Supermonads

One Notion to Bind Them All

Jan Bracker Henrik Nilsson

Functional Programming Lab
School of Computer Science
University of Nottingham, UK
{jzb,nhn}@cs.nott.ac.uk

Abstract

Several popular generalizations of monads have been implemented in Haskell. Unfortunately, because the shape of the associated type constructors do not match the standard Haskell monad interface, each such implementation provides its own type class and versions of associated library functions. Furthermore, simultaneous use of different monadic notions can be cumbersome as it in general is necessary to be explicit about which notion is used where. In this paper we introduce *supermonads*: an encoding of monadic notions that captures several different generalizations along with a version of the standard library of monadic functions that work uniformly with all of them. As standard Haskell type inference does not work for supermonads due to their generality, our supermonad implementation is accompanied with a language extension, in the form of a plugin for the Glasgow Haskell Compiler (GHC), that allows type inference for supermonads, obviating the need for manual annotations.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) Languages; D.3.3 [*Programming Languages*]: Language Constructs and Features—Control structures, Constraints

Keywords functional programming, Glasgow Haskell Compiler, Haskell, monads, syntactic support, type checker plugin

1. Introduction

A number of different notions of computation are used to capture side-effects in Haskell, primarily applicative functors [27], monads [28, 29, 48], and arrows [19]. All of these notions are of such importance that Haskell or GHC provide dedicated syntactic support for them: the do-notation [1, 25] and arrow syntax [35].

While a wide range of effects are expressible using these notions, they, as currently embodied in Haskell, certainly do not cover all use cases. Consequently, a range of generalizations have been investigated, typically aiming to enhance the expressiveness by adding type-level indices or constraints. For monads these efforts have amongst others lead to advanced implementations of session-types [38], effect systems [26, 34], and information flow control [11]. Unfortunately, as realized, these generalizations do not interoperate smoothly, leading to a number of problems.

Firstly, code reuse is hampered, because the type constructors of the various notions have different arities and therefore they require different type classes. Hence, each notion also requires a separate implementation of standard library functions.

Secondly, if more than one notion is used within the same module, writing code using the standard monad syntax can become cumbersome as additional annotations may be required.

It is therefore of interest to find a notion that captures as many of the generalizations as possible in a uniform manner, along with an encoding that fits seamlessly with Haskell's existing monadic support. In earlier work [9], we successfully integrated polymonads [16] into Haskell to mitigate the above problems. While polymonads are very general in some ways, and have the benefit of being compositional, the polymonad theory in its current form does not allow non-phantom indices or constrained 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.

In this article, we explore a different, practically motivated approach addressing the above problems using a notion we call *supermonad*. Supermonads provide a unified representation covering a broad range of generalized monads, including all of those mentioned above, with the supermonad laws being straightforward generalizations of the standard monad laws. We choose the prefix "super", because it means above, over, and inclusive. It is used in the same manner as in "superclass". The notions of polymonad and supermonad share similarities, but their exact relationship remains to be explored.

In our work, we have so far focused on monads, but there are similar concerns with generalizations of other notions of computations such as arrows [21, 30]. Being relatively straightforward, we expect the approach presented here to be applicable without too much difficulty also to those other notions.

We make the following specific contributions:

- A survey of generalized monads (Section 2).
- Supermonads:
  - A Haskell representation of supermonads (Section 3).
  - A library providing the supermonad representation along with supermonad versions of monadic prelude functions.
  - A type checker plugin for GHC to help with type inference for supermonads (Section 5).
  - A formalization of supermonads in Agda, along with proofs that the surveyed generalized monads indeed are supermonads (Section 8).

Haskell'16, September 22-23, 2016, Nara, Japan
ACM. 978-1-4503-4434-0/16/09...
http://dx.doi.org/10.1145/2976002.2976012

- Two case studies that demonstrate that the approach works in practice and that the integration of the different monadic notions indeed is seamless (Section 6).

Adding result type constraints is largely orthogonal to the rest of the development, but has some practical implications. We therefore defer their introduction until section Section 7. Finally, related work is reviewed in Section 9, with a particular emphasis on polymonads.

2. Monadic Notions

We have examined the, to our knowledge, most popular monadic notions to find a notion that captures all of them.

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.

Standard Monads. The first form of monadic notion we looked at are standard monads [28] with their well-known bind and return operations:

$$\texttt{(>>=)} :: \mathsf{m}\ \alpha \to (\alpha \to \mathsf{m}\ \beta) \to \mathsf{m}\ \beta$$
$$\texttt{return} :: \alpha \to \mathsf{m}\ \alpha$$

Standard monads have been used to model side-effects [29, 48] such as state, exceptions, parsing, non-determinism, concurrency, continuations, as well as for embedded domain specific languages.

Hoare/Parameterized/Indexed Monads. Hoare or parameterized monads [4, 49] use a type constructor with two additional indices that express a pre- and postcondition for the computation. The bind and return operation essentially encode the composition and empty statement rules of Hoare logic [17]:

$$\texttt{(>>=)} :: \mathsf{m}\ i\ j\ \alpha \to (\alpha \to \mathsf{m}\ j\ k\ \beta) \to \mathsf{m}\ i\ k\ \beta$$
$$\texttt{return} :: \alpha \to \mathsf{m}\ i\ i\ \alpha$$

Practical applications of Hoare monads include session types [38, 46], typed state [4], and composable continuations [49]. Several packages on Hackage provide Hoare monads [13, 23, 37].

Effect Monads. Effect monads [22, 34, 50] use one additional index to account for the side-effects of their computation more precisely. The index contains elements of a monoid and the bind operation uses the monoid operation to combine the effects of the two computations it composes. Constraints on the indices are required to ensure the monoidal behavior of the indices:

$$\texttt{(>>=)} :: (\mathsf{Cts}\ i\ j) \Rightarrow \mathsf{m}\ i\ \alpha \to (\alpha \to \mathsf{m}\ j\ \beta) \to \mathsf{m}\ (i\ \diamond\ j)\ \beta$$
$$\texttt{return} :: \alpha \to \mathsf{m}\ \varepsilon\ \alpha$$

The return operation uses the neutral element of the monoid to represent the absence of side-effects.

Effect monads have been used to model information flow control [11], heterogeneous state [34], type-level counters and vectors [32]. Several packages on Hackage provide effect monads [23, 32].

To express the constraints on the indices and specify which monoid is being used in a specific instance, libraries usually use GHC's `TypeFamilies` extension that allows the declaration of of associated type synonyms for classes [10].

Constrained/Restricted Monads. The previous generalized notions added indices to the monadic type to describe side-effects more precisely. In contrast, constrained monads introduce constraints on the result types to allow structures that only form a monad if the result types meet a certain criteria.

Constrained monads have been discussed by Hughes [18] as restricted monads. One of Hughes's examples of such a monad is the set monad. The set elements need an ordering to make an efficient implementation of the set operations possible. Similarly to lists, sets constitute a monad, but cannot be made an instance of the standard monad class because this class does not allow constraints on the result type. Finite vectors [47] are another example of constrained monads, and there is a recurring need for constrained monads in the context of domain specific languages [8, 36].

A constrained monad has the following bind and return operations, where constraints are specialized to specific instances through associated type synonyms:

$$\texttt{(>>=)} :: (\mathsf{CtsB}\ \alpha\ \beta) \Rightarrow \mathsf{m}\ \alpha \to (\alpha \to \mathsf{m}\ \beta) \to \mathsf{m}\ \beta$$
$$\texttt{return} :: (\mathsf{CtsR}\ \alpha) \Rightarrow \alpha \to \mathsf{m}\ \alpha$$

Naturally, constrained monads necessitate the introduction of constrained functors and constrained applicative functors.

We know of one package that provides support for constrained monads [41]. In some cases, a deep embedding together with normalization can provide an alternative way to work with constrained monads without going beyond the standard monad class [40].

3. Supermonads

We would like 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 notion. Additionally, it should not be necessary to resort to manual disambiguation when working with more than one monadic notion at a time.

First we need to understand why each monadic notion covered in Section 2 requires a separate type class in Haskell. Their type classes follow this scheme:

```
class SomeMonadicNotion m where ...
```

Here, `m` is the type constructor that is used throughout the bind and return operation. What makes it impossible to give a single type class of this shape that captures all of the mentioned monadic notions is that the associated type constructor in each case has a different arity and thus a different kind; the above type class only allows a type constructor of one specific kind and arity.

A solution to this problem can be found in the work by Kmett [23, 24] and the work on polymonads [9, 16]. Both introduce a type class similar to the following:

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

This way, when defining an instance, `m`, `n` and `p` can be partially applied versions of the type constructor in the instance head.

This approach requires a separation of the bind and return function, because in the new `Bind` class it is unclear which of the three type constructors, `m`, `n` or `p`, should be used for the return operation.

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

Examples of how instances for different kinds of monadic notions can be given using this approach are presented in Section 4.

Insufficient Type Inference. Although these two type classes allow us to express all of the different monadic notions mentioned before, GHC's type inference does not suffice to resolve `Bind` and `Return` constraints in most situations.

In Haskell 2010 [1], type inference is guaranteed for almost all features of the language. However, to implement the `Bind` class we require GHC's language extension `MultiParamTypeClasses` [45, Section 9.8.1.1]. This allows type classes with more than one argument, but it also means there are more cases where types can no longer be inferred automatically. Because the `Bind` class has 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 a bind operation, some of the type constructors may become ambiguous variables as far as GHC can tell.

Additionally, the separation of the bind and return operation into two different classes often means that it is unclear which `Return` instance to use because the used bind operation no longer determines a corresponding return operation.

Let us illustrate the type inference problem through an example:

```
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. This is ambiguous because the inferred variable `m` does not occur on both sides of `=>`. Therefore, the compiler aborts with an error message: there is no unambiguous way of instantiating `m` without jeopardizing the runtime behavior of `plus3`.

Our case studies (Section 6) show that this issue is commonplace in monadic programs. In addition, they show that if our monadic notion has indices, e.g., monad transformers or effect monads, then we cannot infer the type of those indices anymore 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.

Enhancing Type Inference and Constraint Solving. To address the insufficient type inference capabilities for `Bind` and `Return` constraints Kmett [24] added a functional dependency and a specialized version of the return operation that always operated on the `Identity` monad.

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

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

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

The 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 to use his library [23] instead of our supermonad library. There were still many situations that required a manual type annotation to solve ambiguous variables. In addition to these manual annotations, we also had to choose the correct return operation (`return` or `returnM`) depending on the context. Both of these tasks are tedious in nature.

The functional dependency Kmett introduces is not powerful enough to restore type inference. What is actually required 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 `Bind` and `Return` class:

- We lost the connection between the bind and return operation, which was encoded through the single type class that contained both before.
- We also lost the knowledge that all three type constructors in the `Bind` class are partial applications of the same type constructor.
- Finally, we lost the ability to infer the indices through unification with the bind or return operations type signature, because this is only possible if we know which instance we are working with.

To our knowledge it is not possible to address these issues inside Haskell itself. We therefore introduce the unifying monadic notion of a *supermonad* as a *language extension*.

Supermonads in Haskell. We opt to introduce supermonads through their realization in Haskell, deferring a precise definition and formalization of the supermonad notion as such to Section 8. Until then, it suffices to be aware that the supermonad laws are the obvious generalizations of the standard monad laws.

We embody the notion of a supermonad through two type classes along the lines seen above. We then extend the type system by incorporating knowledge about supermonads. Concretely, this is realized by teaching GHC's type checker about the new monadic 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 computation that it would have been able to infer if that computation was using one of the specialized type classes from Section 2, thus restoring any type inference capabilities lost in the process of generalization.

Our two supermonad classes, `Bind` and `Return`, are 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) => Return m where
  type ReturnCts m :: Constraint
  type ReturnCts m = ()
  return :: (ReturnCts m) => a -> m a
```

The required language extensions are `TypeFamilies` [10], `ConstraintKinds` [5] and `MultiParamTypeClasses`.

Generalizing from standard monads, like Kmett, we also introduce `Functor` constraints on each of the partially applied type constructors. In our formalization (Section 8) we verified that for each of the notions we aim to support, the partially applied base constructors do form functors. We have chosen to not consider applicative functors here, leaving that for future work.

The associated type synonyms `BindCts` and `ReturnCts` are added to either class to allow for custom constraints on the *indices* of the type constructor. These constraints are especially important to support effect monads. We default `BindCts` and `ReturnCts` 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 an instance.

The above supermonad classes do not support constraints on the *result types* and thus cannot be instantiated for constrained monads. We will describe the integration of result type constraints in Section 7. Their integration is simple and does not require changes to the plugin, but there are some practical implications. Therefore, we discuss this separately.

Not all instantiations of the supermonad classes constitute valid supermonads, or suffice to support type inference. We thus require:

- Exactly one `Bind` and one `Return` instance per monadic base constructor.
- The constructors of a `Bind` instance are all partial applications of the same base constructor.

These contextual constraints are enforced by our plugin.

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 as such should not be confused with a particular realisation. There might be other ways to realise supermonads or some essentially equivalent notion: our investigation into categorical foundations might shed light on this (Section 10). What we have established is that there is at least one practical way of integrating a unified monadic notion 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 supermonad implementation and library is available as open source [7].

4. Examples of Supermonads

We now return to the monadic notions discussed in Section 2 and demonstrate how they all are instances of supermonads. In the following, the qualifier `P` is used to refer to the standard `Prelude` or any other module that provides a function with a clashing name.

Standard Monads. We exemplify with the `Maybe` monad:

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

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

Note how the supermonad was implemented directly in terms of the original monad above. Unfortunately, this does not always work: if we have a standard monad that is defined in terms of another monad, e.g., through 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 (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' )

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

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 some practical implication that we discuss in Section 6.

Hoare/Parameterized/Indexed Monads. For our example involving Hoare monads we chose to instantiate supermonad instances for the `Session` Hoare monad from the `simple-sessions` package [46]. The `Session` monad implements session types and uses the implementation of Hoare monads provided by the `indexed` package [37]. The bind and return operation provided by the `indexed` package are called `(>>>=)` and `ireturn`. We can give the supermonad instances for this notions by partially applying the `Session` type constructor in the instance head and reusing the existing implementation.

```
instance Bind (Session i j)
              (Session j k)
              (Session i k) where
  (>>=) = (>>>=)

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

Effect Monads. The implementation of effect monads we use for this example is provided by the `effect-monad` package [32]:

```
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 synonym `Unit` and `Plus` together with the kind variable `k` represent the monoid of the effect 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.

Let us illustrate with the `Counter` effect monad:

```
data Counter (n :: Nat) a = Counter { forget :: a }

instance Effect Counter where
  type Inv Counter n m = ()
  type Unit Counter = 0
  -- Type-level addition of naturals.
  type Plus Counter n m = n + m
  -- return :: a -> Counter 0 a
  return a = Counter a
  -- (>>=) :: Counter f a -> (a -> Counter g b)
  --       -> Counter (f + g) b
  (Counter a) >>= k = Counter ( forget ( k a ) )
```

The index is used to keep track of a type-level natural number of kind `Nat`. A special operation increments the index without any runtime effects. This can be used to ensure that operations are performed a certain number of times, for example.

```
instance (h ~ Plus Counter f g)
    => Bind (Counter (f :: Nat))
            (Counter (g :: Nat))
            (Counter (h :: Nat)) where
  type BindCts (Counter (f :: Nat))
               (Counter (g :: Nat))
               (Counter (h :: Nat)) = Inv Counter f g
  (>>=) = (P.>>=)

instance Return (Counter (0 :: Nat)) where
  return = P.return
```

Again, we can see that the original implementation of bind and return can be reused without alteration. Note that we cannot replace `h` with `Plus Counter f g` 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. We could have replaced `Plus`, `Unit` and `Inv` in the `Bind` and `Return` instance with their respective implementation, but we decided to use the more abstract definition as a guide to making any of the effect monads a supermonad.

5. Implementation of the Plugin

As explained in Section 3, by splitting the bind and return operations into different classes, and by allowing different type constructors to be used within the bind operation, the direct connection between the monad operations and the type constructors has been broken. This introduces ambiguities. Our GHC plugin resolves these ambiguities and aids type inference for the `Bind` and `Return` class by exploiting knowledge about supermonads and additional contextual constraints.

The goal of our plugin is to restore GHC's ability to infer any type that could have been inferred using one of the specialized classes before. To clarify: If GHC can infer the type of a given monadic computation using a specialized type class of the form presented in Section 2, then GHC is able to infer the type of the corresponding supermonad using our plugin.

This section explains how the plugin works and discusses the solving algorithm it applies. Before we discuss the details of the plugin we need to give a brief overview of the GHC type checker plugin mechanism. At the time of writing, the plugin has been tested using GHC 7.10.3 and GHC 8.0.1. It will not work with versions of GHC lower then 7.10, because the plugin infrastructure was still under development prior to that version.

5.1 GHC Type Checker Plugins

GHC supports a plugin interface to extend its constraint solver [45, Section 11.3.4]. 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 [12] and units of measure [14]. We have previously used a plugin to integrate polymonads into Haskell [9]. We will content ourselves with a brief explanation here, referring the reader to the earlier work and the GHC documentation 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.

5.2 Supermonad Plugin

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

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

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*.

Assumptions. It is assumed that that a supermonad in Haskell consists of exactly one `Bind` and one `Return` instance. This assumption is true for all of the monadic 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` 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` 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 computation only ever involves a single supermonad; e.g., it is not allowed to use several different supermonads within one do-block. This does not prohibit nesting of monadic computations; it just means that lifting monadic 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 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 variable constructors and no manifest constructors, we check all possible associations between the ambiguous and the variable constructors. 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 monadic 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 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
```

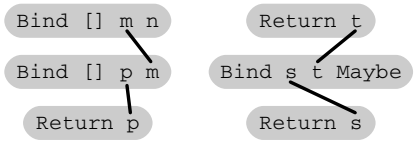

Figure 1. Graph produced by the separation algorithm from the example.

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 1 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.

Explanation of Step 2. If there is just one manifest and no variable constructor within the constraint group, we can equalize all ambiguous constructors with it, because it designates the supermonad this group is working with. Our example from Step 1 demonstrates this. In Figure 1 the ambiguous constructors `m`, `n`, and `p` will be equalized with manifest constructor `[]`, whereas `s` and `t` will be equalized with `Maybe`.

Finding more than one manifest constructor in a constraint group is nonsensical, because that would imply that several different supermonads are being used 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 manifest constructors already designate the supermonad that is used throughout the computation, which means there should not be any variable constructors.

If there are several variable and no manifest constructors in use, the constraint group originates from a function that is polymorphic in the supermonad being used. An example for this would be the `forever` function from the supermonad library:

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

To give this function in the most general form it needs to contain 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. 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 it is the one intended by the programmer and can proceed with it. If there are several possible associations we need to abort, because the function is ambiguous and dedicating to one of them may result in unintended runtime behavior.

We can illustrate this process with the `forever` function. GHC will infer the following constraints for `forever`:

```
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: $\{\texttt{s} \mapsto \texttt{m}\}$ and $\{\texttt{s} \mapsto \texttt{n}\}$. Since only one of the associations is satisfiable by the given constraints of `forever`, i.e., $\{\texttt{s} \mapsto \texttt{n}\}$, 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 implementing the standard library functions shows that this is not a problem in practice, because functions that are polymorphic in the used supermonad tend to be short and 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.

If a programmer were to write a function that ran into serious issues with the runtime of the search, she could easily mitigate that problem by giving additional type annotations throughout the function, thus reducing the number of ambiguous constructors.

Explanation of Step 3. The final step of solving indices through unification with the instance that is supposed to be used is motivated by our last observation. If this step is not done there may be leftover ambiguous type variables in the indices that prevent GHC from solving the constraint with an instance.

For example, if a `Return (Counter i)` constraint resulted from Step 2, we know that the `Counter` effect monad is used. Thus, we can lookup the `Return` instance of the `Counter` effect monad and solve `i` by unifying the instance arguments with the constraint arguments, which results in `i` being equalized with `0`.

That this process works is ensured by the assumption that there is exactly one `Bind` 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 instances exist for each base constructor.

In conclusion, the presented arguments and the conducted case studies (Section 6) give confidence that the plugin restores GHC's ability to infer the type of supermonad computations.

5.3 Using the Plugin

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

- Enable the GHC language extension `RebindableSyntax` [45, Section 9.3.15]. This extensions allows using do-notation with the bind and return operation provided by the supermonad library instead of the standard monad versions.
- Import `Control.Supermonad.Prelude`. This module provides all the functionality of the standard `Prelude`, except that the parts of the prelude relating to standard monads are replaced with counterparts for supermonads. The standard prelude is not imported by default when rebindable syntax is enabled.

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

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

- Finally, the user has to implement instances of the `Bind` and `Return` class for all of her supermonads.

An example of a module that performs the first three steps can be seen in Figure 2. The supermonad library repository [7] contains several examples that demonstrate how supermonads can be used.

6. Case Studies

We pursued a practically driven approach to develop supermonads and the associated plugin. To give evidence that our approach is viable in practice and works as intended, we have conducted case studies. The case studies also represent a stress test of our plugin on a larger code basis.

The source code of these case studies is available in the supermonad library repository [7].

6.1 Teaching Compiler

We chose to apply supermonads to a teaching compiler for our first case study. The compiler is made up of 25 modules containing more then 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.

To adapt the compiler to use supermonads, we applied the first three steps of Section 5.3 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
      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 generalized the instance at the same time. It now 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 mechanical process we described.

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

There was no need to change any of the instances.

Note that we did not generalize these classes and instances as we generalized the `Bind` instance, because they were written having standard monads specifically in mind. Generalizing them to apply to supermonads 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 `Bind` and `Return` constraints individually for each of the class functions, because the implementation of each individual instance may require different bind and return operations.

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

6.2 Chat Server and Client

For our second case study we wanted an example that mixes a generalized monad together with standard monads. Unfortunately, the only examples we found that used generalized monads were outdated, i.e., they did not compile anymore. Therefore, we decided to implement our own application: a chat server. It uses session types as presented by Pucella and Tov [38] in their `simple-sessions` library [46]. The library does not support network communication; instead, our example uses communication between different threads and 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 generalized `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 generalized `Session` monad. If the bind and return operations used by the generalized `Session` monad were not named differently from the standard operations the amount of annotation required would have been considerably higher, because then additional annotations would have been necessary for all of the monadic computations involving standard monads as well.

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.

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

module ExampleModule where

import Control.Supermonad.Prelude
```

Figure 2. Example of a module header that enables the use of supermonads.

- Implementation of supermonad instances for the generalized `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.>>=)
            (>>) = (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 generalized monads.

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

7. Integration of Result Type Constraints

The monadic notions presented in Section 2 also included constrained monads, which allow constraining the result type of our computation. We deferred the integration of constraints on the result types up until now, because of their practical implications.

Representation in Haskell. For the `Bind` and `Return` class to support constrained monads we need to give the result types of our operations as additional arguments to our associated constraints.

```
class (CFunctor m, CFunctor n, CFunctor 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 (CFunctor m) => Return m where
  type ReturnCts m (a :: *) :: Constraint
  type ReturnCts m a = ()
  return :: (ReturnCts m a) => a -> m a
```

We also need to replace the `Functor` constraints with `CFunctor` constraints. This is important, because constrained monads also require constrained functors. Therefore, we also introduce a replacement for the standard functor class.

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

A future implementation of the generalized `Applicative` type class will also require a separate constrained and an unconstrained version.

Practical Implications. Integrating constrained monads comes with some practical implications when writing programs. Evaluation of associated type synonyms is only possible if all arguments are known. This 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`, `ReturnCts` and `CFunctorCts` 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 constrained supermonads. The programmer needs to list `BindCts` constraints for all of possible result types of the given bind operation that occur in the function body. Especially for long polymorphic functions this can become onerous quickly depending on how many different result types there are within a function.

This problem is exacerbated for classes and instances that are polymorphic in the used supermonad. We revisit an example from our first case study to illustrate this point:

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

During the class definition it is unclear which bind and return operations will be involved in the implementation of the member functions; this depends on every individual instance. Thus, we need to allow custom individual constraints for every function in the class:

```
class (Applicative d, Bind d d d, Return d)
      => Diagnostic d where
  type EmitDCts d :: Constraint
  type OrCts d :: * -> Constraint
  -- ...
  emitD :: (EmitDCts d) => String -> d ()
  (|||) :: (OrCts d a) => d a -> d a -> d a
  -- ...
```

An instance might then look as follows:

```
instance (Applicative d, Bind d d d, Return d)
         => Diagnostic (DFT d) where
  type EmitDCts d =
      ( BindCts d d d String String
      , BindCts d d d String () {- ... -})
  type OrCts d a =
      ( BindCts d d d Int a
      , BindCts d d d a () {- ... -})
  -- ...
  emitD :: (EmitDCts d) => String -> d ()
  (|||) :: (OrCts d a) => d a -> d a -> d a
  -- ...
```

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.

This problem does not occur in code that is not polymorphic in the used supermonad: all arguments to `BindCts` and `ReturnCts` are known meaning they can be evaluated and the constraints they produce checked.

Due to this issue with polymorphic functions and classes our supermonad library offers two representations of supermonads; one that supports constraints on the result types and one that does not. This way a programmer can choose if she requires constraints on result types and wants to deal with the above issues. Supermonads with the possibility of constraints on the result types can be used by importing the module

```
Control.Supermonad.Constrained.Prelude
```

instead of the prelude mentioned in Section 5.3.

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, especially while they are new, outweighs this disadvantage.

Examples of Constrained Supermonads. As an example we implement the supermonad instances for the `Set` implementation of the standard library. This implementation of sets uses size-balanced binary trees [2] for efficiency. Therefore, many operations on sets require an ordering constraint (`Ord`). The module `Data.Set` that provides the implementation and functions is referred to as `S` in the following instances.

```
instance CFunctor Set where
  type CFunctorCts 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
```

We can see that both the functor and the `Bind` instance require an `Ord` constraint on `b`, but not on `a`. Note also that the `Return` instance does not require any constraints, because the `singleton` function works for any type.

It may seem that constraints for the `Return` class or `a` in general are superfluous. However, this is not the case. The vectors [40, 47] briefly mentioned in Section 2 require an equality constraint on the `Return` instance and they also require a constraint on `a` in the `Bind` instance. Another example where constraints in any of these locations are necessary would be embedded domain specific languages [8, 36].

8. Formalization of Supermonads in Agda

The different monadic notions presented in Section 2 all have laws associated with them that govern their behavior. However, in Haskell we cannot prove or enforce these laws. Therefore, we formalized the different monadic notions and their laws in the proof assistant Agda [31] to prove that the laws hold for supermonads and our examples.

The supermonad laws are based on the standard monad laws:
$\forall\, a : \alpha, m : \mathsf{M}\ \alpha, f : \alpha \to \mathsf{M}\ \beta, g : \beta \to \mathsf{M}\ \gamma.$

$$m \mathbin{>\!\!>\!=} \mathtt{return} \equiv m$$
$$(\mathtt{return}\ a) \mathbin{>\!\!>\!=} f \equiv f\ a$$
$$m \mathbin{>\!\!>\!=} (\lambda\, x.\ f\ x \mathbin{>\!\!>\!=} g) \equiv (m \mathbin{>\!\!>\!=} f) \mathbin{>\!\!>\!=} g$$

To capture all of the monadic notions that we aim to support, our formalization of supermonads generalizes the bind and return operation to support indices that may change over the course of a computation. We also introduce restrictions for the result types to support constrained monads. The resulting laws are a canonical generalization of the standard monad laws that accommodate for these additions.

Definition 1 (Supermonad)**.** Let $\mathsf{K}\backslash n$ with $n \geqslant 1$ be an n-ary type constructor and $\mathcal{I}$ be a set of arguments that K can be applied to. Define

$$\mathcal{M}_{\mathsf{K}} \stackrel{\text{def}}{=} \{\ \mathsf{K}\ a_1\ \ldots\ a_{n-1} \mid a_1, \ldots, a_{n-1} \in \mathcal{I}\ \}$$

to be the set of unary type constructors generated by the base constructor K.

Further, let $\mathcal{T}_{\mathcal{R}}(\cdot)$ be a function that delivers sets of types and $\mathcal{T}_{\mathcal{B}}(\cdot,\cdot,\cdot)$ be functions that delivers sets of pairs of types given arguments from $\mathcal{M}_{\mathsf{K}}$. Finally, let $\mathcal{B}$ be a set of bind operations of type

$$(\mathsf{M},\mathsf{N}) \triangleright \mathsf{P} \stackrel{\text{def}}{=} \forall\, (\alpha,\beta) \in \mathcal{T}_{\mathcal{B}}(\mathsf{M},\mathsf{N},\mathsf{P}).\ \mathsf{M}\ \alpha \to (\alpha \to \mathsf{N}\ \beta) \to \mathsf{P}\ \beta$$

and $\mathcal{R}$ a set of return operations of type

$$\forall\, \alpha \in \mathcal{T}_{\mathcal{R}}(\mathsf{M}).\ \alpha \to \mathsf{M}\ \alpha$$

with $\mathsf{M},\mathsf{N},\mathsf{P} \in \mathcal{M}_{\mathsf{K}}$.

For $(\mathsf{K}, \mathcal{I}, \mathcal{B}, \mathcal{R}, \mathcal{T}_{\mathcal{B}}(\cdot,\cdot,\cdot), \mathcal{T}_{\mathcal{R}}(\cdot))$ to form a supermonad the following conditions need to hold:

Right identity: $\forall\, \mathsf{M},\mathsf{N} \in \mathcal{M}_{\mathsf{K}}$.

$$\forall\, \alpha, (\mathtt{>\!\!>\!=}) : (\mathsf{M},\mathsf{N}) \triangleright \mathsf{M}, \mathtt{return} : \alpha \to \mathsf{N}\ \alpha.$$
$$(\mathtt{>\!\!>\!=}) \in \mathcal{B} \wedge \mathtt{return} \in \mathcal{R} \wedge \alpha \in \mathcal{T}_{\mathcal{R}}(\mathsf{N}) \wedge (\alpha,\alpha) \in \mathcal{T}_{\mathcal{B}}(\mathsf{M},\mathsf{N},\mathsf{M}) \implies$$
$$[\ \forall\, m : \mathsf{M}\ \alpha.\ m \mathbin{>\!\!>\!=} \mathtt{return} \equiv m\ ]$$

Left identity: $\forall\, \mathsf{M},\mathsf{N} \in \mathcal{M}_{\mathsf{K}}$.

$$\forall\, \alpha, \beta, (\mathtt{>\!\!>\!=}) : (\mathsf{M},\mathsf{N}) \triangleright \mathsf{N}, \mathtt{return} : \alpha \to \mathsf{M}\ \alpha.$$
$$(\mathtt{>\!\!>\!=}) \in \mathcal{B} \wedge \mathtt{return} \in \mathcal{R} \wedge \alpha \in \mathcal{T}_{\mathcal{R}}(\mathsf{M}) \wedge (\alpha,\beta) \in \mathcal{T}_{\mathcal{B}}(\mathsf{M},\mathsf{N},\mathsf{N}) \implies$$
$$[\ \forall\, a : \alpha.\ \forall\, m : \alpha \to \mathsf{N}\ \beta.\ (\mathtt{return}\ a) \mathbin{>\!\!>\!=} f \equiv f\ a\ ]$$

Associativity: $\forall\, \mathsf{M},\mathsf{N},\mathsf{P},\mathsf{S},\mathsf{T} \in \mathcal{M}_{\mathsf{K}}$.

$$\forall\, \alpha, \beta, \gamma,$$
$$(\mathtt{>\!\!>\!=}_1) : (\mathsf{M},\mathsf{N}) \triangleright \mathsf{P}, (\mathtt{>\!\!>\!=}_2) : (\mathsf{S},\mathsf{T}) \triangleright \mathsf{N},$$
$$(\mathtt{>\!\!>\!=}_3) : (\mathsf{N},\mathsf{T}) \triangleright \mathsf{P}, (\mathtt{>\!\!>\!=}_4) : (\mathsf{M},\mathsf{S}) \triangleright \mathsf{N}.$$
$$(\mathtt{>\!\!>\!=}_1), (\mathtt{>\!\!>\!=}_2), (\mathtt{>\!\!>\!=}_3), (\mathtt{>\!\!>\!=}_4) \in \mathcal{B} \wedge$$
$$(\alpha,\gamma) \in \mathcal{T}_{\mathcal{B}}(\mathsf{M},\mathsf{N},\mathsf{P}) \wedge (\beta,\gamma) \in \mathcal{T}_{\mathcal{B}}(\mathsf{S},\mathsf{T},\mathsf{N}) \wedge$$
$$(\beta,\gamma) \in \mathcal{T}_{\mathcal{B}}(\mathsf{N},\mathsf{T},\mathsf{P}) \wedge (\alpha,\beta) \in \mathcal{T}_{\mathcal{B}}(\mathsf{M},\mathsf{S},\mathsf{N}) \implies$$
$$[\ \forall\, m : \mathsf{M}\ \alpha.\ \forall\, f : \alpha \to \mathsf{S}\ \beta.\ \forall\, g : \beta \to \mathsf{T}\ \gamma.$$
$$m \mathbin{>\!\!>\!=}_1 (\lambda\, x.\ f\ x \mathbin{>\!\!>\!=}_2 g) \equiv (m \mathbin{>\!\!>\!=}_4 f) \mathbin{>\!\!>\!=}_3 g\ ]$$

Functor: Every $\mathsf{M} \in \mathcal{M}_\mathsf{K}$ forms a constrained functor, providing the $\mathtt{fmap}_\mathsf{M}$ functor operation, and: $\forall\, \mathsf{M}, \mathsf{N} \in \mathcal{M}_\mathsf{K}$.

$$\forall\, \alpha, \beta, (\texttt{>>=}) : (\mathsf{M}, \mathsf{N}) \rhd \mathsf{M}, \mathtt{return} : \alpha \to \mathsf{N}\ \alpha.$$
$$(\texttt{>>=}) \in \mathcal{B} \wedge \mathtt{return} \in \mathcal{R} \wedge (\alpha, \beta) \in \mathcal{T}_\mathcal{B}(\mathsf{M}, \mathsf{N}, \mathsf{M}), \beta \in \mathcal{T}_\mathcal{R}(\mathsf{N}) \implies$$
$$[\, \forall\, f : \alpha \to \beta.\ \forall\, m : \mathsf{M}\ \alpha.$$
$$m \texttt{ >>= } (\mathtt{return} \circ f) \equiv \mathtt{fmap}_\mathsf{M}\ f\ m\,]$$

Notice that our definition of supermonads allows a set of more than one bind or return operation. This is due to the fact that a class instance containing polymorphic variables in Haskell denotes a scheme from which we can construct a function for every instantiation of those variables. Therefore, they define a set of functions that share the same implementation.

The functions $\mathcal{T}_\mathcal{R}(\cdot)$ and $\mathcal{T}_\mathcal{B}(\cdot,\cdot,\cdot)$ represent the possible constraints on the result types of our operations. The sets they return are the sets of types that instantiate the required constraints.

We proved [6] that all of the monadic notions we aim to support are indeed supermonads using our formalization in Agda.

9. Related Work

Comparison to Kmett's Approach. The basic idea of the generalized encoding of bind operations that we use has already been explored by Kmett [24] in 2007.

As we explained in Section 3 Kmett's work included a functional dependency on the `Bind` class and a specialized return operation. Both were introduced to aid type inference.

Our first case study shows that there are still many situations where manual type annotations and correct choice of return operations are necessary to resolve ambiguous types. Both of these tasks are tedious.

We do not include a functional dependency in our encoding. Our plugin already restores type inference and the functional dependency does not restrict the `Bind` class in a useful manner.

That said, Kmett's approach is more flexible than supermonads as there is no requirement for a single base constructor. This allows encoding of lifting with bind operations, meaning that lifting often can be made implicit. 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 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) [15] into `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 can 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 and require the users of supermonads to express lifting from one notion to another explicitly.

However, even if there were no concerns about the semantics of implicit lifting, 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.

Kmett did not present any laws or a theory for his approach, though we assume he intended a similarly generalized version of the standard monads laws as we presented.

Comparison to Polymonads. Polymonads [16] 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. In our previous work [9] 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 then the laws of supermonads and do not as obviously relate to the standard monad laws. Though it can be shown that the generalized forms of the standard monad laws can be derived 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 unary type constructors. This is a major disadvantage compared to supermonads, because non-phantom indices allow many interesting examples and applications of the monadic notions we aim to support.

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.

Relative Monads. Another generalization of monads are relative monads as presented by Altenkirch et al. [3]. Standard monads are essentially endofunctors with additional laws. Relative monads generalize the notion of a monad by introducing a functor between different categories instead of the same.

Unpublished work by Orchard and Mycroft [33] discusses using relative monads as a categorical model for constrained monads. In this model types (of kind $\star$) are seen as the objects and functions are seen as the morphisms of the category of Haskell. The functor of

the relative monad maps from a subcategory of Haskell which uses a subset of all types (objects) that satisfy the constraints, thereby restricting the results in the way required by constrained monads.

Since relative monads are a categorical notion they can be used as a design pattern for functional programming. Depending on the context this categorical pattern might be applied in different ways within a programming language. Therefore, we cannot say that we support relative monads, but with constrained monads we do support one of their possible applications in Haskell. We are not aware of other application of relative monads in Haskell.

Further work is required to determine the exact relationship between supermonads and relative monads.

Other Related Work. In unpublished work, McBride [26] presents a generalization of monads different from any of the generalizations we have discussed so far. In his generalization he proposes a bind and return operation with the following type signature:

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

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

There is work on a categorical generalization of applicative functors, monads and arrows in unpublished work by Rivas and Jaskelioff [39]. They exhibit the deeper connections between the three notions and unify them as monoids in monoidal categories. However, it is unclear how generalizations of the aforementioned notions relate to their work, though it may provide an approach to find a categorical description of supermonads.

In work preceding their work on polymonads Swamy et al. [44] 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 [20] 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 [42] 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 [43]. 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.

10. Conclusions and Future Work

In conclusion, we can see that supermonads, as we defined them, capture a variety of monadic notions.

We have presented a suitable representation for supermonads in Haskell together with a language extension to support type inference for the representation. The language extension is required, because the representation is too general to maintain type inference. We chose to implement our extension in form of a GHC plugin. To support our claim that type inference is restored, we present two case studies that show how seamlessly supermonads can be used as a replacement of the monadic notions they capture. The case studies also provide a stress test for our extension.

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 the implications or not. However, future work may provide a way to handle constraints on result types in a pleasant manner.

The technique we use to represent and implement supermonads should be general enough to transfer it to other notions of computation, thus, generalizing them as well. We plan on generalizing applicative functors and arrows in future work using this technique.

Another line of work involves generalizing other notions defined in Haskell's standard library. Examples for this include the classes for `MonadPlus` and `Traversable`.

At the time of writing, we are investigating promising categorical notions that capture supermonads. Future work will reflect these efforts and hopefully provide a suitable categorical model for supermonads.

We also need to verify our claim that the plugin restores the type inference capabilities that were lost when unifying the different monadic notions into one representation in Haskell.

Acknowledgments

We want to thank Jonathan Fowler, Graham Hutton and Neil Sculthorpe for their useful feedback on the preliminary version of this article. We also thank the anonymous reviewers for their helpful suggestions and feedback.

References

[1] Haskell 2010 language report, 2010. URL `https://www.haskell.org/onlinereport/haskell2010/`.

[2] S. Adams. Functional pearls efficient sets — a balancing act. *Journal of Functional Programming*, 3:553–561, 1993. doi: 10.1017/S0956796800000885.

[3] T. Altenkirch, J. Chapman, and T. Uustalu. Monads need not be endofunctors. In L. Ong, editor, *Foundations of Software Science and Computational Structures*, volume 6014 of *Lecture Notes in Computer Science*, pages 297–311. Springer Berlin Heidelberg, 2010. doi: 10.1007/978-3-642-12032-9_21.

[4] R. Atkey. Parameterised notions of computation. *Journal of Functional Programming*, 19:335–376, 2009. doi: 10.1017/S095679680900728X.

[5] M. Bolingbroke. Constraint kinds for GHC, Sept. 2011. URL `http://web.archive.org/web/20160422120734/http://blog.omega-prime.co.uk/?p=127`.

[6] J. Bracker. jbracker/polymonad-proofs, 2015. URL `https://github.com/jbracker/polymonad-proofs`.

[7] J. Bracker. jbracker/supermonad, 2016. URL `https://github.com/jbracker/supermonad`.

[8] J. Bracker and A. Gill. *Practical Aspects of Declarative Languages: 16th International Symposium, PADL 2014, San Diego, CA, USA, January 20-21, 2014. Proceedings*, chapter Sunroof: A Monadic DSL for Generating JavaScript, pages 65–80. Springer International Publishing, 2014. doi: 10.1007/978-3-319-04132-2_5.

[9] J. Bracker and H. Nilsson. Polymonad programming in Haskell. In *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages*, IFL '15, pages 3:1–3:12. ACM, 2015. doi: 10.1145/2897336.2897340.

[10] M. M. T. Chakravarty, G. Keller, and S. P. Jones. Associated type synonyms. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 241–253. ACM, 2005. doi: 10.1145/1086365.1086397.

[11] D. Devriese and F. Piessens. Information flow enforcement in monadic libraries. In *Proceedings of the 7th ACM SIGPLAN Workshop on*

Types in Language Design and Implementation, TLDI '11, pages 59–72. ACM, 2011. doi: 10.1145/1929553.1929564.

[12] I. S. Diatchki. Improving Haskell types with SMT. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell*, Haskell 2015, pages 1–10. ACM, 2015. doi: 10.1145/2804302.2804307.

[13] G. Gonzalez. index-core, 2012–2015. URL `http://hackage.haskell.org/package/index-core`.

[14] A. Gundry. A typechecker plugin for units of measure: Domain-specific constraint solving in GHC Haskell. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell*, Haskell 2015, pages 11–22. ACM, 2015. doi: 10.1145/2804302.2804305.

[15] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 48–60. ACM, 2005. doi: 10.1145/1065944.1065952.

[16] M. Hicks, G. Bierman, N. Guts, D. Leijen, and N. Swamy. Polymonadic programming. *Electronic Proceedings in Theoretical Computer Science*, 153:79–99, 2014. doi: 10.4204/EPTCS.153.7.

[17] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. doi: 10.1145/363235.363259.

[18] J. Hughes. Restricted data types in Haskell. In *Haskell Workshop*, volume 99, 1999.

[19] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67 – 111, 2000. doi: 10.1016/S0167-6423(99)00023-4.

[20] M. P. Jones. Simplifying and improving qualified types. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95, pages 160–169. ACM, 1995. doi: 10.1145/224164.224198.

[21] A. M. Joseph. *Generalized Arrows*. PhD thesis, EECS Department, University of California, Berkeley, 2014.

[22] S.-y. Katsumata. Parametric effect monads and semantics of effect systems. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 633–645. ACM, 2014. doi: 10.1145/2535838.2535846.

[23] E. Kmett. monad-param, 2006–2011. URL `http://hackage.haskell.org/package/monad-param`.

[24] E. Kmett. Parameterized monads in Haskell, July 2007. URL `https://web.archive.org/web/20140712183502/http://comonad.com/reader/2007/parameterized-monads-in-haskell/`.

[25] S. Marlow, S. P. Jones, E. Kmett, and A. Mokhov. Desugaring Haskell's do-notation into applicative operations. In *Proceedings of the 2016 ACM SIGPLAN Symposium on Haskell*, Haskell '15. ACM, 2016.

[26] C. McBride. Functional pearl: Kleisli arrows of outrageous fortune. Unpublished, 2011.

[27] C. McBride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18:1–13, 2008. doi: 10.1017/S0956796807006326.

[28] E. Moggi. *Computational Lambda-Calculus and Monads*. IEEE Computer Society Press, 1988.

[29] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55 – 92, 1991. doi: 10.1016/0890-5401(91)90052-4. Selections from 1989 IEEE Symposium on Logic in Computer Science.

[30] H. Nilsson and T. A. Nielsen. Declarative modelling for bayesian inference by shallow embedding. In *Proceedings of the 6th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, EOOLT '14, pages 39–42. ACM, 2014. doi: 10.1145/2666202.2666208.

[31] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[32] D. Orchard. effect-monad, 2013–2014. URL `http://hackage.haskell.org/package/effect-monad`.

[33] D. Orchard and A. Mycroft. Categorical programming for data types with restricted parametricity. Draft submitted and rejected by TFP 2012 post-proceedings, 2012.

[34] D. Orchard and T. Petricek. Embedding effect systems in Haskell. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 13–24. ACM, 2014. doi: 10.1145/2633357.2633368.

[35] R. Paterson. A new notation for arrows. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, pages 229–240. ACM, 2001. doi: 10.1145/507635.507664.

[36] A. Persson, E. Axelsson, and J. Svenningsson. *Generic Monadic Constructs for Embedded Languages*, pages 85–99. Springer Berlin Heidelberg, 2012. doi: 10.1007/978-3-642-34407-7_6.

[37] R. Pope, E. A. Kmett, D. Menendez, and I. Diatchki. indexed, 2004–2012. URL `http://hackage.haskell.org/package/indexed`.

[38] R. Pucella and J. A. Tov. Haskell session types with (almost) no class. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Haskell '08, pages 25–36. ACM, 2008. doi: 10.1145/1411286.1411290.

[39] E. Rivas and M. Jaskelioff. Notions of computation as monoids. Under consideration for publication in Journal of Functional Programming, 2016.

[40] N. Sculthorpe, J. Bracker, G. Giorgidze, and A. Gill. The constrained-monad problem. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 287–298. ACM, 2013. doi: 10.1145/2500365.2500602.

[41] G. Sittampalam and P. Gavin. Rmonad, 2008–2013. URL `http://hackage.haskell.org/package/rmonad`.

[42] P. J. Stuckey and M. Sulzmann. A theory of overloading. *ACM Transactions on Programming Languages and Systems*, 27(6):1216–1269, 2005. doi: 10.1145/1108970.1108974.

[43] P. J. Stuckey, M. Sulzmann, and J. Wazny. The chameleon system. In T. Frühwirth and M. Meister, editors, *First Workshop on Constraint Handling Rules: Selected papers*, pages 13–32. University of Ulm, 2004.

[44] N. Swamy, N. Guts, D. Leijen, and M. Hicks. Lightweight monadic programming in ML. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 15–27. ACM, 2011. doi: 10.1145/2034773.2034778.

[45] The GHC Team. Glasgow Haskell Compiler Users Guide, 2016. URL `http://downloads.haskell.org/~ghc/8.0.1/docs/html/users_guide/index.html`.

[46] J. A. Tov. simple-sessions, 2008–2013. URL `http://hackage.haskell.org/package/simple-sessions`.

[47] J. Vizzotto, T. Altenkirch, and A. Sabry. Structuring quantum effects: superoperators as arrows. *Mathematical Structures in Computer Science*, 16:453–468, June 2006. doi: 10.1017/S0960129506005287.

[48] P. Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 1–14. ACM, 1992. doi: 10.1145/143165.143169.

[49] P. Wadler. Monads and composable continuations. *LISP and Symbolic Computation*, 7(1):39–55, 1994. doi: 10.1007/BF01019944.

[50] P. Wadler. The marriage of effects and monads. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 63–74. ACM, 1998. doi: 10.1145/289423.289429.