2014).

3.2 Indexed monads

Session types are one application where the need for indexed monads arises (Pucella & Tov, 2008). The goal of session types is to verify the proper execution of a communication protocol on the type level. Lets assume we use the following types to express our communication protocol:

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

As we can see in `SomeProtocol` we can express simple protocols on the type level using the `Read`, `Write` and `Done` type. If we want to provide operations that execute this protocol we need to know what the executed protocol looks like and how much of the protocol is executed. A popular way to do this is using Hoare logic (Hoare, 1969) to show the state of the protocol before and after a operation. Thus, we can provide a type `Session i j a` that encodes the execution of a protocol: `i` and `j` represent the protocol before an after the computation and `a` is the result type of the computation. For example, the `read` and `write` operation could look as follows:

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

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

Since `Session` represents a side-effect it would be convenient if it formed a monad to allow us to sequence operations and utilise 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.

As demonstrated above indexed monads (Atkey, 2009; Wadler, 1994) add two indices to the type constructor. They respectively encode information about the internal state before and after the monadic computation is executed. Representations of indexed monads in Haskell usually have the following form:

```
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, the kind of `m` uses the polymorphic kind variable `k` to specify the kind of the indices. A kind variable is used, because it can be instantiated with any kind, allowing instances to choose their indices freely. The bind operation can only compose two computations if their respective post- and precondition match. The result of the bind operation has the precondition of the first computation and the postcondition of the second. The return operation has no side-effects. Therefore, pre- and postcondition can be anything, but are required to match.

The laws for indexed monads are exactly the same as for standard monads (Figure 3). Only the type of the involved variables a, f, g and m change. An overview of the types of the involved variables for each generalisation is given in Figure 2.

In contrast to standard monads, indexed monads have a family of underlying functors: One for each possible pair of indices. If we provide a generalised functor class, we would be able to express this as a superclass for `IndexedMonad`. The generalised functor class would have the following form:

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

If we reuse the standard functor class it is also possible to give suitable instances, but due to the requirement to have functor instance for all pairs of indices it is not possible to express this requirement as a superclass constraint anymore. If we have an indexed monad `M`, the family of associated functors can be given through the following instance of the standard `Functor` class:

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

Wadler demonstrated the usefulness of indexed monads for composable continuations (Wadler, 1994). Other applications of indexed monads include the above session types[5]

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

(Pucella & Tov, 2008) and typed state (Atkey, 2009). Several packages on Hackage provide implementations of indexed monads[6,7,8]. The *indexed* package provides the above examples of the indexed monad and functor type class.

3.3 Graded monads

To give an example of where the need for graded monads arises, we will look at vectors that encode their length as part of their type. 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. We can supply all of the functions that are well-known from lists for `Vector`, but once we try to give a monad instance for `Vector` a problem arises. For lists we can use `concatMap` to implement the bind operation:

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

For vectors the type of `concatMap` 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` cannot be used to implement a bind operation, although its type is similar to that of the bind operation.

A similar issue arises if we try to write down the type of `return` for `Vector`:

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

If we try to give an instance of `Monad` for `(Vector 1)` we do not get the generality we expect the resulting bind operation to have.

There are many more examples where the same problem arises: information flow control (Devriese & Piessens, 2011), heterogeneous state and fine-grained composable control of side-effects (Orchard & Petricek, 2014). All of them intend to encode certain information or invariants (vector length) about their computation in an additional index and then require to compose said index in the result type of the bind operation.

Graded monads (Wadler, 1998; Katsumata, 2014; Orchard & Petricek, 2014) introduce said additional index in their type constructor. This index contains a monoidal type that is used to encode all of the information or invariants of a computation. When two computations are composed using the bind operation their effects are merged 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[9,10] use associated type synonyms to express the neutral element (`Unit`) and the monoid operation (`Plus`):

[6] Hackage: *indexed* - `http://hackage.haskell.org/package/indexed`
[7] Hackage: *monad-param* - `http://hackage.haskell.org/package/monad-param`
[8] Hackage: *index-core* - `http://hackage.haskell.org/package/index-core`
[9] Hackage: *effect-monad* - `http://hackage.haskell.org/package/effect-monad`
[10] Hackage: *monad-param* - `http://hackage.haskell.org/package/monad-param`

```
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 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 associated constraint `Inv`.

Just like indexed monads, graded monads also have an underlying family of functors: One for each element of the monoid. Whether we can specify a superclass constraint for those functors again depends on whether we want to specify a generalised functor class or use the standard one. 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) with a, f, g and m of appropriate type.

3.4 Constrained monads

The final generalisation we want to support introduces constraints on the result types of the monadic computations.

A well-known example of this generalisation is Haskells representation of unordered finite sets `Set`[11] (Hughes, 1999). `Set` would forms a standard monad just like lists do, but cannot because it requires an ordering (`Ord`) on its elements to allow an efficient internal representation (Adams, 1993). 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
```

In Haskell we can allow this 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
```

Again the laws are the same as for standard monads (Figure 3).

Constrained monads necessitate the introduction of constrained functors to implement their underlying functor:

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

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

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

The introduction of this new functor type class is necessary, because unlike for indexed or graded monads that just introduce additional indices the standard functor type class cannot express constraints.

There are other examples of constrained monads, e.g., finite quantum vectors (Vizzotto *et al.*, 2006). Domain specific languages (Persson *et al.*, 2012; Bracker & Gill, 2014) provide a recurring need for constrained monads as well.

We are aware of the two packages `rmonad`[12] and `constraint-classes`[13] that provide support for constrained or restricted monads. A deep embedding together with normalisation can provide an alternative way to work with constrained monads without going beyond the standard monad class (Sculthorpe *et al.*, 2013).

4 Applicative notions

In Haskell a thorough discussion of functors and monads usually also includes applicatives. We will now develop and introduce the associated generalised applicatives of the monadic notions discussed in the previous section.

We are not aware of any literature or project that uses or requires the generalised applicatives we discuss in the following sections. However, we can derive the generalised applicative from each monadic notion. To do so, we look at how a standard applicative can be derived from a standard monad and use the same process on indexed, graded and constrained monads to get their corresponding applicatives. Each of the derivations refer to their formalisation in Agda. Further information on the Agda formalisation can be found in Section 7.1.

As in the previous section we provide an overview of all the different applicative operations and laws in Figure 4 and Figure 5 respectively. Both tables are discussed in the following sections and provide a reference point to compare the different applicative notions with each other.

4.1 Standard applicatives

Drawing from work on parser combinators by Swierstra and Duponcheel (1996), standard applicatives were introduced by McBride and Paterson in 2008. 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
```

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

Applicative	**Type constructor**	$\mathsf{ap} : \forall\ \alpha\ \beta.$	$\mathsf{pure} : \forall\ \alpha.$
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.

Fig. 4. Types of the operators of different generalisations of applicatives.

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

Fig. 5. Laws of different generalisations of applicatives.

`pure id <*>` u	$=$	u	(Identity)
`pure (.) <*>` u `<*>` v `<*>` w	$=$	u `<*>` $(v$ `<*>` $w)$	(Composition)
`pure` f `<*> pure` a	$=$	`pure` $(f\ a)$	(Homomorphism)
u `<*> pure` a	$=$	`pure` $(\$\ a)$ `<*>` u	(Interchange)

Fig. 6. The applicative laws.

Just as monads, applicatives need to fulfil as set of laws (Figure 6) and they also have a variety of use cases. The many popular applications include parsers, concurrency (Marlow *et al.*, 2014), database query languages (Giorgidze *et al.*, 2011) and other EDSLs.

For every standard monad `m` we can derive its applicative 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. We can use this derivation as a technique to derive generalised applicatives from the corresponding generalised monads.

4.2 Indexed applicatives

Deriving the applicative of an indexed monad delivers the expected result[14]:

```
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 did.

4.3 Graded applicatives

The graded applicative can also be derived as expected[15]:

```
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, we have to provide associated type synonyms and constraints to allow the declaration of the type-level monoid to be used by an instance.

[14] Agda proof: `Haskell.Parameterized.Indexed.Applicative.FromIndexedMonad`
[15] Agda proof: `Haskell.Parameterized.Graded.Applicative.FromGradedMonad`

4.4 Constrained applicatives

As for the previous two generalised applicatives, we can derive the constrained applicative from the notion of a constrained monad[16]:

```
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
```

We introduce a new set of associated constraints 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, we do not require our applicatives to be a superclass of our monads as we will explain in Section 5.7.

The formalised derivation[16] is working with the categorical representation of constrained monads and constrained applicatives. Therefore, reading Section 7.2.8, 7.3.5 and 7.4 may be helpful to understand the formalisation and why it applies.

5 Supermonads and superapplicatives

In the previous section we have seen several different generalisations of monads and applicatives. Although each notions can be represented and utilised in Haskell, problems arise once we try to use them in conjunction with each other. Each notion has a separate type class. Therefore, standard library functionality has to be duplicated for each type class. In addition, using several notions side by side in the same module can 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 operations requires manual disambiguation for each do-block or the use of qualified names.

Supermonads and superapplicatives have the goal to foster code reuse by obviating the need to give custom class definitions and adapted versions of the standard library functions for each separate 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 sections, we will first explore the general idea of how to represent each of the different generalisation with a fixed set of type classes (Section 5.1). We will then investigate why Haskell's type inference is insufficient when confronted with said set of type classes (Section 5.2). Based on the insights of that investigation, we will discuss how to remedy this shortcoming of the encoding (Section 5.3). Then, we present the type classes in their current form with and without support for constrained monads and applicatives (Section 5.4 and 5.5). Afterwards, we will explain why we opt for an additional encoding without support for constrained monads and applicatives (Section 5.6). Finally, we close

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

this section with a discussion of why we do not make superapplicatives a superclass of supermonads (Section 5.7).

A categorical classification of supermonads and superapplicatives is deferred to Section 7.3. Until then, it suffices to be aware the laws are exactly the same as for the notions that they intend to capture (Figure 2 and Figure 5); only the type of the involved variables changes.

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

First we need to understand why each notion covered in Section 3 and 4 requires a separate type class in Haskell. Their type classes follow this scheme:

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

Here, `m` is the type constructor that is 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 associated type constructor 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[17,18] and the work on polymonads (Hicks *et al.*, 2014). 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
```

In this generalised type class, when defining an instance, `m`, `n` and `p` can 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.

With this approach, we are forced to put the monadic operations into two separate type classes, because it would be unclear which of the three type constructors, `m`, `n` or `p`, to use for the return operation. It may even be the case that non of the partially applied constructors suit the return operation.

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

The same technique can be used to provide a type class for the `<*>` 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, we can reuse the `Return` type class for this purpose.

Examples of how instances for the different monadic notions can be given are presented in Section 6.

[17] Hackage: *monad-param* - `http://hackage.haskell.org/package/monad-param`
[18] *Parameterized Monads in Haskell (13. July 2007)* - `http://comonad.com/reader/2007/parameterized-monads-in-haskell/`

5.2 Insufficient type inference

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

In Haskell 2010[19], type inference is guaranteed for almost all features of the language. However, to implement the `Bind` and `Applicative` class we require GHC's language extension `MultiParamTypeClasses`. Since both classes have three distinct, unrelated arguments, GHC's type inference has no way of knowing that we intend them all 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.

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.

Let us illustrate 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 will infer 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 jeopardizing the runtime behaviour of `plus3`.

Our case studies in Section 9 show that this issue is commonplace in programs involving the above generalised representation. In addition, they show that if our monadic notion is parametrised, we cannot infer the type of the indices 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 us from choosing an appropriate instance to base this inference on.

5.3 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
```

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

```
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 right choice of return operation restores type inference in some, but not all, cases.

To see how well Kmett's approach works we retrofitted our first case study (Section 9.1) to use his library[20]. There were still many situations that required manual type annotations to solve ambiguous variables. In addition, we also had to choose the correct return operation (`return` or `returnM`) 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 remaining constructor. If we were to add more functional dependencies to address this issue, they would quickly become so restrictive that only standard monad instances are possible.

We now have an understanding of which capabilities were lost by introducing the new type classes:

- We lost the connection between the bind and return operation as well as the ap and return operation. Both connections were previously encoded through the single type class that contained either pair of operations.
- We also lost the knowledge that all three type constructors in the `Bind` or `Applicative` class are partial applications of the same type constructor.
- Finally, we lost the ability to infer the indices through unification with the different operations type signature, because this is only possible if we know which instance we are working with.

To our knowledge it is only possible to address these issues inside Haskell itself, if we add manual type annotation. Our case studies in Section 9 show that providing these annotations is a tedious and extensive task. Therefore, we introduce the unifying notions of a *supermonad* and *superapplicative* as a *language extension*.

5.4 The supermonad and superapplicative type classes

We embody the notion of a supermonad through the `Bind` and `Return` classes along the lines seen above. Superapplicatives are represented through the `Applicative` and `Return` classes. We then extend the type system 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, allowing the GHC constraint solver to ask for help when ambiguities, such as the ones mentioned above, arise. The goal of our plugin is to allow GHC to infer the types of any supermonad or superapplicative computation that it would have

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

been able to infer if that computation was using one of the specialised type classes from Section 3 or Section 4, thus restoring any type inference capabilities lost in the process of generalisation.

Our 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
```

Generalizing from standard monads and applicatives, we also introduce `Functor` constraints on each of the partially applied type constructors. As we discussed in Section 3 and 4 each notion has an associated functor or family of functors.

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

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

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

- 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 our plugin. The programmer is still required to ensure that their instances follow the monad and applicative laws and that all associated functor instances exist.

It might 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, firstly, the notion of supermonads or superapplicatives as such just refers to our particular implementation of generalised monads and applicatives in Haskell. There may be different ways to realise a generalised notion or some essentially equivalent notion in the future. What we have established is that there is at least one practical way of integrating a unified representation of monadic and applicative notions into Haskell. Secondly, the approach we have taken 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 can invalidate contextual constraints occurs also for language extensions such as overlapping instances.

All source code of our implementation and library is available as open source[21].

5.5 Adding constraints

To support constrained monads and applicatives, we need to make the `Bind`, `Applicative` and `Return` class 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 we could simply reuse the `Functor` type class from the standard library, but constrained monads and applicatives also require constrained functors. Therefore, we also need to introduce a replacement for the standard functor class.

```
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
```

5.6 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.

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

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 may not be possible, because it is impossible to determine the necessary constraints and therefore the programmer is required to list all of the `BindCts`, `ApplicativeCts`, `ReturnCts` and `FunctorCts` constraints that occur inside of the code. 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 can become irritating quickly. This can 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 monads:

```
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 will be sufficient to implement the member functions in a given instance. It is especially unclear which constrained result types will be involved, if we were to extend the above definition for constrained monads. Therefore, we need to allow custom individual constraints 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 might 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 problem vanishes in code that is not polymorphic in the used supermonad or applicative, because, in that case, 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, our library offers two representations of supermonads and applicatives: 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.

We are aware that 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, we 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.

We noticed another implication while implementing the standard library functions based on our new type classes. Type signatures that only involve two type constructors can have different sets of constraints depending on their implementation. 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. Our library makes an effort to be systematic when it comes to this choice.

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

Note that our `Applicative` class is not a superclass of our bind class. The reason for this is that the representation of applicatives in Haskell is different (but equivalent) to their representation in category theory (see Section 7.4). In Haskell we have two operations that 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 7.2.4). 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.

To exemplify, we will instantiate each of these functions for the Set data type from the previous section (6.4).

```
(<*>) :: (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 `<*>` function 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. Hence, the only way to create a set of functions is 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 (Swierstra & Duponcheel, 1996; McBride & Paterson, 2008).

We cannot put both interfaces into one type class for two reasons. Firstly, combining the interfaces would force the user to always implement both interfaces, although this may not be possible or useful depending on the involved constraints. Secondly, 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 obvious to the caller immediately.

Therefore, we decide to give the programmer 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.

6 Examples of supermonads and superapplicatives

In Section 3 and Section 4 we presented the monadic and applicative notions that supermonads aim to support. In this section, we demonstrate how the motivating examples from the above sections instantiate supermonads and superapplicatives.

6.1 `Maybe` and the state transformer: Standard monads

A standard monad, e.g., `Maybe`, can instantiate our 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 `Monad` instance forms the supermonad.

Unfortunately, a direct implementation in terms of the original notion does not always work: if we have a standard monad that is defined in terms of another monad, e.g., a monad transformers, it is necessary to reimplement the bind and return function to ensure that the nested monadic notion is also a supermonad. As an example we give the implementation of the `StateT` monad transformer.

```
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' )
```

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

Although we can give monad transformer `Bind` and `Return` instances, using the standard definition of functions like `lift`, `get` or `put` with supermonads is not possible, because they still have a `Monad` constraint that requires standard monads instead of supermonads. There is no problem implementing these functions for supermonads, but generalizing their type class abstractions to use supermonads has the practical implications that we discussed in Section 5.6.

6.2 Fixed-length vectors: a graded monad

To demonstrate how a graded monad instantiates the supermonad interface, we revisit the `Vector` data type from the introduction of graded monads in Section 3.3. For comparison we will present the graded monad type class and instance that is provided by the `effect-monad` package[22]. We will then implement the supermonad in terms of the representation from that package to highlight the changes when transitioning from one representation to the other.

Before we can implement the instances for `Vector` we need to introduce type-level natural numbers and the associated type-level function to multiply them:

```
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`. The `Vector` data type provides the following interface:

```
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 for lifting a value-level constructor to the type level. We omit the details of the implementation for brevity[23].

The `effect-monad` package provides the following type class for graded monads:

[22] Hackage: *effect-monad* - `http://hackage.haskell.org/package/effect-monad`
[23] *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 (f :: k) (g :: k) :: k
  type Inv m (f :: k) (g :: k) :: Constraint
  type Inv m f g = ()

  return :: a -> m (Unit m) a
  (>>=)  :: (Inv m f g)
         => m f a -> (a -> m g b) -> m (Plus m f g) 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 we can implement the `Effect` instance 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, we can see that the original implementation of bind and `return` can be reused without alteration. Note that we cannot 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 we are easily able to provide the `Applicative` instance for `Vector`. The `effect-monad` package does not provide an interface for graded applicatives.

As mentioned in the beginning, we implemented `Bind`, `Applicative` and `Return` 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.

6.3 Sessions types: an indexed monad

For our example involving indexed monads we chose to instantiate supermonad instances for the `Session` indexed monad from the `simple-sessions` package[24]. We gave a brief introduction to the `Session` type in our motivational example to introduce indexed monads (Section 3.2). The `Session` monad implements session types and uses the implementation of indexed monads provided by the `indexed` package[25]. The `indexed` package provides a complete type class hierarchy for indexed monads and indexed 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. We can give the supermonad instances for this notion by partially applying the `Session` type constructor in the instance head and reusing the existing instances.

```
instance Functor (Session i j) where
  fmap = imap
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 our graded monad example.

6.4 The `Set` constrained monad

To give an example of a constrained monad, we implement the supermonad instances for the `Set` data type from the introductory example (Section 3.4). We are required to

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

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`[26] 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.foldr S.union S.empty ( 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. We omit the `Applicative` instance, because, as discussed in Section 5.6, it is not very useful for `Set`.

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

We have seen that applicatives can be generalised in the same ways 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 maximal resource usage for a set of concurrent processes. One application is deadlock avoidance using the Banker's algorithm (Dijkstra, n.d.).

The Banker's algorithm can be used in a setting with a fixed set of different kinds of resources, each with 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 will only work 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, it is easy to see that this might not always be the case due to simple mistakes. 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.

We illustrate how this can be done for processes expressed monadically using an indexed monad, where the pre-state is a pair of the maximal resource need and resources held prior

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

to a 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, we introduce type-level functions to compute the maximum and to check the order of type-level naturals:

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

We reuse the type-level natural numbers from our graded monad example.

For our example, we are assuming two kinds of resources, `A` and `B`. For each process, we need to keep track of the current maximal use and the current allocation for each resource, that is four quantities:

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

Our 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 process representation is indexed on this state representation, with the first index representing the pre-state and the second one the post-state:

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

We further assume 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. The claim operation also returns 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 indexed monad, but here we are only concerned with the monadic interface.

The next step is to introduce the notion of an *executable* process constituting a (possible) unit of scheduling. We only provide an applicative interface, as the objective is to run these concurrently to the extent desirable, subject to constraints related to resource allocations. It is indexed by the maximal resource need and forms a graded applicative as the combined need of two executable processes is scheduled as a single unit if the maximum of the constituents' needs (the representation of the need and the maximum operation forms a monoid). Whether or not to schedule a single unit or to schedule the constituents concurrently is for the scheduler to decide, and we do not concern ourselves further with that. However, the needs of whichever unit that ends up being scheduled concurrently are what is communicated to 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 prior to 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 we need to ensure that the available resources of each kind suffice to satisfy the maximal need of any one process, reflected by the overall maximal need. The type `NatV` and the class `ToNatV` mediate between value- 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.

7 Relationship to category theory

The original formalisation of supermonads in our previous work (Bracker & Nilsson, 2016) was based solely on our 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. We want to disregard our previous formalisation and replace it with a model for supermonads and superapplicatives that is founded in category theory.

Consequently, the notion of supermonad and superapplicative should only refer to our unified encoding and implementation of the different monadic and applicative notions in Haskell. The semantics and the supported generalisations should be characterised by the categorical notions presented in this section.

In Section 7.2 we will introduce prerequisite categorical definitions that are required to understand the following sections. Based on these definitions, we show how the different monadic and applicative notions in Haskell relate to them and even develop categorical adaptations that precisely capture them in Section 7.3 and 7.4. The discussion will lead to categorical structures that capture all of the monadic and applicative notions, respectively. As a result we are able to present a hierarchy of monadic and applicative notions that shows how all of the different notions are related to each other in Section 7.5.

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

As the target audience for this work are functional programmers we only expect a basic understanding of category theory. To understand the following sections we expect the reader to be familiar with the concepts of categories[27], functors[28] and natural transformations[29]. All other required notions will be introduced.

[27] Agda formalisation: `Theory.Category.Definition`
[28] Agda formalisation: `Theory.Functor.Definition`
[29] Agda formalisation: `Theory.Natural.Transformation`

We will use the following naming conventions:

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

7.1 Proofs in the following sections

Unless noted otherwise all definitions and results in the following sections are formalised and verified using the proof assistant Agda[30] (Norell, 2007) in version 2.5.3. This gives us confidence in our results, but also implies some limitations.

Many proofs in the following sections relate structures in Haskell to category theory. To our knowledge there is no complete formal or categorical model of Haskell that we can base our proofs on. Therefore, we formalised the Haskell structures that we work with in the category Set of types and total functions. Whenever we speak of Haskell structures that are equivalent or in one-to-one correspondence to categorical structures our proofs are based on a formalisation of these structures in Set.

To relate our proofs back to Haskell we need to assume that the Haskell structures we talk about are terminating, total and do not contain bottom. These assumptions are necessary because Set and Agda do not reflect any of these properties. We do not think that these limitations weaken our results, because proper instances of monads or applicatives usually fulfil all of these requirements.

The Agda source code of our proofs can be found in a public Git repository[31]. Whenever we refer to a proof or formalisation the Agda module containing it will be referenced as a footnote.

7.2 Introduction of prerequisite categorical structures

We will give an intuition and a full definition for each prerequisite structure here. The intuition should be enough to understand the following sections using the introduced categorical structures.

7.2.1 Discrete and codiscrete collections of morphisms

A category is discrete if the identity morphisms for each object are the only morphisms.

Definition 7.2.1 (*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

[30] *The Agda Wiki* - `http://wiki.portal.chalmers.se/agda`
[31] GitHub: *jbracker/polymonad-proofs* - `https://github.com/jbracker/polymonad-proofs`

- $\mathsf{Hom}_{\mathbb{C}}(a,b) = \emptyset$ 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 7.2.2 (*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 $a,b \in \mathsf{Obj}_{\mathbb{C}}$.

7.2.2 Natural isomorphisms

A natural transformation $\eta : F \stackrel{\cdot}{\longrightarrow} G$ is a natural isomorphism[32] if its components $\eta_a : F(a) \to G(a)$ are guaranteed to have an inverse $\eta_a^{-1} : G(a) \to F(a)$. In other words, if a natural transformation provides a mapping between two functors then a natural isomorphism provides a "bijective" mapping between them.

Definition 7.2.3 (*Natural isomorphism*)
Let $\mathbb{C}$ and $\mathbb{D}$ be categories and $F, G : \mathbb{C} \longrightarrow \mathbb{D}$ functors between them. A natural transformation $\eta : F \stackrel{\cdot}{\longrightarrow} G$ 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. We denote η to be a natural isomorphism as $\eta : F \stackrel{\cong}{\longrightarrow} G$.

7.2.3 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 exists and that they behave as expected when composed. In contrast, the objects have no associated laws or structure. A *monoidal category*[33] (Bénabou, 1963) 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 7.2.4 (*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 $\alpha : ((- \otimes -) \otimes -) \stackrel{\cong}{\longrightarrow} (- \otimes (- \otimes -))$ with components of the form $\alpha_{a,b,c} : (a \otimes b) \otimes c \to a \otimes (b \otimes c)$ called the *associator*,

[32] Agda formalisation: `Theory.Natural.Isomorphism`
[33] Agda formalisation: `Theory.Category.Monoidal`

- 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*,

such that the diagram for the *triangle identity*

$$\begin{tikzcd}
(a \otimes 1_{\mathbb{C}}) \otimes b \arrow[rr, "\alpha_{a,1_{\mathbb{C}},b}"] \arrow[dr, "\rho_a \otimes \mathsf{id}_b"'] & & a \otimes (1_{\mathbb{C}} \otimes b) \arrow[dl, "\mathsf{id}_a \otimes \lambda_b"] \\
& a \otimes b &
\end{tikzcd}$$

and the diagram for the *pentagon identity*

$$\begin{tikzcd}
& (a \otimes b) \otimes (c \otimes d) \arrow[dr, "\alpha_{a,b,c \otimes d}"] & \\
((a \otimes b) \otimes c) \otimes d \arrow[ur, "\alpha_{a \otimes b,c,d}"] \arrow[d, "\alpha_{a,b,c} \otimes \mathsf{id}_d"'] & & (a \otimes (b \otimes (c \otimes d))) \\
(a \otimes (b \otimes c)) \otimes d \arrow[rr, "\alpha_{a,b \otimes c,d}"'] & & a \otimes ((b \otimes c) \otimes d) \arrow[u, "\mathsf{id}_a \otimes \alpha_{b,c,d}"']
\end{tikzcd}$$

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

Example 7.2.1 (*Unit*)
A trivial example of a monoidal category is the unit category **1** with exactly one object and morphism.[34]

Example 7.2.2 (*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.[35]

Example 7.2.3 (*Set*)
The category Set is another example. The cartesian product provides the tensor product and unit its tensor unit.[36]

Example 7.2.4 (*Endofunctors and natural transformations*)
Given a category $\mathbb{C}$ we can form the monoidal category $[\mathbb{C}, \mathbb{C}]_\circ$. 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 $\mathbb{C}$.[37] Note that in the resulting monoidal category the associator and unitors are all strict, i.e., they are identities.

[34] Agda proof: `Theory.Category.Monoidal.Examples.Unit`
[35] Agda proof: `Theory.Category.Monoidal.Examples.Monoid`
[36] Agda proof: `Theory.Category.Monoidal.Examples.SetCat`
[37] Agda proof: `Theory.Category.Monoidal.Examples.FunctorWithComposition`

7.2.4 Lax monoidal functors

A lax monoidal functor[38] (Bénabou, 1963) 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.

Just like 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 7.2.5 (*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 $\mu_{a,b} : F(a) \otimes_{\mathbb{D}} F(b) \overset{\cdot}{\longrightarrow} F(a \otimes_{\mathbb{C}} b)$ for all $a, b \in \mathsf{Obj}_{\mathbb{C}}$,

such that the diagram for *associativity*

$$\begin{tikzcd}
(F(a) \otimes_{\mathbb{D}} F(b)) \otimes_{\mathbb{D}} F(c) \arrow[r, "\alpha^{\mathbb{D}}_{F(a),F(b),F(c)}"] \arrow[d, "\mu_{a,b} \otimes_{\mathbb{D}} \mathsf{id}_{F(c)}"'] & F(a) \otimes_{\mathbb{D}} (F(b) \otimes_{\mathbb{D}} F(c)) \arrow[d, "\mathsf{id}_{F(a)} \otimes_{\mathbb{D}} \mu_{b,c}"] \\
F(a \otimes_{\mathbb{C}} b) \otimes_{\mathbb{D}} F(c) \arrow[d, "\mu_{a \otimes_{\mathbb{C}} b, c}"'] & F(a) \otimes_{\mathbb{D}} F(b \otimes_{\mathbb{C}} c) \arrow[d, "\mu_{a, b \otimes_{\mathbb{C}} c}"] \\
F((a \otimes_{\mathbb{C}} b) \otimes_{\mathbb{C}} c) \arrow[r, "F\left(\alpha^{\mathbb{C}}_{a,b,c}\right)"'] & F(a \otimes_{\mathbb{C}} (b \otimes_{\mathbb{C}} c))
\end{tikzcd}$$

and the diagrams for *left* and *right unitality*

$$\begin{tikzcd}
1_{\mathbb{D}} \otimes_{\mathbb{D}} F(a) \arrow[r, "\eta \otimes_{\mathbb{D}} \mathsf{id}_a"] \arrow[d, "\lambda^{\mathbb{D}}_{F(a)}"'] & F(1_{\mathbb{C}}) \otimes_{\mathbb{D}} F(a) \arrow[d, "\mu_{1_{\mathbb{C}},a}"] \\
F(a) & F(1_{\mathbb{C}} \otimes_{\mathbb{C}} a) \arrow[l, "F(\lambda^{\mathbb{C}}_a)"]
\end{tikzcd}
\qquad
\begin{tikzcd}
F(a) \otimes_{\mathbb{D}} 1_{\mathbb{D}} \arrow[r, "\mathsf{id}_a \otimes_{\mathbb{D}} \eta"] \arrow[d, "\rho^{\mathbb{D}}_{F(a)}"'] & F(a) \otimes_{\mathbb{D}} F(1_{\mathbb{C}}) \arrow[d, "\mu_{a,1_{\mathbb{C}}}"] \\
F(a) & F(a \otimes_{\mathbb{C}} 1_{\mathbb{C}}) \arrow[l, "F(\rho^{\mathbb{C}}_a)"]
\end{tikzcd}$$

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. In this article we only require 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.

[38] Agda formalisation: `Theory.Functor.Monoidal`

7.2.5 Strict 2-categories

Strict 2-categories (Bénabou, 1965; Bénabou, 1967; Kelly & Street, 1974) are a generalisations of categories.

A strict 2-category[39] $\mathbb{C}$ has a collection of objects (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 (1-cells) are provided by the objects of the homomorphism categories. The morphisms of the homomorphism categories represent morphisms between morphisms (2-cells):

$$\begin{tikzcd} a \arrow[rr, bend left, "f\,:\,a\to b", ""{name=U, below}] \arrow[rr, bend right, "g\,:\,a\to b"', ""{name=D}] & & b \arrow[Rightarrow, from=U, to=D, "\eta\,:\,f\Rightarrow g"] \end{tikzcd}$$

a,b,c	object or 0-cell
f,g,h	morphism or 1-cell
η,ι	2-cell

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

$$\begin{tikzcd} a \arrow[r, "f"'] \arrow[rr, bend left, "g\circ f"] & b \arrow[r, "g"'] & c \end{tikzcd}$$

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

$$\begin{tikzcd} \bullet \arrow[r, "f"] \arrow[d, phantom, "\eta\Downarrow"] & \bullet \arrow[r, "g"] \arrow[d, phantom, "\Downarrow\iota"] & \bullet & & \bullet \arrow[r, "g\circ f"] \arrow[d, phantom, "\Downarrow\iota\circ^h\eta"] & \bullet \\ \bullet \arrow[r, "f'"'] & \bullet \arrow[r, "g'"'] & \bullet \arrow[r, phantom, "\longmapsto"] & & \bullet \arrow[r, "g\circ f'"'] & \bullet \end{tikzcd}$$

The second is the vertical composition provided in each of the homomorphism categories:

$$\begin{tikzcd} a \arrow[r] & b \arrow[dd, Rightarrow, bend left=60, "\iota\circ^v\eta"] \\ \Downarrow\eta & \\ a \arrow[r] & b \\ \Downarrow\iota & \\ a \arrow[r] & b \end{tikzcd}$$

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 7.2.6 (*Strict 2-category*)

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

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

We call the elements in $\mathsf{Obj}_{\mathbb{C}}$ *0-cells* and for any $a,b \in \mathsf{Obj}_{\mathbb{C}}$ we call the elements in $\mathsf{Obj}_{\mathbb{C}(a,b)}$ *1-cells* and those in $\mathsf{Hom}_{\mathbb{C}(a,b)}(-,-)$ *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 7.2.5 (*Unit*)
A trivial example is the unit 2-category $\mathbf{1}$ with exactly one 0-cell, one 1-cell and one 2-cell.[40]

Example 7.2.6 (*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.[41]

Example 7.2.7 (*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. We denote this category $\mathsf{Disc}_{\mathbb{C}}$.[42]

[40] Agda proof: `Theory.TwoCategory.Examples.Unit`
[41] Agda proof: `Theory.TwoCategory.Examples.Functor`
[42] Agda proof: `Theory.TwoCategory.Examples.DiscreteHomCat`

Example 7.2.8 (*Monoid 2-category*)
Given a monoid $(M, \diamond, e)$ we can form a strict 2-category Mon^2_M with exactly one 0-cell. The 1-cells are the elements of the carrier M (endomorphisms) and the 2-cells are discrete.[43]

7.2.6 *Lax 2-functors*

A lax 2-functor[44] (Bénabou, 1965; Bénabou, 1967) 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 generalizing these laws, they may stay equalities or become 2-cell isomorphisms. Since we are talking about a *lax* 2-functor these 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 we need 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.

Definition 7.2.7 (*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:

$$\begin{tikzcd}
F_{a,b}(f) \circ_{\mathbb{D}} 1_{F(a)} \arrow[r, Rightarrow, "\mathsf{id}_{F_{a,b}(f)} \circ^h_{\mathbb{D}} \eta_x"] \arrow[d, Rightarrow, "\mathsf{id}_{F_{a,b}(f)}"'] & F_{a,b}(f) \circ_{\mathbb{D}} F_{a,a}(1_a) \arrow[d, Rightarrow, "\mu_{a,a,b}(1_a,f)"] \\
F_{a,b}(f) & F_{a,b}(f \circ_{\mathbb{C}} 1_a) \arrow[l, Rightarrow, "F_{a,b}(\mathsf{id}_f)"]
\end{tikzcd}$$

$$\begin{tikzcd}
1_{F(a)} \circ_{\mathbb{D}} F_{a,b}(f) \arrow[r, Rightarrow, "\eta_x \circ^h_{\mathbb{D}} \mathsf{id}_{F_{a,b}(f)}"] \arrow[d, Rightarrow, "\mathsf{id}_{F_{a,b}(f)}"'] & F_{b,b}(1_b) \circ_{\mathbb{D}} F_{a,b}(f) \arrow[d, Rightarrow, "\mu_{a,b,b}(f,1_b)"] \\
F_{a,b}(f) & F_{a,b}(1_b \circ_{\mathbb{C}} f) \arrow[l, Rightarrow, "F_{a,b}(\mathsf{id}_f)"]
\end{tikzcd}$$

[43] Agda proof: `Theory.TwoCategory.Examples.Monoid`
[44] Agda formalisation: `Theory.TwoFunctor.Definition`

$$\begin{tikzcd}
F_{c,d}(h) \circ_{\mathbb{D}} (F_{b,c}(g) \circ_{\mathbb{D}} F_{a,b}(f)) \arrow[r, Rightarrow, "\mathsf{id}_{(F_{c,d}(h) \circ_{\mathbb{D}} F_{b,c}(g) \circ_{\mathbb{D}} F_{a,b}(f))}"] \arrow[d, Rightarrow, "\mathsf{id}_{F_{c,d}(h)} \circ^{h}_{\mathbb{D}} \mu_{a,b,c}(f,\, g)"'] & (F_{c,d}(h) \circ_{\mathbb{D}} F_{b,c}(g)) \circ_{\mathbb{D}} F_{a,b}(f) \arrow[d, Rightarrow, "\mu_{b,c,d}(g,\, h) \circ^{h}_{\mathbb{D}} \mathsf{id}_{F_{a,b}(f)}"] \\
F_{c,d}(h) \circ_{\mathbb{D}} F_{a,c}(g \circ_{\mathbb{C}} f) \arrow[d, Rightarrow, "\mu_{a,c,d}((g \circ_{\mathbb{C}} f),\, h)"'] & F_{b,d}(h \circ_{\mathbb{C}} g) \circ_{\mathbb{D}} F_{a,d}(f) \arrow[d, Rightarrow, "\mu_{a,b,d}(f,\, (h \circ_{\mathbb{C}} g))"] \\
F_{a,d}(h \circ_{\mathbb{C}} (g \circ_{\mathbb{C}} f)) \arrow[r, Rightarrow, "\mathsf{id}_{F_{a,d}(h \circ_{\mathbb{C}} g \circ_{\mathbb{C}} f)}"'] & F_{a,d}((h \circ_{\mathbb{C}} g) \circ_{\mathbb{C}} f)
\end{tikzcd}$$

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 7.2.9 (*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.[45]

7.2.7 *Relative monads*

Relative monads[46] (Altenkirch *et al.*, 2010) are a generalisation of categorical monads that replaces the underlying endofunctor with an arbitrary functor, i.e., allowing monads between different categories.

A monad in a category $\mathbb{C}$ is usually defined in terms of an endofunctor $T : \mathbb{C} \longrightarrow \mathbb{C}$ and a morphism $\mu_x : T(T(x)) \to T(x)$ natural in $x \in \mathsf{Obj}_{\mathbb{C}}$ called join (see Definition 7.3.1). This definition does not allow generalizing T to a non-endofunctor, because the join operation applies T twice.

Relative monads are instead based on an alternative representation of monads based on the Kleisli-extension instead of join:

$$(-)^* : (a \to T(b)) \to (T(a) \to T(b))$$

$(-)^*$ closely resembles the bind operation Haskell programmers are used to, if we swap the arguments:

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

The Kleisli-extension allows replacing the endofunctor T with a non-endofunctor $T : \mathbb{C} \longrightarrow \mathbb{D}$. However, $a \to T(b)$ now requires a way to map the object $a \in \mathsf{Obj}_{\mathbb{C}}$ into the codomain $\mathsf{Obj}_{\mathbb{D}}$ to remain a valid morphism with $\mathbb{D}$. Therefore, relative monads introduce an additional functor $J : \mathbb{C} \longrightarrow \mathbb{D}$ that only serves the purpose of transferring objects into T's codomain. Thus, leading to the following Kleisli-extension and return morphism:

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

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

[45] Agda proof: `Theory.TwoFunctor.Properties.IsomorphicLaxMonoidalFunctor`
[46] Agda formalisation: `Theory.Monad.Relative`

Definition 7.2.8 (*Relative monad*)
Let $\mathbb{C}$ and $\mathbb{D}$ be categories. A *relative monad* consists of

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

that satisfy the following laws

- for all $a, b \in \mathsf{Obj}_{\mathbb{C}}$, $k \in \mathsf{Hom}_{\mathbb{D}}(J(a), T(b))$ the *right unit* law
$$k = k^* \circ_{\mathbb{D}} \eta_a,$$
- for all $a \in \mathsf{Obj}_{\mathbb{C}}$ the *left unit* law
$$\eta_a^* = \mathsf{id}_{\mathbb{D}} \in \mathsf{Hom}_{\mathbb{D}}(T(a), T(a)) \quad \text{and}$$
- for all $a, b, c \in \mathsf{Obj}_{\mathbb{C}}$, $k \in \mathsf{Hom}_{\mathbb{D}}(J(a), T(b))$, $l \in \mathsf{Hom}_{\mathbb{D}}(J(b), T(c))$ the *associativity* law
$$(l^* \circ_{\mathbb{D}} k)^* = l^* \circ_{\mathbb{D}} k^*.$$

Note that, although the full definition only requires a object mapping $T : \mathsf{Obj}_{\mathbb{C}} \rightarrow \mathsf{Obj}_{\mathbb{D}}$ we can define a canonical functor based on T through the following morphism mapping
$$(f : a \rightarrow b) \mapsto ((\eta_b \circ_{\mathbb{D}} J(f))^* : T(a) \rightarrow T(b))$$

7.2.8 Categorical representation of constraints

We model constraints categorically through concrete categories:

Definition 7.2.9 (*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 7.2.10).

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

Definition 7.2.10 (*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 `T a` these have to be modelled in the source category $\mathbb{O}$ of the underlying functors. Thus `T` becomes `T` $: \mathbb{O} \longrightarrow \mathsf{Set}$. We require $\mathbb{O}$ to be concrete so that we can recover the underlying unconstrained type `a` in Set through the associated functor J.

We exemplify our approach to model constraints by applying it to the definition of a constrained functor for the type `Set`. As explained in Section 3.4, `Set` provides a representation of finite mathematical 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 tree.

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

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

Another difference of our formalisation is that we require 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 our formalisation would have the following form instead:

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

We argue that this would be the morally correct way of implementing `Set` in Haskell, because practically we can only construct empty or singleton sets 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 so.

In our formalisation the constrained source category $\mathbb{O}$ has the dependent pairs of sets (or types) and their instances of `Ord` as objects. The morphisms are total functions between these sets.

$$\begin{aligned} \mathsf{Obj}_{\mathbb{O}_{\texttt{Ord}}} &\overset{\text{def}}{=} \{\, (a, \texttt{Ord}\ a) \mid \forall\, a \in \mathsf{Obj}_{\mathsf{Set}} \text{ that have an instance of } \texttt{Ord} \,\} \\ \mathsf{Hom}_{\mathbb{O}_{\texttt{Ord}}}(a,b) &\overset{\text{def}}{=} \mathsf{Hom}_{\mathsf{Set}}(\mathsf{proj}_1(a), \mathsf{proj}_1(b)) \end{aligned}$$

The identity morphism and composition from Set will also be used. 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. Our formalisation of `Set` indeed forms a categorical functor from $\mathbb{O}_{\texttt{Ord}}$ to Set[47] as we would expect. 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 we presented above.

To create a `Functor` instance that exactly matches the `map` function provided in the Haskell implementation, we could use a different constraint category where the morphisms are dependent pairs of the form $(f : a \to b,\ \texttt{Ord}\ b)$ and the objects are simply sets.

We can see that this technique can be applied to any kind of constraint that can be captured by the `FunctorCts` associated constraints of our constrained variant of the `Functor` class. We only need to be careful that if different constraints apply to `a` and `b` respectively that they need to abide the category laws and allow for proper composition.

[47] Agda proof: `Theory.Haskell.Constrained.Examples.SetFunctor`

Monadic notion	Standard	Graded $\forall i \in M$.	Indexed $\forall i, j \in I$.	Constrained
Underlying functors	$T : \mathbb{C} \longrightarrow \mathbb{C}$	$T_i : \mathbb{C} \longrightarrow \mathbb{C}$	$T_{i,j} : \mathbb{C} \longrightarrow \mathbb{C}$	$T : \mathbb{O} \longrightarrow \mathbb{C}$
Basic model	Monad	—	—	Relative monad
Lax mon. functor	$\begin{array}{rcl} \mathbf{1} & \longrightarrow & [\mathbb{C},\mathbb{C}]_\circ \\ \bullet & \mapsto & T \\ \mathsf{id}_\bullet & \mapsto & \mathsf{id}_T \end{array}$	$\begin{array}{rcl} \mathsf{Mon}_M & \longrightarrow & [\mathbb{C},\mathbb{C}]_\circ \\ i & \mapsto & T_i \\ \mathsf{id}_i & \mapsto & \mathsf{id}_{T_i} \end{array}$	—	– ? –
Lax 2-functor	$\begin{array}{rcl} \mathbf{1} & \longrightarrow & \mathsf{Cat} \\ \bullet & \mapsto & \mathbb{C} \\ \bullet & \mapsto & T \\ \mathsf{id}_\bullet & \mapsto & \mathsf{id}_T \end{array}$	$\begin{array}{rcl} \mathsf{Mon}^2_M & \longrightarrow & \mathsf{Cat} \\ \bullet & \mapsto & \mathbb{C} \\ i & \mapsto & T_i \\ \mathsf{id}_i & \mapsto & \mathsf{id}_{T_i} \end{array}$	$\begin{array}{rcl} \mathbb{I} & \longrightarrow & \mathsf{Cat} \\ i & \mapsto & \mathbb{C} \\ (i,j) & \mapsto & T_{i,j} \\ \mathsf{id}_{(i,j)} & \mapsto & \mathsf{id}_{T_{i,j}} \end{array}$	– ? –

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 1. Overview of categorical formalisations for monadic notions.

7.3 Monadic notions

We will now discuss how the different monadic notions can be modelled categorically. Table 1 provides an overview and guide of the categorical models for the different monadic notions and highlights their common structure. Note that our formalisation of Haskell functors in Set and Set-based functors in category theory are equivalent[48] to each other. Graded and indexed monads have a whole family of underlying functors instead of just one.

7.3.1 Standard monads

In Set standard monads as they are encoded in Haskell are equivalent to categorical monads[49] from Set to Set. A categorical monad[50] is a endofunctor $T : \mathsf{Set} \longrightarrow \mathsf{Set}$ with a natural transformation $\eta : 1 \stackrel{.}{\longrightarrow} T$ (return) and $\mu : T \circ T \stackrel{.}{\longrightarrow} T$ (join).

Definition 7.3.1 (*Monad*)
Let $\mathbb{C}$ be a category. A *monad* consists of

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

[48] Agda proof: `Theory.Functor.Properties.IsomorphicHaskellFunctor`
[49] Agda proof: `Theory.Monad.Properties.IsomorphicHaskellMonad`
[50] Agda formalisation: `Theory.Monad.Definition`

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

$$\begin{tikzcd}
F(F(F(c))) \arrow[r, "F(\mu_c)"] \arrow[d, "\mu_{F(c)}"'] & F(F(c)) \arrow[d, "\mu_c"] \\
F(F(c)) \arrow[r, "\mu_c"'] & F(c)
\end{tikzcd}
\qquad
\begin{tikzcd}
F(c) \arrow[r, "F(\eta_c)"] \arrow[d, "\eta_{F(c)}"'] \arrow[dr, equal] & F(F(c)) \arrow[d, "\mu_c"] \\
F(F(c)) \arrow[r, "\mu_c"'] & F(c)
\end{tikzcd}$$

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

This exactly matches the building blocks of a lax monoidal functor and indeed categorical monads in a category $\mathbb{C}$ are in one-to-one correspondence[51] with lax monoidal functors from the unit category $\mathbf{1}$ to $[\mathbb{C},\mathbb{C}]_\circ$. The object-level mapping selects the functor T in $[\mathbb{C},\mathbb{C}]_\circ$ and the morphism-level mapping maps to the identity natural transformation on T. The η and $\mu_{a,b}$ transformations of the lax monoidal functor are in one-to-one correspondence with the η and μ transformations of the monad.

7.3.2 *Graded monads*

Graded monads (Section 3.3) do not have a standard categorical model we can refer to, but we can generalise[52] the categorical definition of standard monads from Definition 7.3.1 in a straight-forward way.

Definition 7.3.2 (*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 $\eta : \mathsf{id}_{\mathbb{C}} \stackrel{\cdot}{\longrightarrow} F_e$ and
- a family of natural transformations $\mu^{i,j} : F_i \circ F_j \stackrel{\cdot}{\longrightarrow} F_{i \diamond j}$

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

$$\begin{tikzcd}
F_i(F_j(F_k(c))) \arrow[r, "F_i\left(\mu_c^{j,k}\right)"] \arrow[d, "\mu^{i,j}_{F_k(c)}"'] & F_i\left(F_{j\diamond k}(c)\right) \arrow[d, "\mu_c^{i,j\diamond k}"] \\
F_{i\diamond j}(F_k(c)) \arrow[r, "\mu_c^{i\diamond j,k}"'] & F_{i\diamond j\diamond k}(c)
\end{tikzcd}
\qquad
\begin{tikzcd}
F_i(c) \arrow[r, "F_i(\eta_c)"] \arrow[d, "\eta_{F_i(c)}"'] \arrow[dr, equal] & F_i(F_e(c)) \arrow[d, "\mu_c^{i,e}"] \\
F_e(F_i(c)) \arrow[r, "\mu_c^{e,i}"'] & F_i(c)
\end{tikzcd}$$

for all $i, j, k \in M$ and $c \in \mathsf{Obj}_{\mathbb{C}}$.

This generalisation is equivalent[53] to the Set-based (and Haskell inspired) definition of graded monads[54]. Just as with standard monads, there is a one-to-one correspondence[55] with lax monoidal functors (Katsumata, 2014; Gaboardi *et al.*, 2016). The unit category $\mathbf{1}$ is exchanged with the monoidal category Mon_M based on the monoid of effects M

[51] Agda proof: `Theory.Functor.Monoidal.Properties.IsomorphicMonad`
[52] Agda formalisation: `Theory.Haskell.Parameterized.Graded.Monad`
[53] Agda proof: `Theory.Haskell.Parameterized.Graded.Monad.Properties.IsomorphicHaskellGradedMonad`
[54] Agda formalisation: `Haskell.Parameterized.Graded.Monad`
[55] Agda proof: `Theory.Functor.Monoidal.Properties.IsomorphicGradedMonad`

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 object-level each element of M is mapped to the functor with that element as index and the identity morphisms is mapped to the identity transformation on the corresponding functor.

7.3.3 Indexed monads

Indexed monads (Section 3.2) do not have a standard categorical model either, but as with graded monads we can generalise[56] Definition 7.3.1 to arrive at a suitable definition.

Definition 7.3.3 (*Parametrised monad*)
Let $\mathbb{C}$ and $\mathbb{I}$ be a categories. $\mathbb{I}$ is the category of indices. A *parametrised 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 $\eta^i : \mathsf{id}_\mathbb{C} \overset{\cdot}{\longrightarrow} F_{\mathsf{id}_i}$ for all $i \in \mathsf{Obj}_\mathbb{I}$, and
- a family of natural transformations $\mu^{f,g} : F_g \circ F_f \overset{\cdot}{\longrightarrow} F_{g \circ_\mathbb{I} f}$ 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

$$\begin{tikzcd}
F_h\big(F_g\big(F_f(c)\big)\big) \arrow[r, "F_h\left(\mu_c^{f,g}\right)"] \arrow[d, "\mu^{g,h}_{F_f(c)}"'] & F_h\big(F_{g\circ_\mathbb{I} f}(c)\big) \arrow[d, "\mu_c^{g\circ_\mathbb{I} f,h}"] \\
F_{h\circ_\mathbb{I} g}\big(F_f(c)\big) \arrow[r, "\mu_c^{f,h\circ_\mathbb{I} g}"'] & F_{h\circ_\mathbb{I} g\circ_\mathbb{I} f}(c)
\end{tikzcd}
\qquad
\begin{tikzcd}
F_f(c) \arrow[r, "F_f(\eta_c^i)"] \arrow[d, "\eta^j_{F_f(c)}"'] \arrow[dr, equal] & F_f(F_{\mathsf{id}_i}(c)) \arrow[d, "\mu_c^{\mathsf{id}_i,f}"] \\
F_{\mathsf{id}_j}\big(F_f(c)\big) \arrow[r, "\mu_c^{f,\mathsf{id}_i}"'] & F_f(c)
\end{tikzcd}$$

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 $h \in \mathsf{Hom}_\mathbb{I}(k,l)$.

Note, that instead of using a simple set to provide the indices we are using a category where the morphisms provide the indices. Also notice that this definition has a structure very similar to that of a graded monad. Indeed, if we use Mon_M (for any monoid M) as the category of indices both definitions are equivalent[57] to each other. We call this new definition a parametrised monad instead of an indexed monad, because it captures standard and graded monads, as well as indexed monads.

As before this definition is equivalent[58] to the Set-based (and Haskell inspired) definition[59] of indexed monads. Unfortunately, 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. We are required to look at the more general structure of a lax 2-functor to find a matching categorical notion. The category of possible indices 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

[56] Agda formalisation: `Theory.Haskell.Parameterized.Indexed.Monad`
[57] Agda proof: `Theory.Haskell.Parameterized.Indexed.Monad.Properties.IsomorphicGradedMonad`
[58] Agda proof: `Theory.Haskell.Parameterized.Indexed.Monad.Properties.IsomorphicHaskellIndexedMonad`
[59] Agda formalisation: `Haskell.Parameterized.Indexed.Monad`

parametrised monad maps 0-cells to the underlying category $\mathbb{C}$ of our parametrised 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 our parametrised monad[60].

We are aware that Atkey (2009) gave a categorical model for indexed monads, but we have noticed that it does not exactly fit what we expect of an indexed monad. Atkeys work is discussed in Section 10.1.

7.3.4 Unified categorical representation of parametrised 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[61] to a lax 2-functors from Mon^2_M to Cat where the 0-cell mapping is constant to $\mathbb{C}$. Since the lax monoidal functors of standard[62] and graded[63] monads fulfil these conditions they are in one-to-one correspondence with the lax 2-functors of the corresponding structure.

Due to this correspondence we have found 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 by generalizing the categorical definition of monads to graded and indexed monads we have found 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 we deem them useful, especially for functional programmers that have limited familiarity with category theory.

7.3.5 Constrained monads

We can use relative monads to model constrained monads in the same way as described for functors in Section 7.2.8. Given a category of constraints $\mathbb{O}$ that is concrete on $\mathbb{C}$ we can use the associated functor $J_{\mathbb{O}}$ as the functor J from Definition 7.3.4. Thus, we can define a constrained monad as a relative monad from $\mathbb{O}$ to the underlying category $\mathbb{C}$. For a given object mapping T we then need to define η_a and the Kleisli extension $-^*$.

In our `Set` example we let underlying category be Set. We can then use $\mathbb{O}_{\texttt{Ord}}$ and $J_{\mathbb{O}_{\texttt{Ord}}}$ again. This results in the following morphism for η_a

$$\begin{gathered}\forall\,(a,\ \texttt{Ord}\ a)\in\mathsf{Obj}_{\mathbb{O}_{\texttt{Ord}}}.\ J_{\mathbb{O}_{\texttt{Ord}}}(a,\ \texttt{Ord}\ a)\rightarrow T(a,\ \texttt{Ord}\ a)\\ =\\ \forall\,(a,\ \texttt{Ord}\ a)\in\mathsf{Obj}_{\mathbb{O}_{\texttt{Ord}}}.\ a\rightarrow T(a,\ \texttt{Ord}\ a)\end{gathered}$$

[60] Agda proof: `Theory.TwoFunctor.Properties.IsomorphicIndexedMonad`
[61] Agda proof: `Theory.TwoFunctor.Properties.IsomorphicLaxMonoidalFunctor`
[62] Agda proof: `Theory.TwoFunctor.Properties.IsomorphicMonad`
[63] Agda proof: `Theory.TwoFunctor.Properties.IsomorphicGradedMonad`

and the following mapping for the Kleisli extension

$$\begin{aligned}
&\forall\,(a,\ \texttt{Ord}\,a),(b,\ \texttt{Ord}\,b)\in\mathsf{Obj}_{\mathbb{O}_{\texttt{Ord}}}.\\
&\quad(J_{\mathbb{O}_{\texttt{Ord}}}(a,\ \texttt{Ord}\,a)\rightarrow T(b,\ \texttt{Ord}\,b))\ \mapsto\ (T(a,\ \texttt{Ord}\,a)\rightarrow T(b,\ \texttt{Ord}\,b))\\
&\qquad\qquad=\\
&\forall\,(a,\ \texttt{Ord}\,a),(b,\ \texttt{Ord}\,b)\in\mathsf{Obj}_{\mathbb{O}_{\texttt{Ord}}}.\\
&\quad(a\rightarrow T(b,\ \texttt{Ord}\,b))\ \mapsto\ (T(a,\ \texttt{Ord}\,a)\rightarrow T(b,\ \texttt{Ord}\,b)).
\end{aligned}$$

Which reflects a constrained return and bind operation. Our formalisation of `Set` forms a relative monad in this way[64].

Note that in our Haskell representation of constrained monads we have 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, we have not yet found a categorical structure that captures both lax 2-functors and relative monads. As pointed out by Altenkirch (2010) there is no obvious way to form a join operation, i.e., a transformation $T \circ T \overset{.}{\longrightarrow} T$, as would be required by the lax 2-functors that are equivalent to the parametrised notions. Altenkirchs paper does show that a 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. In future work we may explore if the left Kan extension exists in our model of constrained monads.

Even though we could not find a standard categorical notion that captures lax 2-functors and relative monads, we can apply the same technique of generalisation that we used to define parameterised monads to relative monads. This leads to the notion of a parameterised relative monad.

Definition 7.3.4 (*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 $J : \mathbb{C} \longrightarrow \mathbb{D}$,
- a object mapping $T_f : \mathsf{Obj}_{\mathbb{C}} \rightarrow \mathsf{Obj}_{\mathbb{D}}$ 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 $\eta^i_a \in \mathsf{Hom}_{\mathbb{D}}(J(a), T_{\mathsf{id}_i}(a))$ 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 $k \in \mathsf{Hom}_{\mathbb{D}}\big(J(a), T_f(b)\big)$ a morphism $k^*_{f,g} \in \mathsf{Hom}_{\mathbb{D}}\big(T_g(a), T_{g\circ_{\mathbb{I}} f}(b)\big)$

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 $k \in \mathsf{Hom}_{\mathbb{D}}\big(J(a), T_f(b)\big)$ the *right unit* law
$$k = k^*_{f,\mathsf{id}_j} \circ_{\mathbb{D}} \eta^j_a,$$

[64] Agda proof: `Theory.Haskell.Constrained.Examples.SetMonad`

Applicative notion	Standard	Graded $\forall i \in M$.	Indexed $\forall i, j$.	Constrained
Underlying functors	$T : \mathbb{C} \longrightarrow \mathbb{C}$	$T_i : \mathbb{C} \longrightarrow \mathbb{C}$	$T_{i,j} : \mathbb{C} \longrightarrow \mathbb{C}$	$T : \mathbb{O} \longrightarrow \mathbb{C}$
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*)	$\mathbf{1} \longrightarrow [\mathbb{C},\mathbb{C}]_{\text{Day}}$ $\bullet \mapsto T$ $\mathsf{id}_\bullet \mapsto \mathsf{id}_T$	$\mathsf{Mon}_M \longrightarrow [\mathbb{C},\mathbb{C}]_{\text{Day}}$ $i \mapsto T_i$ $\mathsf{id}_i \mapsto \mathsf{id}_{T_i}$	—	$\mathbf{1} \longrightarrow [\mathbb{O},\mathbb{C}]_{\text{Day}}$ $\bullet \mapsto T$ $\mathsf{id}_\bullet \mapsto \mathsf{id}_T$

M – 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 2. Overview of categorical formalisations for applicative notions.

- 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

$$\left(\eta_a^i\right)^*_{\mathsf{id}_i,f} = \mathsf{id}_{T_f(a)} \in \mathsf{Hom}_\mathbb{D}\left(T_f(a), T_f(a)\right) \quad \text{and}$$

- for all $a,b,c \in \mathsf{Obj}_\mathbb{C}$, $s,t,u,v \in \mathsf{Obj}_\mathbb{I}$, $f \in \mathsf{Hom}_\mathbb{I}(s,t)$, $g \in \mathsf{Hom}_\mathbb{I}(t,u)$, $h \in \mathsf{Hom}_\mathbb{I}(u,v)$, $k \in \mathsf{Hom}_\mathbb{D}(J(a), T_g(b))$ and $l \in \mathsf{Hom}_\mathbb{D}\left(J(b), T_f(c)\right)$ 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 a parameterised relative monad is fully equivalent[65] to that of a parameterised monad if $\mathbb{C}$ and $\mathbb{D}$ are the same. If $\mathbb{I}$ is the unit category they are equivalent[66] to relative monads. Due to their relationship with parameterised monads they are equivalent[67] to certain lax 2-functors if the $\mathbb{C}$ and $\mathbb{D}$ are the same and the 0-cell mapping of the lax 2-functor is constant. Hence, we have developed a custom categorical notion that is able to provide a model for all of the monadic notions we set out to model (given their Set-based representation).

Unpublished work by Orchard and Mycroft (2012) discusses using relative monads as a categorical model for constrained monads in a similar manner to our work.

7.4 Applicative notions

As can be seen in Table 2, all generalisations of applicatives are based on the same underlying functors as the previously presented monadic notions.

[65] Agda proof: `Theory.Haskell.Parameterized.Relative.Monad.Properties.IsomorphicIndexedMonad`
[66] Agda proof: `Theory.Haskell.Parameterized.Relative.Monad.Properties.IsomorphicRelativeMonad`
[67] Agda proof: `Theory.TwoFunctor.Properties.IsomorphicParameterizedRelativeMonad`

7.4.1 Standard applicatives

Standard applicatives in Set are equivalent[68] to lax monoidal functors (McBride & Paterson, 2008). 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

$$\eta : \begin{cases} \top \rightarrow F(\top) \\ \texttt{()} \mapsto \texttt{pure ()} \end{cases}$$

and the natural transformation $\mu_{a,b}$ is given by

$$\mu_{a,b} : \begin{cases} F(a) \times F(b) \rightarrow F(a \times b) \\ (\texttt{u},\texttt{v}) \mapsto \texttt{fmap (\textbackslash x y -> (x,y)) u <*> v}. \end{cases}$$

In contrast to monads, applicatives do not require their underlying functors to be endofunctors. This is key to model graded and constrained applicatives.

7.4.2 Graded applicatives

To represent a graded applicative (Section 4.3) we only have to "add in" the monoid $(M, \bullet, e)$ of effects that it is parametrised over. This can be achieved by using a product of the monoidal category 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

$$\eta : (e, \top) \rightarrow F(e, \top)$$
$$\mu_{a,b} : F(i,a) \times F(j,b) \overset{\cdot}{\longrightarrow} F(i \bullet j, a \times b)$$

to form a graded applicative and is indeed equivalent[69] 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) (μa,b (ff , fa))
```

We can generalise the definition of a lax monoidal functor in a similar manner as we did for graded monads:

Definition 7.4.1 (*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

- a family of functors $F_i : \mathbb{C} \longrightarrow \mathbb{C}$ for all $i \in M$,
- a morphism $\eta : 1_\mathbb{D} \rightarrow F_e(1_\mathbb{C})$, and

[68] Agda proof: `Theory.Functor.Monoidal.Properties.IsomorphicHaskellApplicative`
[69] Agda proof: `Theory.Functor.Monoidal.Properties.IsomorphicGradedApplicative`

- a family of natural transformations $\mu^{i,j}_{a,b} : F_i(a) \otimes_{\mathbb{D}} F_j(b) \stackrel{\cdot}{\longrightarrow} F_{i \diamond j}(a \otimes_{\mathbb{C}} b)$ for all $i, j \in M$ and $a, b \in \mathsf{Obj}_{\mathbb{C}}$,

such that the diagram for *associativity*

$$\begin{tikzcd}
(F_i(a) \otimes_{\mathbb{D}} F_j(b)) \otimes_{\mathbb{D}} F_k(c) \arrow[r, "\alpha^{\mathbb{D}}_{F_i(a),F_j(b),F_k(c)}"] \arrow[d, "\mu^{i,j}_{a,b} \otimes_{\mathbb{D}} \mathsf{id}_{F_k(c)}"'] & F_i(a) \otimes_{\mathbb{D}} (F_j(b) \otimes_{\mathbb{D}} F_k(c)) \arrow[d, "\mathsf{id}_{F_i(a)} \otimes_{\mathbb{D}} \mu^{j,k}_{b,c}"] \\
F_{i \diamond j}(a \otimes_{\mathbb{C}} b) \otimes_{\mathbb{D}} F_k(c) \arrow[d, "\mu^{i \diamond j,k}_{a \otimes_{\mathbb{C}} b,c}"'] & F_i(a) \otimes_{\mathbb{D}} F_{j \diamond k}(b \otimes_{\mathbb{C}} c) \arrow[d, "\mu^{i,j \diamond k}_{a,b \otimes_{\mathbb{C}} c}"] \\
F_{(i \diamond j) \diamond k}((a \otimes_{\mathbb{C}} b) \otimes_{\mathbb{C}} c) \arrow[r, "F_{i \diamond j \diamond k}\left(\alpha^{\mathbb{C}}_{a,b,c}\right)"'] & F_{i \diamond (j \diamond k)}(a \otimes_{\mathbb{C}} (b \otimes_{\mathbb{C}} c))
\end{tikzcd}$$

and the diagrams for *left* and *right unitality*

$$\begin{tikzcd}
1_{\mathbb{D}} \otimes_{\mathbb{D}} F_i(a) \arrow[r, "\eta \otimes_{\mathbb{D}} \mathsf{id}_a"] \arrow[d, "\lambda^{\mathbb{D}}_{F_i(a)}"'] & F_e(1_{\mathbb{C}}) \otimes_{\mathbb{D}} F_i(a) \arrow[d, "\mu^{e,i}_{1_{\mathbb{C}},a}"] \\
F_i(a) & F_{e \diamond i}(1_{\mathbb{C}} \otimes_{\mathbb{C}} a) \arrow[l, "i \overset{\mathsf{def}}{=} e \diamond i", "F_i\left(\lambda^{\mathbb{C}}_a\right)"']
\end{tikzcd}
\qquad
\begin{tikzcd}
F_i(a) \otimes_{\mathbb{D}} 1_{\mathbb{D}} \arrow[r, "\mathsf{id}_a \otimes_{\mathbb{D}} \eta"] \arrow[d, "\rho^{\mathbb{D}}_{F_i(a)}"'] & F_i(a) \otimes_{\mathbb{D}} F_e(1_{\mathbb{C}}) \arrow[d, "\mu^{i,e}_{a,1_{\mathbb{C}}}"] \\
F_i(a) & F_{i \diamond e}(a \otimes_{\mathbb{C}} 1_{\mathbb{C}}) \arrow[l, "i \overset{\mathsf{def}}{=} i \diamond e", "F_i\left(\rho^{\mathbb{C}}_a\right)"']
\end{tikzcd}$$

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[70] with any lax monoidal functor of the form $\mathsf{Mon}_M \times \mathsf{Set} \longrightarrow \mathsf{Set}$.

7.4.3 Constrained applicatives

In case of a constrained applicative (Section 4.4) we use a concrete category that models the constraints as source for the lax monoidal functor. This works analogous to the modelling of constrained functors and monads described in previous sections. The difference is that our 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. We will exemplify the implications of this with $\mathbb{O}_{\texttt{Ord}}$. Firstly, a tensor product has to exist:

$$\otimes : \begin{cases} \mathbb{O}_{\texttt{Ord}} \times \mathbb{O}_{\texttt{Ord}} \longrightarrow \mathbb{O}_{\texttt{Ord}} \\ ((a,\ \texttt{Ord}\ \ a),\ (b,\ \texttt{Ord}\ \ b)) \mapsto (a \times b,\ \texttt{Ord}\ \ (a \times b)) \\ (f : a \to b,\ g : c \to d) \mapsto ((\lambda (x,\ y).\ (f(x),\ g(y))) : a \times c \to b \times d) \end{cases}$$

Secondly, a tensor unit is required, i.e., $(\top,\ \texttt{Ord}\ \ \top)$ is required to be in $\mathsf{Obj}_{\mathbb{O}_{\texttt{Ord}}}$. Finally, we need to make sure that the associator, left unitor and right unitor can be defined and obey triangle and pentagon identity. Our formalisation of `Set` forms a lax monoidal functor from $\mathbb{O}_{\texttt{Ord}}$ to Set as expected[71].

[70] Agda proof: `Theory.Haskell.Parameterized.Graded.LaxMonoidalFunctor.Properties.IsomorphicLaxMonoidalFunctor`

[71] Agda proof: `Theory.Haskell.Constrained.Examples.SetApplicative`

Given the lax monoidal functor that represents a standard, graded or constrained applicative we *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 them a lax monoidal structure very similar to that of the monadic notions.

We 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.

7.4.4 Indexed applicatives

Unfortunately, just as their monadic counterpart, indexed applicatives (Section 4.2) do not match the structure of a lax monoidal functor. We suspect that there is a more abstract structure similar to that of a lax 2-functor that will capture all of the applicative notions we have discussed.

As a first approximation of this more general structure we can generalise the definition of a lax monoidal functor in the same manner as we did for parametrised monads:

Definition 7.4.2 (*Parametrised 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 *parametrised lax monoidal functor* between $\mathbb{C}$ and $\mathbb{D}$ consists of

- a family of functors $F_f : \mathbb{C} \longrightarrow \mathbb{C}$ for every $f \in \mathsf{Hom}_\mathbb{I}(i, j)$,
- a morphism $\eta : 1_\mathbb{D} \rightarrow F_{\mathsf{id}_i}(1_\mathbb{C})$ for every $i \in \mathsf{Obj}_\mathbb{I}$, and
- a family of natural transformations $\mu^{f,g}_{a,b} : F_f(a) \otimes_\mathbb{D} F_g(b) \overset{\cdot}{\longrightarrow} F_{g \circ_\mathbb{I} f}(a \otimes_\mathbb{C} b)$ 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*

$$\begin{tikzcd}
(F_f(a) \otimes_\mathbb{D} F_g(b)) \otimes_\mathbb{D} F_h(c) \arrow[r, "\alpha^\mathbb{D}_{F_f(a),F_g(b),F_k(c)}"] \arrow[d, "\mu^{f,g}_{a,b} \otimes_\mathbb{D} \mathsf{id}_{F_k(c)}"'] & F_i(a) \otimes_\mathbb{D} (F_j(b) \otimes_\mathbb{D} F_k(c)) \arrow[d, "\mathsf{id}_{F_i(a)} \otimes_\mathbb{D} \mu^{g,h}_{b,c}"] \\
F_{g \circ f}(a \otimes_\mathbb{C} b) \otimes_\mathbb{D} F_k(c) \arrow[d, "\mu^{g \circ f,h}_{a \otimes_\mathbb{C} b,c}"'] & F_f(a) \otimes_\mathbb{D} F_{h \circ g}(b \otimes_\mathbb{C} c) \arrow[d, "\mu^{f,h \circ g}_{a,b \otimes_\mathbb{C} c}"] \\
F_{h \circ (g \circ f)}((a \otimes_\mathbb{C} b) \otimes_\mathbb{C} c) \arrow[r, "F_{h \circ g \circ f}\left(\alpha^\mathbb{C}_{a,b,c}\right)"'] & F_{(h \circ g) \circ f}(a \otimes_\mathbb{C} (b \otimes_\mathbb{C} c))
\end{tikzcd}$$

and the diagrams for *left* and *right unitality*

$$\begin{tikzcd}
1_\mathbb{D} \otimes_\mathbb{D} F_f(a) \arrow[r, "\eta \otimes_\mathbb{D} \mathsf{id}_a"] \arrow[d, "\lambda^\mathbb{D}_{F_f(a)}"'] & F_{\mathsf{id}_i}(1_\mathbb{C}) \otimes_\mathbb{D} F_f(a) \arrow[d, "\mu^{\mathsf{id}_i,f}_{1_\mathbb{C},a}"] & F_f(a) \otimes_\mathbb{D} 1_\mathbb{D} \arrow[r, "\mathsf{id}_a \otimes_\mathbb{D} \eta"] \arrow[d, "\rho^\mathbb{D}_{F_f(a)}"'] & F_f(a) \otimes_\mathbb{D} F_{\mathsf{id}_j}(1_\mathbb{C}) \arrow[d, "\mu^{f,\mathsf{id}_j}_{a,1_\mathbb{C}}"] \\
F_f(a) & F_{f \circ \mathsf{id}_i}(1_\mathbb{C} \otimes_\mathbb{C} a) \arrow[l, "f \overset{\mathsf{def}}{=} f \circ \mathsf{id}_i", "F_f(\lambda^\mathbb{C}_a)"'] & F_f(a) & F_{\mathsf{id}_j \circ f}(a \otimes_\mathbb{C} 1_\mathbb{C}) \arrow[l, "f \overset{\mathsf{def}}{=} \mathsf{id}_j \circ f", "F_f(\rho^\mathbb{C}_a)"']
\end{tikzcd}$$

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

ZU064-05-FPR paper 12 December 2017 15:21

This structure is equivalent[72] to the Set-based definition of indexed applicatives if we use a codiscrete category of indices. It is also equivalent[73] to standard lax monoidal functors if $\mathbb{I}$ is the unit category and it is equivalent[74] to our definition of graded lax monoidal functors if $\mathbb{I}$ is the category Mon_M formed by a monoid M. Thus, our definition of parameterised lax monoidal functors captures all of the different applicative notions that we presented.

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.

7.5 Conclusion of categorical models

In conclusion, we have made a considerable step towards a unified categorical model for different generalisations of monads and applicatives.

We have proposed categorical notions that give a unified model for all of the monadic and applicative notions respectively. Our definition of parameterised relative monads captures all of the monadic notions and the parameterised lax monoidal functors capture all of the applicative notions.

As a result of this work we can present a hierarchy of monadic and applicative notions. Figure 7 contains the hierarchy of models for the monadic notions and Figure 8 contains the hierarchy of models for the applicative notions. The arrow relationship in these diagrams states the following relationship:

$$(A \to B) \text{ iff } \big(\exists\, B'.\, (A \cong B') \land (B' \subseteq B)\big)$$

The $\cong$-symbol is supposed to denote a one-to-one correspondence between A and B'. For example, the arrow between parameterised monads and lax 2-functors states that a subset of all possible lax 2-functors is in one-to-one correspondence with all possible parameterised monads. Hence, lax 2-functors subsume parameterised monads.

Even though we are still missing an overarching pre-existing categorical model for either notion (as indicated by the question marks in the diagrams), our work contributes a considerable step towards a unified theory of these notions.

Firstly, we have given an overview of the different notions and put them into context with each other. We are not aware of any previous work that has provided a detailed categorical context in this form.

Secondly, we have formalised all of the involved monadic, applicative and categorical notions in the proof assistant Agda. Based on this formalisation we have given hard evidence of the discussed relationships between the different notions.

[72] Agda proof: `Theory.Haskell.Parameterized.Indexed.LaxMonoidalFunctor.Properties.IsomorphicHaskellIndexedApplicative`
[73] Agda proof: `Theory.Haskell.Parameterized.Indexed.LaxMonoidalFunctor.Properties.IsomorphicLaxMonoidalFunctor`
[74] Agda proof: `Theory.Haskell.Parameterized.Indexed.LaxMonoidalFunctor.Properties.IsomorphicGradedLaxMonoidalFunctor`

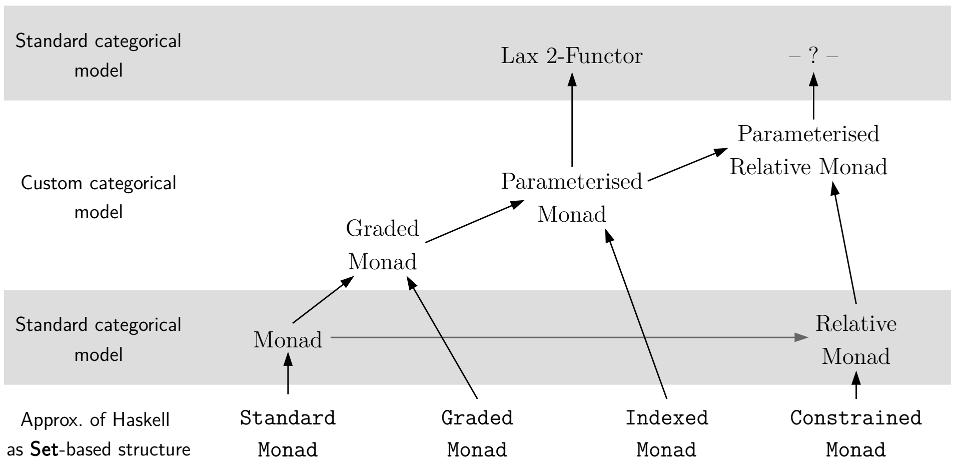

Fig. 7. Hierarchy of categorical models for monadic notions.

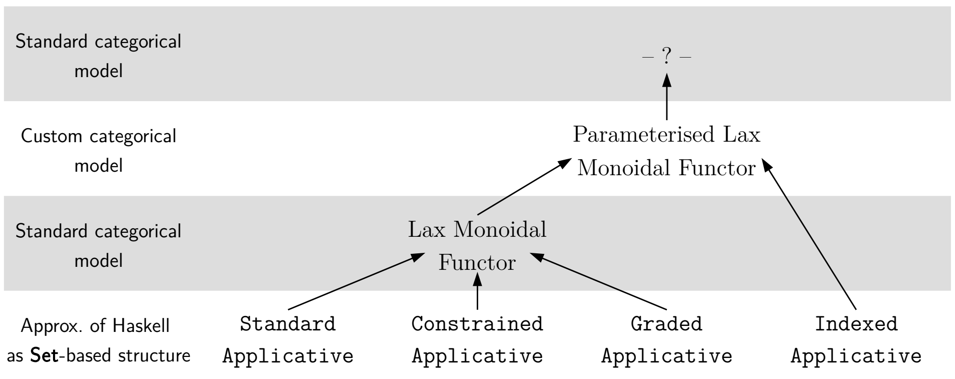

Fig. 8. Hierarchy of categorical models for applicative notions.

8 Implementation and use of the GHC plugin

As explained in Section 5.2, 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. In this section we explain how our 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 or 4 that is encoded with supermonads as if they were written with the corresponding specialised type class.

Before we discuss the details of the plugin we give a brief overview of the GHC type checker plugin mechanism. At the time of writing, the supermonad plugin has been tested using GHC 7.10.3, 8.0.1 and 8.2.1. It will definitely not work with versions of GHC lower than 7.10, because the plugin infrastructure was still under development prior to that version.

8.1 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 (Diatchki, 2015) and units of measure (Gundry, 2015). We have previously used a plugin to integrate polymonads into Haskell (Bracker & Nilsson, 2015). We will content ourselves with a brief explanation here, referring the reader to the earlier work and Section 11.3.4 of the GHC user's guide[75] for details.

GHC type checks code in program fragments, e.g., top-level function definitions. For each fragment, type checking and inference produce 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. The constraint solver solves wanted constraints iteratively. If the constraint solver is not able to solve a constraint or make progress, it will ask available plugins for help. The plugin can then process the constraints and either provide evidence to be used for them or create new constraints to guide the constraint solver.

8.2 Supermonad Plugin

We start the tour of the plugin by recapitulating the type inference capabilities we aim to support:

- A connection between the bind, ap, and return operation for each supermonad or applicative.
- Enforcing and using the knowledge that all three type constructors in the head of `Bind` or `Applicative` instances are partial applications of the same base constructor.
- Inference of the indices through unification with the bind or return operations type signature of specific instances.

When talking about the algorithm, we have to distinguish cases based on the base constructors that are found in the supermonad constraints. Therefore, we refer to base constructors that are not type variables as *manifest constructors*, base constructors that are ambiguous type variables as *ambiguous constructors* and base constructors that are unambiguous type variables as *variable constructors*.

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

Note that the plugin cannot enforce that the `Bind`, `Applicative` and `Return` instances actually form valid supermonads or applicatives according to our categorical model. This especially means that the plugin does not check or enforce that the equational laws associated with each notion are satisfied. The user still has 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. It is assumed that a supermonad or superapplicative in Haskell consists of exactly one `Bind` (or `Applicative`) and one `Return` instance. This assumption is true for all of the monadic and applicative notions we aim to support.

Since it is not possible to enforce this assumption directly in Haskell, the plugin checks that there is only one `Bind` (or `Applicative`) and `Return` instance and that their arguments are applications of the same base constructor. If all instances conform, 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.

We also make the assumption that a monadic and applicative computation only ever involves a single supermonad or applicative. For examples, 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 out of the following steps:

1. Construct a graph that connects two wanted supermonad or applicative constraints if and only if they share an ambiguous constructor. We call each connected component of the graph a *constraint group*.
2. For each constraint group, solve the ambiguous constructors:
   - If the group only involves 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 only 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. We choose to group them by overlapping ambiguous constructors. These constructors can only overlap between two constraints if they are actually used

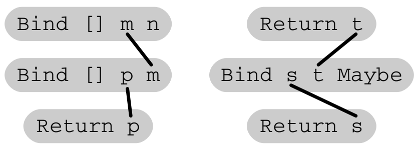

Fig. 9. Graph produced by the separation algorithm from the example.

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 9 shows the graph produced by the separation step. We can see that 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.

Explanation of Step 2. If there is just one manifest and no variable constructor within the constraint group, we can equalise all ambiguous constructors with it, because it designates the supermonad or applicative this group is working with. Our example from Step 1 demonstrates this. In Figure 9 the ambiguous constructors `m`, `n`, and `p` will be equalised with manifest constructor `[]`, whereas `s` and `t` will be 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 applicatives within the same computation. Therefore, we need to abort in this case.

If there are manifest and variable constructors involved with the constraint group we also need 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 applicative 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. We 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, we are required to use several variable constructors to express the function's type. In this case all of the given `Bind` and `Return` constraints form the supermonad we are working with.

To solve the ambiguous constructors we need to check all of the associations between ambiguous and variable constructors and see which associations are satisfiable by the given constraints. If there is only one possible association, we know that is the one intended by the programmer. If there are several possible associations we need to abort, because the function's type is ambiguous and committing to one of them may result in unintended runtime behaviour.

We can illustrate this process with `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 the same.

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 of `forever`, i.e., $\{\, \mathtt{s} \mapsto \mathtt{n} \,\}$, the plugin will use the second association to solve the ambiguous constructor and ignore the first association.

The runtime of checking all associations is exponential in the number of ambiguous and variable constructors. However, our 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.

We can construct examples with multiple possible associations by adding constraints that are not necessary to solve the ambiguous constructors. However, in practice we have not encountered a polymorphic function with several satisfiable associations. We suspect this is due to the fact that we only provide the minimal amount of constraints necessary to type the polymorphic functions we wrote.

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 may 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 our 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, we know that the graded monad `Vector` is used. Thus, we 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 we cannot unify 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 9) give confidence that the plugin restores GHC's ability to infer the type of supermonad and applicative computations.

8.3 Using the Plugin

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

- To use do-notation for the monadic notions the GHC language extension `RebindableSyntax` needs to be enabled. This allows using the bind and return operation provided by the supermonad library instead of the standard monad versions.
- 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 the appropriate counterparts. By default, the standard prelude is not imported when rebindable syntax is enabled.
- Activate the type checker plugin by inserting the following line at the top of the module:

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

- Finally, the user has to implement instances of the `Bind` (and/or `Applicative`) and `Return` class for all of their supermonads. Most of the standard library and monad transformer instances are provided by our library.

An example of a module that performs the first three steps can be seen in Figure 10. The supermonad library repository[76] contains several examples that demonstrate how supermonads can be used.

[76] 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
```

Fig. 10. Example of a module header that enables the use of supermonads and applicatives.

A collection of supermonad and superapplicative related functions that are not in the standard prelude can be found in the module `Control.Super.Monad.Functions`.

To work with the constrained variant of supermonads and applicatives use the alternative prelude `Control.Super.Monad.Constrained.Prelude`. Non-prelude functions of the constrained variant can be found in `Control.Super.Monad.Constrained.Functions`.

9 Case studies

We pursued a practically driven approach to develop supermonads, applicatives and the associated plugin. To provide evidence of the practicality of our approach, and to check that everything works as intended, we carried out a couple of case studies. The case studies also constitute a stress test of our plugin on a larger code base. The source code of these case studies is available in the supermonad library repository[77].

9.1 Teaching Compiler

We chose to apply supermonads and applicatives to a teaching compiler for our first case study. The compiler is made up of 25 modules containing more than 3800 lines of code (not counting blank lines and comments). A majority 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 applicatives.

To adapt the compiler to use supermonads, we applied the first three steps of Section 8.3 to each module and provided instances of the `Bind` and `Return` classes for each of the custom monads defined in the compiler. To exemplify this, we will look at a monad transformer that adds the handling of failures:

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

Originally the monad instance had the following form:

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

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

```
        Just a  -> unDFT (f a) )
```

Without changing the implementation we can translate this 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) )
```

We also generalised the instance at the same time. This allows arbitrary supermonads to be wrapped in `DFT`, because we use the constraint `Bind m n p` instead of `Bind m m m`.

In addition, we had to modify functions and classes that are polymorphic in their monad. We had to replace their `Monad m` constraints with `Bind m m m` and `Return m` constraints and add the associated bind constraints `BindCts m m m` to every function involving a bind operation.

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

```
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 mechanical process we described in Section 5.6.

```
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 we could only apply the naive conversion, because the classes were written having standard monads specifically in mind. Generalizing them to apply to generalised monads would require a careful redesign of their use of base constructors and their class structure. Depending on how general the adjusted classes are, it might be necessary to list the required constraints of each class function in an individual associated constraint as demonstrated in Section 5.6.

As can be seen, porting code from standard to supermonads only involved adjusting for the `Applicative`, `Bind` and `Return` class and activating the plugin. Type inference was not affected by the change to supermonads and the adjustments to utilise the supermonad classes were reasonably mechanical.

9.2 Chat Server and Client

For our second case study, we wanted an example involving generalised as well as standard monads. Unfortunately, the only examples we found using generalised monads did no longer compile. Therefore, we decided to implement our own application: a chat server. It uses session types as presented by Pucella and Tov (Pucella & Tov, 2008) in their `simple-sessions` library[78]. As the library does not support network communication, our example uses communication between 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). A majority of that code uses the do-notation to express computations involving the standard monads `IO` and `STM` in addition to the generalised `Session` monad.

We 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 generalised `Session` monad were not named differently from the standard operations, the amount of annotation 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 `Session` indexed monad.

The refactoring to use supermonads only required the changes we expected:

- 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 we could not use the `where` notation to add annotations. Therefore, we had to use local `let` bindings, which cluttered the code:

```
do
  -- ...
  run ( let (>>=) = (Prelude.>>=)
```

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

```
            (>>) = (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, we need to disambiguate for every monadic computation.

After refactoring to use supermonads, we could remove these local `let` bindings:

```
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%). 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, we can see how supermonads ease the use of different monadic notions in the same application and enable the reuse of code.

10 Related Work

10.1 Atkey's parameterised notions of computation

Atkey (2009) suggested a way of modelling indexed monads categorically. He used 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 operation to model them.

His approach is applicable to our formalisation in Set, but there is one major difference between Atkey's and our model. The indexed monads that we draw from Haskell have an associated functor $F_{i,j} : \mathbb{C} \longrightarrow \mathbb{C}$ for all indices $i, j \in \mathsf{Obj}_{\mathbb{I}}$. The homomorphism mapping of this functor has the following form:

$$\mathsf{Hom}_{\mathbb{C}}(a,b) \to \mathsf{Hom}_{\mathbb{C}}(F_{i,j}(a), F_{i,j}(b))$$

Notice, that i and j are invariant under the functor $F_{i,j}$. The indices of any parametrised monad in Haskell are never assumed to be functorial. Therefore, T is a more general model of indexed monads in Haskell, because in T the indices are functorial. If the index category $\mathbb{I}$ of Atkeys parameterised monad is discrete and the index category of our indexed monad is its codiscrete counterpart they are in one-to-one correspondence[79] with each other.

[79] Agda proof: `Theory.Haskell.Parameterized.Indexed.Monad.Properties.IsomorphicAtkeyParameterizedMonad`

10.2 Comparison to Kmett's approach

The basic idea of the generalised encoding of bind operations that we use was explored by Kmett[80] in 2007. As we explained in Section 5.3, Kmett's work included a functional dependency on the `Bind` class and a specialised return operation. Both were introduced to aid type inference. We applied Kmett's approach to our first case study (Section 9.1). This revealed many cases where manual type annotations and a correct choice of the return operation were necessary to resolve ambiguous types. Both of these tasks are tedious.

In contrast, our plugin restores type inference. Therefore, including the functional dependency in our encoding does not restrict the `Bind` or `Applicative` class in a useful manner. That said, Kmett's approach is more flexible than supermonads are as there is no requirement for a single base constructor. This allows the encoding of implicit lifting within the bind and ap operations. 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 effectively has been integrated into the bind operation. This leads to the question of why supermonads and applicatives do not allow these lifting instances?

Implicit lifting can 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 we have a chain of several bind operations where the first computation uses the `Maybe` monad and the last computation uses the list monad, when do we lift into the list monad? 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 can also have an impact on the performance and the runtime behavior of the resulting program. For example, if we provide a bind operation from `STM` (software transactional memory) (Harris *et al.*, 2005) to `IO`, 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, and could even lead to deadlocks or other undesirable behavior. What if the lifting decides the instance of a class that will be used? In that case the lifting can, again, influence the runtime behavior.

There are no obviously correct answers to these questions. Hence, we decided to not allow lifting bind operations, but to require users of supermonads to express lifting from one notion to another explicitly.

Our categorical model also supports the decision to only allow single base constructor instances as it does not allow implicit lifting and mixing different monads or applicatives together.

However, even if there were no concerns about the semantics of implicit lifting, we still have to disallow it, because our solving algorithm is based on the assumption that all `Bind` instance arguments are partial applications of the same base constructor.

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

Kmett did not present any laws or a theory for his approach, though we assume he intended a generalised version of the standard monads laws just like those our categorical models provide.

10.3 Comparison to polymonads

Polymonads (Hicks *et al.*, 2014) are similar to supermonads in that they also use a set of bind operations that allow a different type constructor in each position and that they also have a set of unary type constructors. The encoding of polymonads is also very similar to that of supermonads. In our previous work (Bracker & Nilsson, 2015) we implemented a plugin for GHC that added type inference for polymonads to the compiler.

Though supermonads and polymonads may seem similar at first glance, especially when looking at their representation in Haskell, there are several differences:

- Polymonads do not have specific return operations. They encode their return operations through a bind operation with the identity monad in the first two positions.
- There is not necessarily a common base constructor for a given polymonad.
- All polymonads also have to contain a distinguished type constructor that acts like the identity monad.
- 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.

The laws of polymonads are more complex than the laws of our categorical model and do not as obviously relate to the standard monad laws. Though it can be shown that the that the standard monad laws can be derived[81] 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 to have principal polymonads for solving they only support *phantom* indices as arguments to their partially applied type constructors. This is a major disadvantage compared to supermonads, because non-phantom indices allow for many interesting examples and applications of supermonads and superapplicatives.

The polymonad theory also does not offer 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.

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 we have discussed, including the implicit lifting bind operations we mentioned in the comparison with Kmett's approach.

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

10.4 Other approaches to generalisation

There are other generalisations of functors, applicatives and monads.

In unpublished work, McBride (McBride, 2011) presents a generalisation of monads different from any of the generalisations we have discussed so far. In his generalisation he proposes a bind and return operation with the following type signature:

$$(\texttt{>>=}) :: \mathsf{m}\ \alpha\ i \rightarrow (\forall\ j.\ \alpha\ j \rightarrow \mathsf{m}\ \beta\ j) \rightarrow \mathsf{m}\ \beta\ i$$
$$\texttt{return} :: \alpha\ i \rightarrow \mathsf{m}\ \alpha\ i$$

He exemplifies the use cases of his generalisation by using it to encode indexed monads and statically typing the open or closed state of a file handle. Both examples can be modelled using the range of monadic notions that are supported by supermonads and we are not aware of any other use cases for his generalisation that could not be expressed using supermonads. In addition, it is also not obvious how his generalisation can be used within the do-notation except by encoding indexed monads.

Another generalisation is given in the package `rank2classes`[82]. In this package the `Functor` and `Applicative` classes have been generalised to support functors on the category of endofunctors on Hask and natural transformations between them. Thus leading to definitions similar to the following:

```
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
```

This generalisation cannot be incorporated by our encoding. We could allow it by adding additional associated type synonyms to specify the type of arrow with which we are working. Exploring this design space, and if there is a corresponding monadic notion, remains future work.

We also found that invertible syntax descriptions (Rendel & Ostermann, 2010) require a generalised form of functor that uses partial isomorphisms instead of normal functions as mapping arrow:

```
newtype Iso a b = Iso (a -> Maybe b) (b -> Maybe a)

class IsoFunctor f where
  fmap :: Iso a b -> f a -> f b
```

As in the previous example this requires a way to define a custom type for the involved mapping function.

[82] Hackage: *rank2classes* - `http://hackage.haskell.org/package/rank2classes`

10.5 Other related work

There is work on a categorical generalisation of applicatives, monads and arrows by Rivas and Jaskelioff (Rivas & Jaskelioff, 2017). They exhibit the deeper connections between the three notions and unify them as monoids in monoidal categories. However, it is unclear how generalisations of the aforementioned notions relate to their work.

In work preceding their work on polymonads Swamy et al. (Swamy *et al.*, 2011) presented a way to automatically insert bind and return operations into pure functional programs. Their work provides a way of writing implicitly monadic programs and also covers the integration of morphisms between different monads. However, their work does not solve the problems we described during our discussion of implicit lifting in the context of Kmett's approach.

Jones (1995) suggested a possible alternative to the GHC type checker plugins. His work on custom improvements describes a system to aid constraint solving by associating patterns of constraints containing open type variables with equations involving those type variables. Stuckey and Sulzmann (Stuckey & Sulzmann, 2005) developed a theory of constraint handling rules that applies custom improvements to functional languages. They also developed a prototype language with constraint handling rules called Chameleon (Stuckey *et al.*, 2004). Unfortunately, their implementation is not available publicly anymore and there is no implementation of constraint handling rules for GHC. Therefore, to our knowledge, GHC plugins are the most practical way of implementing supermonads.

The "rmonad" (restricted monads) package[83] provides an alternative encoding of constrained monads and functors. 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 in our encoding 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 their constrained monads are less cluttered with constraints. Our encoding has the advantage that it is more flexible as it allows for constraints that can distinguish between `a` and `b`.

11 Conclusion

We have presented a variety of different generalised monadic notions that are already in use in many programs today. Based on these notions we developed corresponding generalised notions of applicatives that, to our knowledge, have not been explored before.

Our unified representation of the generalised monadic notions in Haskell, called supermonad, has been extended to also support the new applicative notions as superapplicatives. Alongside the encoding we have also extended the language extension, provided as a GHC plugin, to add support for superapplicatives. This enables programmers to use and

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

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.

Due to the practical implications of supporting constraints on result types, we offer a separate prelude that allows programmers to choose whether they want to deal with these implications or not. However, future work may provide a way to handle constraints on result types in a more pleasant manner.

The exploration of categorical models for the different generalised notions has shown that there are abstract structures that connect many of the presented notions. Namely, we found that 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, we proposed the custom definitions of *parameterised relative monads* and *parameterised lax monoidal functors* to capture all of the monadic and applicative notions. Although we did not find a pre-existing categorical notion that subsumes our proposed definitions, we have made great progress towards a categorical model that connects all of the different notions discussed. The term supermonad and superapplicative should only refer to the Haskell encoding, since the categorical models we found already have suitable names. Although we propose categorical notions as the semantics of supermonads and applicatives, the plugin cannot enforce their laws and proper implementation, just as GHC cannot ensure that instances of the standard `Monad` class are correct.

In future work we will try to apply our technique to generalisation of arrows (Joseph, 2014; Nilsson & Nielsen, 2014). As the supermonad approach is relatively uncomplicated in many ways, and the generalisation of applicatives to superapplicatives along those lines proved to be straightforward, we are optimistic that this is feasible. Another line of work involves generalisation of other notions from Haskell's standard library, such as `MonadPlus`, `Alternative` and `Traversable`.

Acknowledgements

We want to thank Sandra Dylus, Paolo Capriotti and Martin Handley for their useful feedback on the preliminary version of this article.

References

Adams, Stephen. (1993). Functional pearls efficient sets — a balancing act. *Journal of functional programming*, **3**, 553–561.

Altenkirch, Thorsten, Chapman, James, & Uustalu, Tarmo. (2010). Monads need not be endofunctors. *Pages 297–311 of:* Ong, Luke (ed), *Foundations of Software Science and Computational Structures*. Lecture Notes in Computer Science, vol. 6014. Springer Berlin Heidelberg.

Atkey, Robert. (2009). Parameterised notions of computation. *Journal of functional programming*, **19**, 335–376.

Bénabou, Jean. (1963). Categories avec multiplication. *C. R. Acad. Sci., Paris*, **256**, 1887–1890.

Bénabou, Jean. (1965). Categories rélatives. *C. R. Acad. Sci., Paris*, **260**, 3824–3827.

Bénabou, Jean. (1967). *Introduction to bicategories*. Berlin, Heidelberg: Springer Berlin Heidelberg. Pages 1–77.

Bracker, Jan, & Gill, Andy. (2014). *Practical aspects of declarative languages: 16th international symposium, padl 2014, san diego, ca, usa, january 20-21, 2014. proceedings*. Cham: Springer International Publishing. Chap. Sunroof: A Monadic DSL for Generating JavaScript, pages 65–80.

Bracker, Jan, & Nilsson, Henrik. (2015). Polymonad programming in Haskell. *Pages 3:1–3:12 of: Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages*. IFL '15. New York, NY, USA: ACM.

Bracker, Jan, & Nilsson, Henrik. (2016). Supermonads: One notion to bind them all. *Pages 158–169 of: Proceedings of the 9th International Symposium on Haskell*. Haskell 2016. New York, NY, USA: ACM.

Chakravarty, Manuel M. T., Keller, Gabriele, & Jones, Simon Peyton. (2005). Associated type synonyms. *Pages 241–253 of: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*. ICFP '05. New York, NY, USA: ACM.

Devriese, Dominique, & Piessens, Frank. (2011). Information flow enforcement in monadic libraries. *Pages 59–72 of: Proceedings of the 7th ACM SIGPLAN Workshop on Types in Language Design and Implementation*. TLDI '11. New York, NY, USA: ACM.

Diatchki, Iavor S. (2015). Improving Haskell types with SMT. *Pages 1–10 of: Proceedings of the 8th ACM SIGPLAN Symposium on Haskell*. Haskell 2015. New York, NY, USA: ACM.

Dijkstra, Edsger W. (n.d.). *Een algorithme ter voorkoming van de dodelijke omarming*. circulated privately.

Gaboardi, Marco, Katsumata, Shin-ya, Orchard, Dominic, Breuvart, Flavien, & Uustalu, Tarmo. (2016). Combining effects and coeffects via grading. *Pages 476–489 of: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ICFP 2016. New York, NY, USA: ACM.

Giorgidze, George, Grust, Torsten, Schreiber, Tom, & Weijers, Jeroen. (2011). *Haskell boards the ferry*. Berlin, Heidelberg: Springer Berlin Heidelberg. Pages 1–18.

Gundry, Adam. (2015). A typechecker plugin for units of measure: Domain-specific constraint solving in GHC Haskell. *Pages 11–22 of: Proceedings of the 8th ACM SIGPLAN Symposium on Haskell*. Haskell 2015. New York, NY, USA: ACM.

Harris, Tim, Marlow, Simon, Peyton-Jones, Simon, & Herlihy, Maurice. (2005). Composable memory transactions. *Pages 48–60 of: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '05. New York, NY, USA: ACM.

Hicks, Michael, Bierman, Gavin, Guts, Nataliya, Leijen, Daan, & Swamy, Nikhil. (2014). Polymonadic programming. *Electronic proceedings in theoretical computer science*, **153**, 79–99.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the acm*, **12**(10), 576–580.

Hughes, John. (1999). Restricted data types in Haskell. *Haskell Workshop*, vol. 99.

Hughes, John. (2000). Generalising monads to arrows. *Science of computer programming*, **37**(1–3), 67 – 111.

Joseph, Adam Megacz. (2014). *Generalized arrows*. Ph.D. thesis, EECS Department, University of California, Berkeley.

Katsumata, Shin-ya. (2014). Parametric effect monads and semantics of effect systems. *Pages 633–645 of: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. New York, NY, USA: ACM.

Kelly, G. M., & Street, Ross. (1974). *Review of the elements of 2-categories*. Berlin, Heidelberg: Springer Berlin Heidelberg. Pages 75–103.

Marlow, Simon, Brandy, Louis, Coens, Jonathan, & Purdy, Jon. (2014). There is no fork: An abstraction for efficient, concurrent, and concise data access. *Pages 325–337 of: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP '14. New York, NY, USA: ACM.

Marlow, Simon, Peyton Jones, Simon, Kmett, Edward, & Mokhov, Andrey. (2016). Desugaring Haskell's do-notation into applicative operations. *Pages 92–104 of: Proceedings of the 9th International Symposium on Haskell*. Haskell 2016. New York, NY, USA: ACM.

McBride, Conor. (2011). *Functional pearl: Kleisli arrows of outrageous fortune*. Unpublished.

McBride, Conor, & Paterson, Ross. (2008). Applicative programming with effects. *Journal of functional programming*, **18**, 1–13.

Moggi, Eugenio. (1988). *Computational lambda-calculus and monads*. IEEE Computer Society Press.

Moggi, Eugenio. (1991). Notions of computation and monads. *Information and computation*, **93**(1), 55 – 92. Selections from 1989 IEEE Symposium on Logic in Computer Science.

Nilsson, Henrik, & Nielsen, Thomas A. (2014). Declarative modelling for bayesian inference by shallow embedding. *Pages 39–42 of: Proceedings of the 6th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. EOOLT '14. New York, NY, USA: ACM.

Norell, Ulf. (2007). *Towards a practical programming language based on dependent type theory*. Ph.D. thesis, Chalmers University of Technology.

Orchard, Dominic, & Petricek, Tomas. (2014). Embedding effect systems in Haskell. *Pages 13–24 of: Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*. Haskell '14. New York, NY, USA: ACM.

Paterson, Ross. (2001). A new notation for arrows. *Pages 229–240 of: Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*. ICFP '01. New York, NY, USA: ACM.

Persson, Anders, Axelsson, Emil, & Svenningsson, Josef. (2012). *Generic monadic constructs for embedded languages*. Berlin, Heidelberg: Springer Berlin Heidelberg. Pages 85–99.

Pucella, Riccardo, & Tov, Jesse A. (2008). Haskell session types with (almost) no class. *Pages 25–36 of: Proceedings of the First ACM SIGPLAN Symposium on Haskell*. Haskell '08. New York, NY, USA: ACM.

Rendel, Tillmann, & Ostermann, Klaus. (2010). Invertible syntax descriptions: Unifying parsing and pretty printing. *Pages 1–12 of: Proceedings of the Third ACM Haskell Symposium on Haskell*. Haskell '10. New York, NY, USA: ACM.

Rivas, Exequiel, & Jaskelioff, Mauro. (2017). Notions of computation as monoids. *Journal of functional programming*, **27**(Oct.).

Sculthorpe, Neil, Bracker, Jan, Giorgidze, George, & Gill, Andy. (2013). The constrained-monad problem. *Pages 287–298 of: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. New York, NY, USA: ACM.

Stuckey, Peter J., & Sulzmann, Martin. (2005). A theory of overloading. *Acm transactions on programming languages and systems*, **27**(6), 1216–1269.

Stuckey, Peter J., Sulzmann, Martin, & Wazny, Jeremy. (2004). The chameleon system. *Pages 13–32 of:* Frühwirth, Thom, & Meister, Marc (eds), *First Workshop on Constraint Handling Rules: Selected papers*. University of Ulm.

Swamy, Nikhil, Guts, Nataliya, Leijen, Daan, & Hicks, Michael. (2011). *Lightweight monadic programming in ML*. Tech. rept. MSR-TR-2011-39. Microsoft Research.

Swierstra, S. Doaitse, & Duponcheel, Luc. (1996). *Deterministic, error-correcting combinator parsers*. Berlin, Heidelberg: Springer Berlin Heidelberg. Pages 184–207.

Vizzotto, Juliana, Altenkirch, Thorsten, & Sabry, Amr. (2006). Structuring quantum effects: superoperators as arrows. *Mathematical structures in computer science*, **16**(June), 453–468.

Wadler, Philip. (1992). The essence of functional programming. *Pages 1–14 of: Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '92. New York, NY, USA: ACM.

Wadler, Philip. (1994). Monads and composable continuations. *Lisp and symbolic computation*, **7**(1), 39–55.

Wadler, Philip. (1998). The marriage of effects and monads. *Pages 63–74 of: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*. ICFP '98. New York, NY, USA: ACM.