Polymonad Programming in Haskell

Jan Bracker
University of Nottingham
Nottingham, United Kingdom
jzb@cs.nott.ac.uk

Henrik Nilsson
University of Nottingham
Nottingham, United Kingdom
nhn@cs.nott.ac.uk

ABSTRACT

Polymonads were recently introduced by Hicks et al. as a unified approach to programming with different notions of monads. Their work was mainly focused on foundational aspects of the approach. In this article, we show how to incorporate the notion of polymonads into Haskell, which is the first time this has been done in a full-scale language. In particular, we show how polymonads can be represented in Haskell, give a justification of the representation through proofs in Agda, and provide a plugin for the Glasgow Haskell Compiler (GHC) that enables their use in practice. Finally, we demonstrate the utility of our system by means of examples concerned with session types and the parameterized effect monad. This work provides a common representation of a number of existing approaches to generalized monads in Haskell.

CCS Concepts

•**Software and its engineering** → **Functional languages; Control structures; Constraints;** *Syntax; Semantics;*

Keywords

Glasgow Haskell Compiler; Haskell; monad; polymonad; syntactic support; type checker plugin

1. INTRODUCTION

Several different notions have been developed for expressing effectful computations in a pure functional setting, in particular in the context of Haskell. Examples include monads [25, 26, 45], arrows [15], and applicative functors [24]. Indeed, the notions of monads and arrows have proven useful enough to merit special syntactic support in Haskell [32, 1].

Over the last few years, generalizations such as constrained [37] and parameterized monads [3] have been investigated, in a quest for more flexibility and enhanced static checks partly enabled by improved support for type-level programming [20, 47]. Parameterized monads have proven to be especially popular: they have been used to model session-types [34], effect systems [30, 23], information flow control [9] and composable continuations [46].

IFL '15, September 14-16, 2015, Koblenz, Germany

DOI: http://dx.doi.org/10.1145/2897336.2897340

Unfortunately, in the setting of Haskell at least, such generalizations are incompatible with the standard notions. For example, the above generalizations require new type classes, because the standard type for bind is too specific. This incompatibility leads to problems such as:

- Hampered code reuse. For example, there are several libraries on hackage that offer different definitions of parameterized or constrained monads [33, 29, 21, 38].
- Undermined syntactic support. In particular, using more than one monad in a module requires explicit manual disambiguation through annotations for each monadic computation.
- Instance incompatibility. Although mathematically a classical monad is also a parameterized monad, we cannot give parameterized monad instances for standard monads in Haskell because their kinds differ.

Similar problems arise when generalizing other notions of computations such as arrows [18, 27].

It is therefore of interest to look for unifying notions to reduce the proliferation of approaches to effectful programming. As a start, this paper considers the case of parameterized monads by turning to *polymonads* introduced by Hicks et al. [13] to see if they can provide a practical as well as appealing solution in the context of Haskell. We do this both through a theoretical investigation and through an implementation that works with GHC through a type checker plugin.

Before delving into details, we would like to emphasize that the research question with which we ultimately are concerned is weather it is possible to design and implement a practical, unifying framework that allows various generalizations of monads to be used as seamlessly and easily as basic monads. Our current implementation is thus only a means to an end, not an end unto itself. In particular, while we believe that our implementation is useful and hope that others will find it compelling, we are not saying that our implementation at this point necessarily is the "best" way to program with parameterized monads. However, if our quest is successful, then that would pave the way for integrating such a unifying notion into a future version of Haskell (and its ecosystem of tools and libraries). At that point the approach proposed here would offer distinct advantages over handling generalized monads on a case-by-case basis, which at present is the only option.

Polymonads provide a notion that captures standard as well as parameterized monads in a way that makes them compatible with each other. A polymonad consists of a pair $(\mathscr{M}, \Sigma)$ together with associated laws. $\mathscr{M}$ is a set of unary type constructors that always includes the constructor Id, where $\mathsf{Id}\ \tau = \tau$, and Σ is a set of bind operations. Some of the type constructors in $\mathscr{M}$ are typically par-

tially applied type constructors of arity two or higher, thus accommodating parameterized monads. The bind operations in Σ have the following type schema

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

where $\mathsf{M}, \mathsf{N}, \mathsf{P} \in \mathscr{M}$. This type schema is a more general version of the one for the standard bind operation. Polymonads thus generalize the notion of a monad in a way that allows the involved type constructors and their indices to vary over a computation as long as the prerequisite laws are satisfied. Based on this we can provide a type class in Haskell that captures the notion of a bind operation for standard as well as parameterized monads.

The main problem with using polymonads in Haskell is that the standard Haskell constraint solver cannot handle the resulting class constraints due to their potential ambiguity. While Hicks et al. [13] do show that the constraints are decidable in many cases, they only do this for a small toy language [4]. Further, their technique assumes that the polymonad under consideration is known. This is a problem in Haskell, because many polymonad instances may be in scope simultaneously making it unclear which operations belong to the current polymonad and which do not.

A limitation of the foundational work of Hicks et al. [13] is that polymonads are required to be *principal* (see Section 3). This ensures that constraint solving works properly. Unfortunately, principality has so far only been established for parameterized monads with phantom indices. It might be possible to lift this limitation, but we leave that as future work and concern ourselves exclusively with polymonad with phantom indices in the following.

The present paper makes the following contributions:

- Formalization of polymonads (Section 3):
  - Full formalization of polymonads in the proof assistant Agda [28], paving the way for formally establishing properties central for a correct implementation.
  - A proof that two bind operations with the same type are extensionally equal, justifying using type classes for implementing polymonads.
  - A formal study of the union of polymonads and its principality.

  Our work revealed an imprecision in the definition of principality in Hicks et al. [13].
- The first implementation of polymonads for a full-scale language, Haskell in our case:
  - A representation of polymonads in Haskell, including a type class, common library functions and adapted prelude. (Section 4).
  - A demonstration of the utility of our system through examples concerned with session types and the effect monad (Section 4.3 and Section 4.4).
  - A plugin for the GHC type checker to solve polymonad constraints (Section 5).

Note that the work presented here is orthogonal to the recent changes in GHC's `Monad` type class hierarchy and therefore not impacted by it.

Additionally, we present a novel algorithm for detecting the correct set of bind operations for the current polymonad (Section 6). Since principality is preserved by the union of polymonads, the algorithm is not essential, but we include it regardless as it can improve efficiency and might aid future generalizations.

The paper concludes with discussions on related work (Section 7) and how limitations of the current approach might be addressed and how other monad-like notions, such as constrained monads, might be covered (Section 8).

2. MOTIVATING EXAMPLE

To give an example of the notational advantages of integrating polymonads into Haskell, we present a small application written using the session type implementation provided by Pucella and Tov [34]. It is available through the `simple-sessions` package [44] and uses the parameterized monad implementation from the `indexed` package [33]. Our example application will be a server and a client that exchange "Ping" and "Pong" messages for a set amount of three times. Session types capture the protocol statically, ensuring that the participating processes will work together correctly.

The nub of the protocol specification is captured by the types `Ping` and `Pong`:

```
type Ping = Eps :+: (String :!: String :?: Var Z)
type Pong = Eps :&: (String :?: String :!: Var Z)
```

The type `Ping` says that a process with this type can make an *active* choice (`:+:`) between, on the one hand, terminating (`Eps`), and, on the other, sending a string (`:!:`), then receiving a string (`:?:`), and then following whatever protocol that is bound to the variable with de Bruijn index zero (`Var Z`). Such variables are typically, as here, used to allow protocols to be recursive, the recursive knot being tied in a suitable context. The type `Pong` says that a process with this type can make a choice as *required* between, on the one hand, terminating and, on the other, receiving a string, sending a string, and then following a protocol bound to the variable. Note that `Ping` and `Pong` are complementary and thus fit together.

The code for the "Ping" server is as follows. The type `Cap` represents the capability to run a protocol, providing an environment that binds all free variables, while the type `Session s s' a` is the actual indexed monad, representing a computation that evolves from state `s` to state `s'` producing a result of type `a`.

```
ping :: Int -> Session (Cap (Ping, ()) Ping) () ()
ping 0 = do
    sel1; close
    where
        ma >> mb = ma >>>= \_ -> mb
ping n = do
    sel2; send "Ping"
    rsp <- recv
    io $ putStrLn rsp
    zero; ping (n - 1)
    where
        (>>=) = (>>>=)
        ma >> mb = ma >>>= \_ -> mb
```

When the counter `n` reaches zero, the server closes the connection after first having indicated that it has opted for the first choice (according to the protocol) through the session action `sel1`. If the counter is greater than zero, the repeating path is chosen. In this case the server again indicates its choice (the second choice, `sel2`), and then sends a "Ping" message and waits for the "Pong" reply. The call to `zero` restores the zeroth protocol from the environment, matching the de Bruijn index of the session variable `Var Z`, paving the way for the next repetition of `ping`.

As the session monad is indexed, it cannot be made an instance of the standard monad class. Instead, a dedicated bind operation `>>>=` is provided. A custom pre-processor then uses this bind operation for translating the `do`-notation. However, a custom pre-

processor is a somewhat heavy-handed solution, and does not work well with other monads.

An alternative is to use GHC's support for *rebindable syntax* [43, Section 7.3.16]. For translation of the `do`-notation, this means that whatever version of bind (`>>=`) and sequence (`>>`) that are in scope are used in place of the standard operations. However, unless we are in a situation where we only ever work with a single monad, in which case that one can be brought into scope once and for all while hiding the standard definitions, the implication is that we need to ensure that the right version of these operations are in scope for each `do`-block; e.g., through `where`-clauses as here. Clearly, this will quickly becomes tedious, and having nested computations involving other monads would make the situation even worse. Using rebindable syntax is expedient for many purposes, but it is not a replacement for proper overloading.

The "Pong"-client is similar to the "Ping"-server. It offers the server to either close the connection or receive its "Ping" message and respond with a "Pong":

```
pong :: Session (Cap (Pong, ()) Pong) () ()
pong = offer close $ do
    rsp <- recv
    io $ putStrLn rsp
    send "Pong"
    zero; pong
    where
        (>>=) = (>>>=)
        ma >> mb = ma >>>= \_ -> mb
```

Again, note the `where`-clause.

Finally, we can run our server and client by setting up a rendezvous between them. The server is forked into a new thread while the client runs in the current thread:

```
main :: IO ()
main = do
  rv <- newRendezvous
  _  <- forkIO $ accept rv
                 $ enter >> ping 3
  request rv $ enter >> pong
```

Here, we are using the standard monad, and, as the standard bind and sequence operations are the only ones in scope, we don't need to be explicit about which ones to use in any `where`-clause. Had the parameterized monad implementation re-used the names of the standard monadic operations, as is the case with several other libraries [38, 29, 21], we would have had to be explicit about which operators to use in this case as well.

A single, general monad notion, such as polymonads, covering indexed as well as standard monads, obviates the need for explicitly choosing the right monad by allowing the choice to be handled by overloading resolution in the standard way. Another advantage is that common library functions, such as `filterM`, `when` or `foldM`, can be provided centrally, instead of having to be reimplemented for each new monadic type. Our Haskell library [6] containing the polymonad plugin indeed provides these library function for polymonads.

3. FORMALIZATION OF POLYMONADS

We have formalized polymonads and their principality in Agda, allowing us to establish properties central for a correct implementation in Haskell. The formalization also revealed a slight imprecision in the notion of principality given in Hicks et al. [13]. The Agda sources of the formalization and proofs are available on GitHub [7].

In the interest of brevity we are going to use the following notation when referring to the type schema of a bind operation:

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

To make the notions with which we are working clear, we start by recapitulating the definitions of polymonads (Definition 1) and principal polymonads (Definition 3) from Hicks et al. [13].

Definition 1. A *polymonad* $(\mathscr{M},\Sigma)$ consists of a collection $\mathscr{M}$ of unary type constructors, with a distinguished element $\mathsf{Id} \in \mathscr{M}$, such that $\mathsf{Id}\ \tau = \tau$, and a collection Σ of bind operations such that the laws below hold. For all $\mathsf{M},\mathsf{N},\ldots,\mathsf{U} \in \mathscr{M}$:

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

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

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

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

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

The functor law ensures, that for each type constructor there is a bind operation resembling the mapping function of a functor; it also assures that this mapping function only operates on the computed value and does not produce side-effects, as to be expected from a functor. Thereby, it resembles the functor prerequisite of standard monads. The functor law we present here slightly deviates from the one presented by Hicks et al. [13]. Our version includes a separate quantification for the equation. In the original version the quantification of the equation is missing and it therefore seems as if the equation only needs to hold for the one existing bind operation. Both versions of Definition 1, the original and ours, are equivalent as we prove in our formalization.

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

The diamond and associativity laws are tied together. They express that a sequence of three monadic operations can be associated either way, and that the choice of intermediate monadic type (P or S) that may arise due to the generality of the polymonadic bind operators is irrelevant.

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

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

Definition 2. The *basis* of a polymonad is the smallest set of type constructors form which all unary type constructors of a polymonad can be obtained by partial application. △

We now turn to principality of polymonads. The following definition is a refinement of the original version, addressing the aforementioned imprecision:

Definition 3. A polymonad $(\mathscr{M}, \Sigma)$ is a *principal polymonad* if and only if for any set $F \subseteq \mathscr{M}^2$ with $F \neq \emptyset$, and any $\{\mathsf{M}_1, \mathsf{M}_2\} \subseteq \mathscr{M}$ such that

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

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

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

We call $\hat{\mathsf{M}}$ the principal join of F and write it as $\bigsqcup F$. △

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

As an example, assume we have two bind operations with types $(\mathsf{M}, \mathsf{N}) \rhd \mathsf{t}$ and $(\mathsf{t}, \mathsf{Id}) \rhd \mathsf{P}$, where $\mathsf{M}, \mathsf{N}, \mathsf{P} \in \mathscr{M}$ and t is an ambiguous variable that could be resolved to one of a number of type constructors. The closure law tells us that there has to exist a bind operation of type $(\mathsf{M}, \mathsf{N}) \rhd \mathsf{P}$. Since we know that our polymonad is principal this implies that there has to exist a type constructor that is suitable for t and that is $\bigsqcup F$. That this choice yields a correct and sound solution was shown by Hicks et al. [13].

The definition of principality presented here deviates from the original definition in that it insists that F be non-empty. This restriction is important, because if $F = \emptyset$ the preconditions are always fulfilled and that means there has to exist an $\hat{\mathsf{M}}$ for any M_1 and M_2 that we choose. Personal communication with the authors of the polymonad programming paper [13] confirmed that this behavior is not intended.

We have verified that standard monads and certain parameterized monads give rise to polymonads using our formalization. We also verified that standard monads and parameterized monads with phantom indices are principal. These results were mentioned in an unpublished paper by Guts et al. [12] and the polymonad programming paper [13].

The formalization of polymonads in Agda was straightforward. The preconditions of the lemmata presented in the following sections mostly arose in the process of trying to prove them without the respective precondition. We assume $\mathsf{Id}\ \tau \stackrel{\text{def}}{=} \tau$ from this point on.

3.1 Uniqueness of bind operations

Although polymonads do not restrict the number of bind operations with the same type, we have shown that bind operations with the same type have the same behavior; i.e., they are extensionally equal:

Lemma 1 (Uniqueness of bind operations)**.** Let $(\mathscr{M}, \Sigma)$ be a polymonad. Then, for all $\mathsf{M}, \mathsf{N}, \mathsf{P} \in \mathscr{M}$:

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

Our result justifies using type classes for overloading the bind operation: there can only be one instance of a type class per combination of arguments to the type class head, but, as two bind operations with the same type are equal, that suffices.

The same argument also works in the opposite direction. If there is a constraint without type variables and several overlapping type class instances that match it, then we can pick an arbitrary matching instance without jeopardizing the runtime behavior.

3.2 Union of polymonads

In the introduction we mentioned that the polymonad instances are all collected centrally in Haskell. It is therefore important to know whether a union of polymonads constitutes a polymonad in its own right, and whether polymonad union preserves principality. We start by considering the first question.

Lemma 2 (Identity polymonad)**.** Define the function `bindId` as

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

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

We assume that Id is the distinguished type constructor in all of our polymonads, as will be the case in Haskell. Based on this assumption we prove that the union of polymonads, if they share the `bindId` bind operation, is a polymonad itself.

Lemma 3 (Union of polymonads)**.** Let $(\mathscr{M}_1, \Sigma_1)$ and $(\mathscr{M}_2, \Sigma_2)$ be polymonads with the same distinguished type constructor Id such that

(1) $\mathscr{M}_{\mathsf{Id}} = \mathscr{M}_1 \cap \mathscr{M}_2$ and $\Sigma_{\mathsf{Id}} = \Sigma_1 \cap \Sigma_2$

(2) $\forall\, b : (\mathsf{M}, \mathsf{N}) \rhd \mathsf{Id}.\ [b \in \Sigma_1 \implies \mathsf{M} = \mathsf{Id} \wedge \mathsf{N} = \mathsf{Id}]$

(3) $\forall\, b : (\mathsf{M}, \mathsf{N}) \rhd \mathsf{Id}.\ [b \in \Sigma_2 \implies \mathsf{M} = \mathsf{Id} \wedge \mathsf{N} = \mathsf{Id}]$

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

$$\forall\, b : (\mathsf{M}, \mathsf{N}) \rhd \mathsf{Id}.\ [b \in (\Sigma_1 \cup \Sigma_2) \implies \mathsf{M} = \mathsf{Id} \wedge \mathsf{N} = \mathsf{Id}]$$

If $(\mathscr{M}_1 \cup \mathscr{M}_2, \Sigma_1 \cup \Sigma_2)$ forms a polymonad we call $(\mathscr{M}_1, \Sigma_1)$ and $(\mathscr{M}_2, \Sigma_2)$ *unionable*. □

Thus, we know that, under mild restrictions, the union of polymonads remains a polymonad. We consider the restrictions of Lemma 3 mild because, at least in the context of Haskell, each of the excluded bind operations with type $(\mathsf{M}, \mathsf{N}) \rhd \mathsf{Id}$ resembles a function to run the computations of the polymonad without giving the additional arguments that are usually required to execute the side-effects modeled by it. As far as we are aware, the only polymonads that could provide such bind operations are the polymonads derived from the identity-monad (or monads isomorphic to it) and Haskell's `ST`-monad. The bind operation of the identity polymonad is allowed by the precondition and therefore the identity polymonad remains unionable. `ST` can still be made a unionable polymonad as long as it does not introduce bind operations of the form $(\mathsf{M}, \mathsf{N}) \rhd \mathsf{Id}$. Although the `ST` monad is generalized by the `IO` monad the same problem does not arise there, because there is no safe way to compute a pure value from an `IO` computation. Even if we consider the use of unsafe functions, such as `unsafePerformIO`, both, `ST` and `IO`, remain unionable polymonad as long as they do not introduce bind operations of the form $(\mathsf{M}, \mathsf{N}) \rhd \mathsf{Id}$.

Both precondition 2 and 3 are important because, without them, the polymonad laws would require bind operations that involve type constructors from $\mathscr{M}_1$ and $\mathscr{M}_2$ as well as the existence of bind operations with type $(\mathsf{M},\mathsf{N}) \rhd \mathsf{Id}$. Many popular monads and generalized monads fail to meet that requirement.

3.3 Preservation of principality

We now turn to the question of whether polymonad union preserves principality. It turns out that it does, and we have formally proved this (postulating some basic results from classic set theory):

Lemma 4 (Polymonad union preserves principality)**.** Let $(\mathscr{M}_1,\Sigma_1)$ and $(\mathscr{M}_2,\Sigma_2)$ be polymonads with the same distinguished type constructor Id that are unionable and principal. Then their union $(\mathscr{M}_1 \cup \mathscr{M}_2, \Sigma_1 \cup \Sigma_2)$ is a principal polymonad as well. □

All proofs are straightforward. The interested reader is referred to our formalization for details [7].

4. POLYMONADS IN HASKELL

4.1 Representation

In Haskell we use the `Identity` type from the library `base` as our distinguished Id type constructor, because `Identity` is isomorphic to Id.

A key advantage of supporting polymonads in Haskell is that this allows the `do`-notation to be used without having to be explicit about which monad to use: cf. Section 2. An implementation of polymonads thus needs to provide a replacement for the core functions, `>>=`, `>>`, `return` and `fail`, used to translate the `do`-notation.

The `>>=` and `>>`-operator are represented through the `Polymonad` type class:

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

The functions `return` and `fail` are regular polymorphic functions:

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

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

When implementing a polymonad the programmer needs to provide instances for all bind operations required by the polymonad laws (Definition 1). An example of this can be seen in Section 4.2. In particular, an instance `Polymonad Identity Identity m`, serving as the return operation of the monad, has to be provided, enabling the use of `return` as a convenient shorthand. As mentioned earlier, using type classes to represent the bind operations is valid because there can be at most one semantic version for each type of bind operation. Overlapping instances are not a problem for the same reason.

To use our polymonad implementation in a module, the programmer has to do three things:

- Enable the GHC language extension `RebindableSyntax` [43, Section 7.3.16]. This enables the `>>=`-operator currently in scope to be used in `do`-blocks instead of the standard one.
- Import `Control.Polymonad.Prelude`. This module provides all the functionality of the standard `Prelude`, which is not imported by default when rebindable syntax is enabled, except that the standard monad functions are replaced with those for polymonads.
- Activate our polymonad type checker plugin by inserting the following line at the top of the module:

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

An example of a module that performs these steps can be seen in Figure 1. The source code for the examples in this section and those in Section 5 are available on GitHub [6].

4.2 Example: Standard monad

Every standard monad is a polymonad. An unpublished paper by Guts et al. [12] shows how to define a polymonad version of a given monad.

Lemma 5. If M is a monad with `return` and `>>=` as its return and bind operations, then M forms the polymonad $(\mathscr{M}_\mathsf{M},\Sigma_\mathsf{M})$, where:

$$\mathscr{M}_\mathsf{M} \stackrel{\text{def}}{=} \{\mathsf{Id},\mathsf{M}\}$$

$$\begin{aligned}\Sigma_\mathsf{M} \stackrel{\text{def}}{=} \{&(\lambda\texttt{m f.f m}) : (\mathsf{Id},\mathsf{Id}) \rhd \mathsf{Id},\\ &(\texttt{>>=}) : (\mathsf{M},\mathsf{M}) \rhd \mathsf{M},\\ &(\lambda\texttt{m f.return (f m)}) : (\mathsf{Id},\mathsf{Id}) \rhd \mathsf{M},\\ &(\lambda\texttt{m f.m >>= (}\lambda\texttt{x.return (f x))}) : (\mathsf{M},\mathsf{Id}) \rhd \mathsf{M},\\ &(\lambda\texttt{m f.f m}) : (\mathsf{Id},\mathsf{M}) \rhd \mathsf{M}\}\end{aligned}$$

□

We start with the `>>=`-operator and the `return`-function, both of which we require to compose computations and lift pure values into a monadic computation. Therefore, they supply the binds of type $(\mathsf{M},\mathsf{M}) \rhd \mathsf{M}$ and $(\mathsf{Id},\mathsf{Id}) \rhd \mathsf{M}$. The functor law then requires the existence of the $(\mathsf{M},\mathsf{Id}) \rhd \mathsf{M}$ and $(\mathsf{Id},\mathsf{Id}) \rhd \mathsf{Id}$ bind operation and the paired morphism law requires the $(\mathsf{Id},\mathsf{M}) \rhd \mathsf{M}$ bind operation. Once we have all five bind operations all of the other laws work out as well.

Given this lemma we can create polymonad instances that work for all monads:

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

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

instance P.Monad m => Polymonad m m m where
  m >>= f = m P.>>= f

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

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

These instances allow programmers to easily port existing monadic code to polymonads by just updating the header of their modules as described in Section 4.1. The example also illustrates how the identity polymonad is part of every polymonad and why it needs to be handled specially within the union of polymonads (Lemma 3).

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

module ExamplePolymonadModule where

import Control.Polymonad.Prelude
```

Figure 1: Example of a module header that enables the use of polymonads.

4.3 Example: Hoare monad

To give an example of how a parameterized monad can become a polymonad, we will look at the kind of parameterized monad [46, 3] that was used in our motivating example. The bind and return operation of this kind of parameterized monad closely resemble Hoare logic [14], where bind is the rule of composition and return is the empty statement axiom.

```
class HoareMonad m where
  hoareBind :: m i j a -> (a -> m j k b)
               -> m i k b
  hoareRet  :: a -> m i i a
```

The polymonad instances for `HoareMonad` are given in Figure 2. The necessity of each instance is given by a lemma that is analogue to Lemma 5. Note that the type class `Polymonad` now describes Hoare monads and standard monads alike. There is no need to introduce different type classes for different monadic notions. (The introduction of the `HoareMonad` class is only a convenience: we could just as well have given the instances for the `Session` type constructor directly.) This promotes code reuse, because we can now provide library functions and utilities for polymonads once and for all, rather than separately for each monadic notion.

Any Hoare-like polymonad can now be used by simply providing the appropriate bind and return operations through an instance of the class `HoareMonad`. In the case of the `simple-sessions` package [44] the instance is simply:

```
instance HoareMonad Session where
  hoareBind = (>>>=)
  hoareRet  = ireturn
```

where `>>>=` and `ireturn` are supplied by the `indexed` package [33] that was used to implement `simple-sessions`. By following our guide at the end of Section 4.1, the programmer can now implement the server and client of the motivating example as follows:

```
type Ping = Eps :+: (String :!: String :?: Var Z)
type Pong = Eps :&: (String :?: String :!: Var Z)

main :: IO ()
main = do
  rv <- newRendezvous
  _  <- forkIO $ accept rv
                 $ enter >> ping 3
  request rv $ enter >> pong

ping :: Int -> Session (Cap (Ping, ()) Ping) () ()
ping 0 = do
  sel1; close
ping n = do
  sel2; send "Ping"
  rsp <- recv
  io $ putStrLn rsp
  zero; ping (n - 1)

pong :: Session (Cap (Pong, ()) Pong) () ()
pong = offer close $ do
  rsp <- recv
  io $ putStrLn rsp
  send "Pong"
  zero
  pong
```

The `where`-clauses have been removed as it is no longer necessary to specify which bind and return operation to use where, resulting in code that is significantly less cluttered and thus easier to both write and maintain. In particular, the `do`-notation is used just like for standard monads: proper syntactic support for monadic programming has been regained.

4.4 Example: Effect monad

As a second example, we look at the effect monad [19] and how its Haskell realization [30] can be used as a polymonad. The effect monad provides a composable way of modeling effects. It has one index describing the (possible) effects of a computation. This index has a monoidal structure, where the neutral element indicates that there are no effects, and the monoidal operation combines the effects of two computations. The `effect-monad` [29] package provides the following type class capturing these ideas:

```
class Effect m where
  type Unit m :: k
  type Plus m f g :: k
  type Inv m f g :: Constraint

  return :: a -> m (Unit m) a
  (>>=) :: Inv m f g
        => m f a -> (a -> m g b)
           -> m (Plus m f g) b
  (>>)  :: Inv m f g
        => m f a -> m g b -> m (Plus m f g) b
```

`Unit`, `Plus` and `Inv` are associated type synonyms that describe the index monoid. `Unit` is the neutral element, used in the type signature of `return` to show that return has no side effects. `Plus` is the monoidal operation. The signatures of bind and sequence show how the effects of the two constituent computations are combined in the type of the overall computation. Finally, the `Inv`-synonym gives constraints that are necessary to ensure the correct monoidal behavior of the indices and `Plus`.

Figure 3 demonstrates how to implement the `Polymonad` instances for effect monads. Note that the implementation is essentially the same as for Hoare monads. The major difference are the additional constraints describing the monoidal structure of the indices. Most of these constraints are a direct consequence of the types involved in the implementation. We want to highlight the validity of a particular constraint that may cause confusion:

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

Because it is recursive, it may seem unsolvable. However, t is solvable as long as the `Effect` instance ensures that its instantiations of `Unit` and `Plus` follow the monoid laws. As the constraint encodes

```
instance HoareMonad m => Polymonad (m i j) (m j k) (m i k) where
  (>>=) = hoareBind

instance HoareMonad m => Polymonad Identity Identity (m i i) where
  (>>=) ma f = (hoareRet . runIdentity . f . runIdentity) ma

instance HoareMonad m => Polymonad (m i j) Identity (m i j) where
  (>>=) ma f = hoareBind ma (hoareRet . runIdentity . f)

instance HoareMonad m => Polymonad Identity (m i j) (m i j) where
  (>>=) ma f = f (runIdentity ma)
```

Figure 2: `Polymonad` instances for Hoare monads.

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

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

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

instance (Effect m) => Polymonad Identity (m (g :: k)) (m (g :: k)) where
  a >>= f = f (runIdentity a)
```

Figure 3: `Polymonad` instances for certain effect monads.

the right identity of a monoid, `Plus m f (Unit m)` will always evaluate to `f` in a law abiding instance and the constraint becomes trivial.

The kind annotations are necessary: without them GHC would infer kind `*` that would not match the lifted data kind indices that are usually used with effect monads. We have to use type equality constraints to specify the shape of the indices, because type synonym applications are not allowed to appear on the right side of the instance head. To implement the instances we need to activate the following language extensions: `TypeFamilies`, `DataKinds`, `PolyKinds`, `FlexibleInstances`, `UndecidableInstances` and `MultiParamTypeClasses`. The qualifier `E` refers to the module `Control.Effect` from the `effect-monad` package.

We will now illustrate this using a specific effect monad `Counter`, first directly, without using polymonads, and then using our polymonad extension. `Counter` is an instance of `Effect` where the index is a type-level natural number and the monoidal operation is addition. The index can be used to count how often certain operations have been invoked and thereby limit the use of those operations. The `tick` function sets the counter to `1`. Consequently, the type of a combined computation will reflect how many `tick`s it contains. Operationally, `tick` is just the identity function:

```
tick :: a -> Counter 1 a
```

For our example we define a function `specialOp`, which adds two numbers. We want to limit the use of our special operation and therefore it increments the type-level counter.

```
specialOp :: Int -> Int -> Counter 1 Int
specialOp n m = tick (n + m)
```

We then define a function, `limitedOp` where we, through its type signature, promise that our special operation is called exactly three times. In the body of the function we then call our special operation three times to meet the promise:

```
limitedOp :: Int -> Int -> Int -> Int
             -> Counter 3 Int
limitedOp a b c d = do
  ab  <- specialOp a b
  abc <- specialOp ab c
  specialOp abc d
  where (>>=) :: Counter n a -> (a -> Counter m b)
                 -> Counter (n + m) b
        (>>=) = (E.>>=)
        fail = E.fail
        return :: a -> Counter 0 a
        return = E.return
```

Each call increments the type-level counter as per the type of bind. As in our motivating example, without our polymonad extension, we need to give a `where`-clause for the `do`-block to specify which monadic operations to use. There are two important things to note here. First, the type signature for the bind operation in the `where`-clause is required. This requirement arises because the correct kind for the indices cannot be inferred. Second, we have to provide a definition for `return`, despite it not being used here. This is a peculiarity of the implementation of rebindable syntax: the appropriate `return` needs to be in scope even if it is not used in a particular `do`-block. It might be possible to relax this requirement through a refined implementation of rebindable syntax, but at present it can lead to some confusing errors.

To call our `limitedOp` we write the following `main`-function:

```
main :: IO ()
main = do
  print $ forget (limitedOp 1 2 3 4)
  where return :: (Monad m) => a -> m a
        return = P.return
```

The function `forget` runs the `Counter` effect monad. Again we are required to define `return`, which this time is the standard one as we are in the IO monad.

Note that type signatures are required for `return` in both cases.

The reason for this is the monomorphism restriction [1, Section 4.5.5]. If we activate the extension `NoMonomorphismRestriction`, the type signature would no longer be required. Without a type signature and the monomorphism restriction in place, GHC tries to pick an appropriate instance for `m`. This fails with an ambiguity error as `return` is not used.

We did not have these issues in our motivating example because the `simple-sessions` package avoided a name clash by calling its return operation `ireturn`. Therefore, the rebindable syntax translation picked the *standard* `return` from the Prelude, which is in scope globally in our case, despite the monadic computation of the `do`-block being of a different type. This behavior can lead to interesting errors if a `return` is introduced into the code at some later point.

If we instead make use of our polymonad extension, all `where`-clauses can be removed as appropriate bind operations are chosen automatically:

```
main :: IO ()
main = do
  print $ forget (limitedOp 1 2 3 4)

specialOp :: Int -> Int -> Counter 1 Int
specialOp n m = tick (n + m)

limitedOp :: Int -> Int -> Int -> Int
             -> Counter 3 Int
limitedOp a b c d = do
  ab <- specialOp a b
  abc <- specialOp ab c
  specialOp abc d
```

The resulting code is again less cluttered and easier to write and maintain. It is no longer necessary to specify which bind and return operations to use. The `do`-notation can be used the same way as with standard monads: syntactic support has been regained.

While the above example is neat, it should be pointed out that there are many instances of effect monads where the index is not phantom; i.e., where the index affects the runtime behavior. Such effect monads are not supported by the current polymonad theory (Section 1) nor by the solving algorithm. Consequently, our plugin may pick the wrong instances and produce programs yielding unintentional results. We hope to lift this restriction in future work (Section 8).

5. IMPLEMENTATION OF THE GHC PLUGIN

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

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

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

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

Note the ambiguous type variable `m`: as it does not appear in the type, there will be no way to decide what type to instantiate it with in order to pick the right `Polymonad` instance. This is why programming with polymonads requires special support.

Another issue that may arise are overlapping instances. An example for this can be seen in the standard monad instances presented in Section 4.2. In case of the `Identity` type constructor several of these instances can match, meaning we need to choose one. The uniqueness of bind operations (Lemma 1) ensures that we can choose arbitrarily: the choice is inconsequential.

The issues described above can be addressed during the constraint solving step of GHC's type checker. Since version 7.10, GHC supports a plugin interface for extending the constraint solver [43, Section 9.3.4]. The plugins are provided as Haskell modules that GHC loads during compilation. Our polymonad plugin is implemented using this interface and solves polymonad constraints.

A number of type-system extensions have already been realized using the plugin interface; for example, type level natural numbers [10] and units of measure [11]. We refer the reader to this earlier work for a more in-depth description of the plugin mechanism itself, contenting ourselves with only a brief explanation here.

For each program fragment, GHC collects the set of given constraints; e.g., as provided by the type signature of a function. It also collects a set of derived constraints; e.g., derived by other plugins or language extensions. Type checking the fragment results in a set of wanted constraints that need to be solved. The constraint solver iteratively tries to solve these. If the constraint solver is unable to make progress, it asks the available type checker plugins for help. A plugin processes the given, derived and wanted constraints of the fragment and, if it can make progress, delivers any new derived constraints and evidence for any wanted constraints that it was able to solve.

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

- Identify the `Polymonad` class and `Identity` type constructor as it cannot proceed without them.
- If there are `Polymonad` constraints that do not contain any ambiguous type variables, choose an instance for them and provide their evidence.
- Look at the `Polymonad` instances and constraints to invoke the detection algorithm and figure out which polymonads we are currently working with. There can be several polymonads involved in the constraints because local `do`-blocks for different polymonads can be used within a `do`-block. The following steps are then applied to each polymonad and its associated constraints:
- Try to simplify the constraints using the simplification rules presented in Hicks et al. [13]. This reduces the number of constraints to solve and also makes error messages more readable. If simplification made any progress, control is given back to GHC. It can then try to solve the constraints using the newly available information. Should GHC get stuck again, it will ask the plugin for further assistance.
- If the simplification did not make any progress, we invoke the solving algorithm that is based on the coherence proof of Hicks et al. [13].

The polymonad plugin is fully implemented, but there are still some examples for which issues arise. We will return to this in Section 8.

6. DETECTION OF BIND-OPERATIONS

The simplification and solving algorithm presented by Hicks et al. [13] is based on the knowledge of the current polymonad. Since we know that principal polymonads preserve principality under union,

a detection algorithm to determine the correct set of bind operations for a polymonad is not necessary. However, such an algorithm can still be useful. Firstly, it can improve the overall efficiency of the plugin. Secondly, and more importantly, detecting the correct set of bind operations may also be necessary in the setting of further generalizations (Section 8).

Assumptions.

The detection algorithm assumes that the basis (Definition 2) of a polymonad has no elements in common with any other polymonad basis except for Id. Further, it is also assumed that all type constructors in the basis of the polymonad in question are used in a `do`-block.

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

Algorithm.

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

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

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

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

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

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

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

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

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

Correctness.

Formally proving the correctness of this algorithm is complicated and remains future work. We content ourselves by stating the key points for the correctness in the remainder of this section.

Due to our assumption that every `do`-block uses all type constructors in the basis of the polymonad it is applying, we know that our algorithm cannot collect more or fewer type constructors than belong to that polymonad. We can also be sure that all relevant instances are found, because we assume that polymonads do not share type constructors.

7. RELATED WORK

Swamy et al. [42] previously worked on automatically inserting bind and return operations into programs as necessary. This allows a programmer to mix pure and monadic computations without having to explicitly lift values or mechanically insert bind operations. The system also covers morphisms between monads. If morphisms are provided, the system automatically picks the correct overall monad and lifts values and computations through the provided morphisms as necessary. The polymonad programming paper [13] builds on this work, generalizing the approach to polymonads.

Edward Kmett developed a package called *monad-param* [21]. His package introduces a generalization of monads that appears very similar to polymonads:

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

Kmetts class requires all involved type constructors to be functors and also separates the `return`-function from his `Bind`-class. The major differences to polymonads are the imposed functional dependency and that the `return`-function always returns its results in the `Identity` monad. As he describes in a blog post [22], the functional dependency is necessary to make type inference in Haskell feasible when programming with Kmetts generalization. The drawback of the functional dependencies is that they prohibit morphisms into several different result type constructors. If we have three monads M_1, M_2, M_3, and we know how to transform instances of M_1 into and instance of M_2 or M_3, then we can give an instance of `Bind` M_1 `Identity` M_2 or `Bind` M_1 `Identity` M_3, but we can't provide both instances. The `return`-function being hard-wired to the `Identity` monad results in the requirement to manually lift monadic computations in some cases. Kmett also does not state which laws should hold and he does not give rules for which instances are necessary for his approach to work properly. Although he does point out a pattern for instances involving `Identity` that is very similar to the instances required by a polymonad.

Another generalization of monads is the notion of relative monads introduced by Altenkirch et al. [2]. Standard monads are essentially endofunctors, with additional laws. Relative monads generalize the notion of a monad by looking at functors between different

categories instead of endofunctors.

Jones [16] suggests the integration of custom improvements for constraints within Haskell. Custom improvements associate patterns of constraints containing open type variables with equations involving those type variables to aid constraint solving. Stuckey and Sulzmann [40] developed a theory of constraint handling rules for functional languages with constraint systems based on this idea. Though the theory behind constraint handling rules shows that they are Turing complete [39], it is unclear for us how to formulate the solving algorithm for polymonads using this approach. Another problem is that there is no working implementation of constraint handling rules for GHC. Although there was an implementation in form of the custom language Chameleon [41], this implementation is not publicly available anymore. Thus, at present, GHC plugins are to our knowledge the most practical way of making polymonads work with a full-scale language.

Unpublished work by Rivas and Jaskelioff [35] presents a categorical generalization of applicative functors, monads and arrows as monoids in monoidal categories. This work unifies these different notions of computation into one categorical framework, thereby exhibiting their deeper connections. How the different *generalizations* of monads that we are concerned with here translate into this framework is, however, beyond the scope of their work. However, their insights might be useful for generalizing arrows and applicative functors into "polyarrows" and "applicative polyfunctors" in a similar way to how polymonads generalize the notion of a monad.

8. FUTURE WORK

The polymonad paper [13] identifies a few limitations of the approach. In particular, it is assumed that the indices of a polymonad are phantom arguments; i.e., that they do not influence the runtime behavior of the polymonad. Lifting this restriction would be interesting as that would allow for polymonads such as delimited continuations, or state monads where the type of the state can change in the course of the computation. We think it may be possible to lift this restriction, although we suspect that non-phantom indices in many cases would break principality, which in turn would necessitate generalizing the constraint solving. On the other hand, it may be that our detection algorithm (or some variation of it) could turn out to be useful in a setting where principality does not hold more generally.

Another limitation of the theory of polymonads presented in Hicks et al. [13] is that it only allows constraints on the indices of type constructors, but not for the result type. Constrained monads [37] thus do not fit into the present polymonad framework. We suspect that the introduction of constrained polymonads is possible by using techniques similar to those used when introducing constrained monads. For constrained monads a popular approach is to use associated constraint synonyms [5, 8, 31]. A realization of this approach may look as follows:

```
class Polymonad m n p where
  type Constr m n p a b :: Constraint
  (>>=) :: (Constr m n p a b)
        => m a -> (a -> n b) -> p b
```

The associated type synonym of constraint kind then allows instances to supply constraints for the result types of the bind operation. It remains future work to ensure that such an approach does not break the polymonad theory or the constraint solving process.

GHC supports many Haskell extensions. Of relevance here are the extensions to the type system, such as type families [8, 36]. So far, we have focused on supporting those type system extensions that were needed for our examples, including type families, but we have yet to carry out comprehensive testing. At present, we are aware of one problem related to the production of evidence for some polymonad instances, leading to warnings in the core linting step of GHC. The discussion of the problem on the GHC mailing list led to the conclusion that the evidence produced by our plugin revealed a bug in the implementation of GHC. We have reported the issue in the GHC bug tracker[1].

However, we want to emphasize that all examples presented in this paper work. When compiling modules using our polymonad plugin, we currently recommend enabling the option `-dcore-lint` to ensure that such errors do not get ignored silently. We have yet to cover extensions that were not used in our examples, including functional dependencies [17], leaving that as future work.

9. CONCLUSIONS

We provided a representation of polymonads in Haskell and a plugin for GHC that enables their use as a possible unifying notion for generalizations of monads and standard monads. This is the first time support for polymonads has been added to a full-scale language.

To provide a solid foundation for our work, we formalized polymonads in the theorem prover Agda to establish that the actions of our plugin are sound. We proved that choosing between two bind operations with the same type is sound, because they are extensionally equal. We also studied the union of polymonads and how it preserves principality.

Although principality is preserved under polymonad union, we provide a detection algorithm to detect the polymonad used and collect polymonad instances relevant for that polymonad. This may be useful when we look at future generalizations.

All of this work enables the use of polymonads as a unifying notion for standard monads and parameterized monads with phantom indices. This obviates the need to give several different notions of monad-like structures by unifying them into a single type class. Code reuse is also promoted as utility functions can now be defined once for all polymonads, rather than providing separate versions for different monadic notions. Finally, polymonads re-establish proper syntactic support for all supported monadic structures through the `do`-notation. Without polymonads, it becomes tedious and error prone to use more than one monad notion at a time as the right monad notion needs to be brought into scope explicitly for each `do`-block. With polymonads, the `do`-notation can be used without any additional "annotations", just like standard monads.

In conclusion, the work so far is a step towards a general unified notion of monad-like structures and their support in Haskell. There is still work to be done. In particular, the polymonad constraint solving algorithm has to be generalized such that it also supports parameterized monads with non-phantom indices and constrained monads.

Acknowledgements

We acknowledge the help of Michael Hicks and Nikhil Swamy in answering questions about their work on polymonads and giving us a better understanding of the material.

We also need to thank Richard Eisenberg and Adam Gundry for their continued support and comprehensive answers to our questions about GHC, its internals and the type checker plugin infrastructure provided by it.

A special thanks to Jonathan Fowler for his insights and helpful comments on the proof that union preserves principality.

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

Finally, we thank Paolo Capriotti, Jonathan Fowler, Graham Hutton, and the anonymous reviewers for useful feedback on preliminary versions of the paper.

References

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

[2] 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.

[3] R. Atkey. Parameterised notions of computation. *Journal of Functional Programming*, 19:335–376, 2009.

[4] G. Bierman, N. Guts, M. Hicks, D. Leijen, and N. Swamy. Type coercions for program rewriting. http://research.microsoft.com/en-us/projects/coco/.

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

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

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

[8] M. M. T. Chakravarty, G. Keller, and S. P. Jones. Associated type synonyms. *SIGPLAN Notices*, 40(9):241–253, Sept. 2005.

[9] 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, New York, NY, USA, 2011. ACM.

[10] I. S. Diatchki. Improving haskell types with SMT. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell*, Haskell 2015, pages 1–10, New York, NY, USA, 2015. ACM.

[11] 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, New York, NY, USA, 2015. ACM.

[12] N. Guts, M. Hicks, N. Swamy, D. Leijen, and G. Bierman. Polymonads. Technical report, University of Maryland Department of Computer Science, 2012. Extended version of POPL'13 submission.

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

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

[15] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67 – 111, 2000.

[16] 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, New York, NY, USA, 1995. ACM.

[17] M. P. Jones. Type classes with functional dependencies. In G. Smolka, editor, *Programming Languages and Systems*, volume 1782 of *Lecture Notes in Computer Science*, pages 230–244. Springer Berlin Heidelberg, 2000.

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

[19] S.-y. Katsumata. Parametric effect monads and semantics of effect systems. *SIGPLAN Notices*, 49(1):633–645, Jan. 2014.

[20] O. Kiselyov, S. P. Jones, and C. chieh Shan. Fun with type functions. In A. Roscoe, C. B. Jones, and K. R. Wood, editors, *Reflections on the Work of C.A.R. Hoare*, pages 301–331. Springer London, 2010.

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

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

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

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

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

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

[27] 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, New York, NY, USA, 2014. ACM.

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

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

[30] 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, New York, NY, USA, 2014. ACM.

[31] D. Orchard and T. Schrijvers. Haskell type constraints unleashed. In M. Blume, N. Kobayashi, and G. Vidal, editors, *Functional and Logic Programming*, volume 6009 of *Lecture Notes in Computer Science*, pages 56–71. Springer Berlin Heidelberg, 2010.

[32] R. Paterson. A new notation for arrows. *SIGPLAN Notices*, 36(10):229–240, Oct. 2001.

[33] R. Pope, E. A. Kmett, D. Menendez, and I. Diatchki. indexed, 2004–2012. https://github.com/reinerp/indexed.

[34] R. Pucella and J. A. Tov. Haskell session types with (almost) no class. *ACM SIGPLAN Notices*, 44(2):25–36, Sept. 2008.

[35] E. Rivas and M. Jaskelioff. Notions of computation as monoids. Submitted to the Journal of Functional Programming, 2014.

[36] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. *SIGPLAN Notices*, 43(9):51–62, Sept. 2008.

[37] N. Sculthorpe, J. Bracker, G. Giorgidze, and A. Gill. The constrained-monad problem. *ACM SIGPLAN Notices*, 48(9):287–298, 2013.

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

[39] J. Sneyers, T. Schrijvers, and B. Demoen. The computational power and complexity of constraint handling rules. *ACM Transactions on Programming Languages and Systems*, 31(2):8:1–8:42, Feb. 2009.

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

[41] 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.

[42] N. Swamy, N. Guts, D. Leijen, and M. Hicks. Lightweight monadic programming in ML. *ACM SIGPLAN Notices*, 46(9):15–27, 2011.

[43] The GHC Team. The Glorious Glasgow Haskell Compilation System User's Guide, 2015. `http://downloads.haskell.org/~ghc/7.10.1/docs/html/users_guide/index.html`.

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

[45] 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, New York, NY, USA, 1992. ACM.

[46] P. Wadler. Monads and composable continuations. *LISP and Symbolic Computation*, 7(1):39–55, 1994.

[47] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66, New York, NY, USA, 2012. ACM.