# The Constrained-Monad Problem

(Corrected Version, 9th June 2014)


Neil Sculthorpe
ITTC
The University of Kansas
neil@ittc.ku.edu

Jan Bracker
Institut für Informatik
Christian-Albrechts-Universität
jbra@informatik.uni-kiel.de

George Giorgidze
Institut für Informatik
Universität Tübingen
george.giorgidze@uni-tuebingen.de

Andy Gill
EECS / ITTC
The University of Kansas
andygill@ittc.ku.edu



## Abstract

In Haskell, there are many data types that would form monads were it not for the presence of type-class constraints on the operations on that data type. This is a frustrating problem in practice, because there is a considerable amount of support and infrastructure for monads that these data types cannot use. Using several examples, we show that a monadic computation can be restructured into a normal form such that the standard monad class *can* be used. The technique is not specific to monads, and we show how it can also be applied to other structures, such as applicative functors. One significant use case for this technique is *domain-specific languages*, where it is often desirable to compile a deep embedding of a computation to some other language, which requires restricting the types that can appear in that computation.



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

***Keywords*** Class Constraints; Monads; Deep Embeddings; Haskell


## 1. Motivation

The use of *monads* to structure computation was first suggested by Moggi [27, 28] and Spivey [36], and was then enthusiastically taken up by Wadler [41, 42]. Monads have since proved to be a popular and frequently occurring structure, and are now one of the most prevalent abstractions within the Haskell language.

However, there are many data structures that are monad-like, but cannot be made instances of the Monad type class (Figure 1) because of type-class constraints on their operations. The classic example of this is the Set data type, which imposes an ordering constraint on its operations. This is unfortunate, because the Haskell language and libraries provide a significant amount of infrastructure to support arbitrary monads, including special syntax. We will refer to this situation where type-class constraints prevent a Monad instance from being declared as the "constrained-monad problem". This paper is about investigating solutions to this problem, with an emphasis on the particular solution of normalizing a deep embedding of a monadic computation. We aim to understand the utility of the various solutions, publicize their usefulness, and explain how they relate to each other.

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

- return $a$ >>= $k \equiv k\ a$ (left identity)
- $ma$ >>= return $\equiv ma$ (right identity)
- ($ma$ >>= $h$) >>= $k \equiv ma$ >>= ($\lambda a \to h\ a$ >>= $k$) (associativity)

**Figure 1.** Monads and the monad laws.

We begin by giving three concrete examples of the constrained-monad problem: three data types that would be monads were it not for the presence of class constraints on their operations.

### 1.1 Sets

The `Data.Set` module in the Haskell standard library provides an abstract representation of sets that is implemented using size-balanced binary trees [1] for efficiency. The operations from the module that we will use are as follows:

```
singleton :: a → Set a
toList    :: Set a → [a]
fromList  :: Ord a ⇒ [a] → Set a
unions    :: Ord a ⇒ [Set a] → Set a
```

Notice the Ord class constraint on two of the operations; this is a consequence of the binary-tree implementation.

Ideally we would like to define a Set monad that behaves analogously to the list monad (i.e. modelling non-determinism), except that it should combine duplicate results. For example, we would like to write the following set comprehension,

```
do n ← fromList [3, 2, 1, 2]
   c ← fromList ['a', 'b']
   return (n, c)
```

and be able to evaluate it to the set:

```
{(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b'), (3, 'a'), (3, 'b')}
```

Using the operations provided by `Data.Set`, it appears straightforward to define return and >>= (pronounced "bind") functions that satisfy the monad laws (Figure 1):

```
returnSet :: a → Set a
returnSet = singleton

bindSet :: Ord b ⇒ Set a → (a → Set b) → Set b
bindSet sa k = unions (map k (toList sa))
```

However, the use of unions introduces an Ord constraint in the type of bindSet, which means that a straightforward attempt to define a Monad instance will not type check:

```
instance Monad Set where
  return = returnSet
  (>>=)  = bindSet    -- does not type check
```

The problem is that ≫= must be parametrically polymorphic in its two type parameters, whereas bindSet constrains its second type parameter to ordered types.

### 1.2 Vectors

In their work on structuring quantum effects, Vizzotto et al. [39] discuss how quantum values can be represented as a vector associating a complex number to each element of a finite set. Furthermore, they observe that such vectors have a structure that is almost a monad, with the application of a linear operator to a vector matching the type of ≫= and obeying the monad laws. Adapting their code slightly, this can be defined as follows[1]:

```
class Finite (a :: *) where
  enumerate :: [a]

type Vec (a :: *) = (a → Complex Double)

returnVec :: Eq a ⇒ a → Vec a
returnVec a = λb → if a ≡ b then 1 else 0

bindVec :: Finite a ⇒ Vec a → (a → Vec b) → Vec b
bindVec va k = λb → sum [va a * (k a) b | a ← enumerate]
```

Notice the presence of class constraints on both returnVec and bindVec; as with the set example, these constraints prevent these functions being used to define a Monad instance. However, the situation is not quite the same as for sets: bindVec has a constraint on its first type parameter, whereas bindSet had a constraint on its second type parameter. Additionally, returnVec is constrained, whereas returnSet was not; and furthermore, the constraint on returnVec differs from the constraint on bindVec.

### 1.3 Embedding JavaScript

Our Sunroof library [6] provides an Embedded Domain-Specific Language (EDSL) [5, 10] for expressing JavaScript computations in Haskell, and for compiling them to executable JavaScript. Our initial embedding of a JavaScript computation in Haskell used the following Generalized Algebraic Data Type (GADT) [31]:

```
data JS :: * → * where
  Prompt :: JSString                → JS JSString
  Alert  :: JSString                → JS ()
  If     :: JSBool → JS a → JS a → JS a
```

We consider only a small selection of constructors for brevity: Prompt and Alert are deep embeddings of the standard-library functions of the same names; If is a deep embedding of a conditional statement. We have enumerated the functions Prompt and Alert for clarity here, but will later abstract over an arbitrary function (Section 4.3.2).

To allow JS to be compiled, we must constrain any polymorphic types to those that can be successfully translated to JavaScript — a common requirement for this style of EDSL [8, 37]. We represent this constraint with a class called Sunroof, which can generate new variables, print values in their JavaScript form, and, given a right-hand side, generate a JavaScript assignment:

```
type JSCode = String

class Show a ⇒ Sunroof (a :: *) where
  mkVar     :: String → a                -- New variable of type a
  showJS    :: a → JSCode                -- Show as JavaScript
  assignVar :: a → JSCode → JSCode       -- Assign to a variable
  assignVar a c = showJS a ++ "=" ++ c ++ ";"
```

The () type is the outlier, so we give its instance here. It is interesting because values of type () are not stored in variables, and expressions of type () are never even computed.

```
instance Sunroof () where
  mkVar _        = ()
  showJS ()      = "null"
  assignVar () _ = ""
```

To compile to JavaScript, we use the pattern of compiling JS into a pair consisting of JavaScript code and a JavaScript variable referring to the result of that code. We work inside a state monad [42] called CompM for name supply purposes.

```
compileJS :: Sunroof a ⇒ JS a → CompM (JSCode, a)
compileJS (Prompt s) = do
  (decl, v) ← newVar
  return (concat [decl
                 , assignVar v ("prompt(" ++ showJS s ++ ")")], v)
compileJS (Alert s) =
  return (concat ["alert(", showJS s, ");"], ())
compileJS (If b ja1 ja2) = do
  (decl, v) ← newVar
  (c1, a1)  ← compileJS ja1
  (c2, a2)  ← compileJS ja2
  return (concat [decl
                 , "if(", showJS b, ") {"
                 , c1, assignVar v (showJS a1)
                 , "} else {"
                 , c2, assignVar v (showJS a2)
                 , "}"], v)
```

As JavaScript computations are usually sequences of commands, with some commands returning values that can influence which commands follow, a natural way of expressing these sequences would be to use a monad. In particular, **do**-notation would allow for code that emulates JavaScript closely. For example:

```
js1 :: JS ()
js1 = do reply ← Prompt (string "Name?")
         Alert (string "Hello: " ⋄ reply)

string :: String → JSString   -- only signature given
```

For monadic code such as this to be valid, we need to write a Monad instance for JS. Note that we do not want the monadic operations to *interpret* the JavaScript (a shallow embedding); rather we want to construct a representation of the monadic operations (a deep embedding) so that we can later compile them to JavaScript. Thus we add Return and Bind constructors to the JS data type, using the *higher-order abstract syntax* [33] approach of representing a variable binding as a Haskell function:

```
data JS :: * → * where
  ...
  Return :: a                    → JS a
  Bind   :: JS x → (x → JS a) → JS a
```

Note that the semantics of GADTs are such that the $x$ on the Bind constructor is an *existential type* [22]: it is instantiated to a specific type when a Bind is constructed, and then inaccessibly encapsulated by the constructor until the Bind is deconstructed through case analysis.

Using Return and Bind, it is trivial to declare a Monad instance:

```
instance Monad JS where
  return = Return
  (≫=)  = Bind
```

The constrained-monad problem strikes when we try to compile this deep embedding into JavaScript. Initially, compiling the Return and Bind constructors may seem to be straightforward:

```
compileJS (Return a)  = return ("", a)
compileJS (Bind jx k) = do
  (c1, x) ← compileJS jx   -- does not type check
  (c2, a) ← compileJS (k x)
  return (c1 ++ c2, a)
```

[1] Actually, Vec should be defined as a **newtype** because of limitations on the use of type synonyms in Haskell. However, as this introduces some syntactic clutter, we elide this detail.

However, this does not type check. The problem is that the existential type introduced by the Bind constructor is not constrained by the Sunroof type class, and thus we cannot recursively call compileJS on the argument $jx$. There appears to be a simple solution though — add the Sunroof constraint to the Bind constructor:

$$\mathsf{Bind} :: \mathsf{Sunroof}\ x \Rightarrow \mathsf{JS}\ x \rightarrow (x \rightarrow \mathsf{JS}\ a) \rightarrow \mathsf{JS}\ a$$

However, because the type of Bind is now more constrained than the type of $\gg\!=$, the Monad instance for JS no longer type checks, leaving us in a similar situation to the vector example.

Yet while the same issue has arisen, this example is fundamentally different from the previous two examples. There we wanted to define return and $\gg\!=$ operations that would use their arguments to compute a new set/vector (i.e. a shallow embedding), whereas for Sunroof we are constructing a computation that we can then compile to JavaScript (i.e. a deep embedding).

### 1.4 Contributions

We have seen three examples of the constrained-monad problem. In each case, the problem was slightly different. The set example only required the second type parameter of $\gg\!=$ to be constrained, the vector example involved two distinct constraints, and in the Sunroof example the objective was to *compile* the computation to JavaScript, rather than to *evaluate* the computation. We will refer to the situation where we want to evaluate a monadic computation to the same type that we are operating on as the *shallow* constrained-monad problem, and to the situation where we want to compile a monadic computation as the *deep* constrained-monad problem (also known as the monad-reification problem).

The problem generalizes beyond monads. There are many other type classes with methods that are parametrically polymorphic in one or more arguments (e.g. Functor). We would like data types with operations that obey the relevant laws to be able to use the existing type-class hierarchy and infrastructure, even if those operations constrain the polymorphic arguments. As the problem is that we have a type class and a data type that are incompatible, a solution must therefore involve either modifying the class, modifying the data type, or both (or modifying the language).

This paper brings together several techniques for working with monads, constraints and deep embeddings, to present a framework for addressing the constrained-monad problem. In the process we demonstrate the compatibility of many of these techniques, and how they can be used to complement each other. The principal solution that we describe involves defining a normal form for a monadic computation, and a GADT to represent a deep embedding of that computation. The GADT explicitly constrains all existential types within the GADT, while still allowing a Monad instance to be declared for that GADT. The deep embedding can then be given multiple interpretations, which can impose different constraints. Finally, we show how the technique can be applied to other control structures, such as applicative functors.

We also survey other solutions to the constrained-monad problem, and related techniques. Some of the techniques we describe are well known, but have not been applied to this problem before. Others are known in functional-programming “folklore” to be able to address either the shallow or deep version of the problem, but have either not been documented, or generalized to the other half of the problem. There is one existing published solution to the deep constrained-monad problem [30], with which we compare in detail.

Our solution is direct, and we show how it can be applied to structures other than monads. Additionally, our solution provides a use case demonstrating the utility of the recent *constraint kinds* [3] GHC extension.

In summary, the contributions of this paper are as follows:

- We describe our solution to the constrained-monad problem. Specifically, we construct a normalized deep embedding of a monadic computation with explicit constraints on the existential types. The normalization eliminates any unconstrained types, allowing a Monad instance to be declared. We demonstrate our solution by applying it to each of the three examples. (Section 2)
- We present a general solution by abstracting from the solutions for each example. (Section 3)
- We survey other solutions to the constrained-monad problem, and compare them to each other and to our solution. (Section 4)
- We apply our solution to several other structures. (Section 5)
- We show how Jones et al. [17]'s technique for specifying constraints at use sites is compatible with our solution, and how it supports multiple interpretations of a computation. (Section 6)

## 2. Normality can be Constraining

In this section we present our technique for addressing the constrained-monad problem: constraining all existential types by normalizing a deep embedding of the monadic computation. For clarity, we present specialized solutions for each of the three examples.

### 2.1 Overview of Technique

The main steps of our technique are as follows:

- Separate the monadic structure of the computation from its primitive operations.
- Restructure the computation into a normal form by applying the monad laws.
- Capture that structure in a deep embedding that constrains all existential types.
- Only permit constrained primitive operations to be lifted into the embedding.
- Declare a Monad instance for the embedding.
- Constrain the type parameter of the computation when interpreting the computation.

By *monadic structure* we mean all uses of return and $\gg\!=$. By *primitive operations*, we mean other functions that produce values of type $m\ a$ for the specific monad $m$ with which we are working. The literature refers to primitive operations under a variety of names, including “non-proper morphisms”, “effect basis”, “instruction sets” and “generic effects”. For example, the primitive operations of the Maybe monad are the constructors Just and Nothing, whereas the primitive operations of the state monad [42] are get and put.

The monadic normal form [2] consists of a sequence of right-nested $\gg\!=$s terminating with a return, where the first argument to each $\gg\!=$ is a primitive operation. This can be seen graphically in Figure 2. Any monadic computation constructed from $\gg\!=$, return and primitive operations can be restructured into this normal form by applying the monad laws.

As well as allowing us to overcome the constrained-monad problem, there are two well-known benefits from constructing a normalized deep embedding. First, by applying the monad laws to normalize the structure, the monad laws are enforced *by construction* [24]. Second, by using a deep embedding, the construction of the monadic computation is separated from its interpretation. Consequently, multiple interpretations can be given to a computation [2, 24]. For example, a monadic computation over a probability distribution can be interpreted either non-deterministically by using a state monad where the state is a pseudo-random seed, or by computing the weights of all possible outcomes [2]. Furthermore, of particular relevance to the constrained-monad problem, different interpretations can impose different constraints.

In the remainder of this section, we demonstrate how normalization is the tool we need to constrain all existential types in a monadic computation, by applying it to our three examples.

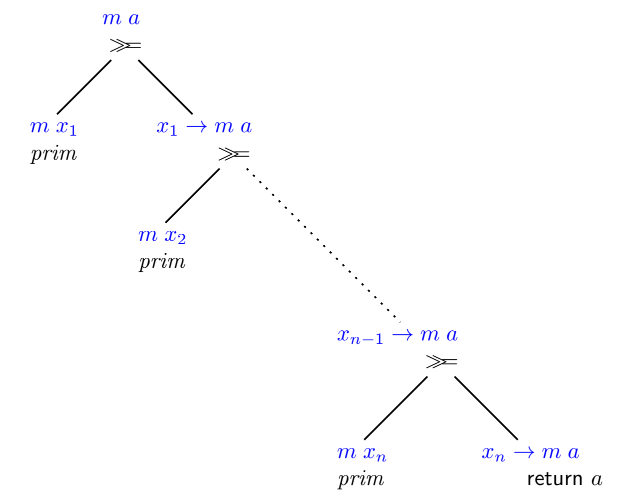


**Figure 2.** A normal form for monadic computations.

### 2.2 Sets

We begin by defining a deep embedding of normalized monadic computations over Set as a GADT:

```
data SetM :: * → * where
  Return :: a                       → SetM a
  Bind   :: Set x → (x → SetM a) → SetM a
```

Notice that the first argument to Bind is an actual Set (i.e. the type of primitive operations), not a recursive occurrence of SetM. We recommend that the reader takes a few moments to convince herself that this data type only permits monadic computations in the normal form presented in Figure 2.

We now declare a Monad instance for SetM, using the monad laws to restructure the computation into the normal form:

```
instance Monad SetM where
  return :: a → SetM a
  return = Return

  (>>=) :: SetM a → (a → SetM b) → SetM b
  (Return a)   >>= k = k a                        -- left identity
  (Bind sx h) >>= k = Bind sx (λx → h x >>= k)   -- associativity
```

The use of the monad laws is essential: attempting to define ≫= simply as Bind would not type check, because the first argument to ≫= is a SetM, whereas the first argument to Bind is a Set.

We now need a way to lift a primitive Set into this deep embedding. As there is no constructor for a solitary primitive operation, we use the right-identity law to introduce a Bind and Return:

```
liftSet :: Set a → SetM a
liftSet sa = Bind sa Return    -- right identity
```

If this does not seem obvious, consider that it is similar to creating a singleton list using $(\lambda x \to x : [\,])$.

We can now construct monadic computations over sets by working in the SetM monad. For example, we can express a set comprehension as follows:

```
s1 :: SetM (Int, Char)
s1 = do n ← liftSet (fromList [3, 2, 1, 2])
        c ← liftSet (fromList ['a', 'b'])
        return (n, c)
```

Finally, we need to lower a SetM to a Set. This is a simple matter of folding over the GADT, replacing Return and Bind with returnSet and bindSet (from Section 1.1), respectively:

```
lowerSet :: Ord a ⇒ SetM a → Set a
lowerSet (Return a)   = returnSet a
lowerSet (Bind sx k) = bindSet sx (lowerSet ∘ k)
```

Thus, to express monadic set computations, we can lift primitive Sets into the SetM type, write computations in the SetM monad, and then lower the resulting SetM to a Set. But why does this work? Observe from Figure 2 that, after normalization, the second type parameter of ≫= is the same type $(a)$ throughout the computation, and that this type is the same type as the top-level type parameter. Hence, by constraining the top-level type parameter in the type signature of lowerSet, we constrain the second type parameter of all occurrences of Bind. As the bindSet function only requires its second type parameter to be constrained, we are thus always able to use it in place of Bind. If we had not normalized the computation, then arbitrary existential types could appear as the second type parameter of Bind, and so we could not fold the computation using bindSet.

### 2.3 Vectors

Having solved the constrained-monad problem for sets, it would seem straightforward to do the same for vectors. And indeed we can define a normalized deep embedding (VecM), declare a monad instance for it, and provide a lifting function, exactly as we did for SetM. However, an attempt to define a similar lowering function fails to type check:

```
-- does not type check
lowerVec :: (Eq a, Finite a) ⇒ VecM a → Vec a
lowerVec (Return a)   = returnVec a
lowerVec (Bind vx k) = bindVec vx (lowerVec ∘ k)
```

The problem is that bindVec constrains its first type parameter (whereas bindSet constrains its second type parameter). That first parameter is not present in the type signature of lowerVec; indeed it is not even a single type: it needs to be instantiated to every existential type within the GADT (types $x_1$ to $x_n$ in Figure 2). There is no type signature that we can give to lowerVec to express this constraint.

However, we can change the deep embedding such that it constrains the existential types. In this case, we attach the Finite constraint to the Bind constructor:

```
data VecM :: * → * where
  Return :: a                                   → VecM a
  Bind   :: Finite x ⇒ Vec x → (x → VecM a) → VecM a
```

The definition of lowerVec now type checks, because the type of bindVec matches this type of Bind. Indeed, we can even drop the Finite constraint from the type of lowerVec,

```
lowerVec :: Eq a ⇒ VecM a → Vec a
```

as that constraint need only appear on the first type parameter of bindVec. Conversely, the Eq constraint must remain, as that is required by returnVec.

But does adding this constraint interfere with defining a Monad instance for VecM? Perhaps surprisingly, it does not; we can define the monad instance exactly as before:

```
instance Monad VecM where
  return :: a → VecM a
  return = Return

  (>>=) :: VecM a → (a → VecM b) → VecM b
  (Return a)   >>= k = k a                        -- left identity
  (Bind vx h) >>= k = Bind vx (λx → h x >>= k)   -- associativity
```

Initially this may appear magical: a function without class constraints is constructing a GADT containing class constraints. However, observe that (≫=) isn't introducing any new types; it is merely re-associating the existing computation. In the recursive case, it pattern matches on Bind, introducing the Finite constraint, before constructing a new Bind with the same existential type, and thus the necessary constraint is already in scope.

Of course, the Finite constraint has to come from somewhere, and that somewhere is the lifting function, which constructs a Bind without deconstructing an existing Bind:

```
liftVec :: Finite a ⇒ Vec a → VecM a
liftVec va = Bind va Return   -- right identity
```

This addition of a class constraint to the lifting function is key. Consider, the only ways to create a VecM are by using liftVec and Return. As liftVec is constrained, the only way to introduce an unconstrained type is through Return. But by performing normalization, all occurrences of Return except the final occurrence are eliminated. And the type of that final Return can be constrained by the lowering function. Hence, in combination, we can constrain all types within a VecM computation.

### 2.4 Embedding JavaScript

In the previous two examples we introduced a deep embedding as a means of normalizing a monadic computation, and then defined a lowering function that interpreted the deep embedding as the same type as the underlying primitive operations. The Sunroof example differs in that rather than interpreting the computation as the underlying type, we instead want to compile it to JavaScript.

We begin by splitting a JavaScript computation into two mutually recursive data types[2], one for the primitive operations and one for the (normalized) monadic structure:

```
data JS :: ∗ → ∗ where
  Prompt :: JSString                    → JS JSString
  Alert  :: JSString                    → JS ()
  If     :: JSBool → JSM a → JSM a → JS a

data JSM :: ∗ → ∗ where
  Return :: a                                  → JSM a
  Bind   :: Sunroof x ⇒ JS x → (x → JSM a) → JSM a
```

Notice that, as with VecM, we constrain the existential type.

A Monad instance for JSM, and an accompanying lifting function, are defined in exactly the same way as for VecM. We can then successfully complete the compiler for the JS monad:

```
compileJSM :: Sunroof a ⇒ JSM a → CompM (JSCode, a)
compileJSM (Return a)  = return ("", a)
compileJSM (Bind jx k) = do (c1, x) ← compileJS jx
                            (c2, a) ← compileJSM (k x)
                            return (c1 ++ c2, a)
```

We do not repeat the definition of compileJS as it is mostly unchanged from Section 1.3; the only difference is that the recursive calls to compileJS are replaced with calls to compileJSM.

### 2.5 Discussion

The key to this technique is *normalization*. Once normalized, the only types within a monadic computation are either the type parameters of primitive operations, or the top-level type parameter. Consequently, by constraining the primitives and constraining the top-level type parameter, we can constrain *all* types within the computation. Thus, when interpreting the computation, we can use functions that have class constraints.

As shown by the vector example, the class constraints on the ≫=-like and return-like functions can differ. In general, any constraints on the first type parameter of the ≫=-like function should appear in the context of the Bind constructor, constraining the existential type. Whereas any constraints on either the type parameter of the return-like function, or on the second type parameter of the ≫=-like function, should appear in the context of the interpretation function, constraining the type parameter of the computation.

[2] A non-recursive and more reusable alternative would be to paramaterize JS on a type $m :: * \to *$, and then instantiate that type to JSM. Such an approach is advocated by Lin [24].

Note that while the deep embedding is used to *facilitate* normalization, it is not *necessary*: there are other ways to normalize a monadic computation, as we will discuss in Section 4.4. Furthermore, even using a deep embedding, defining a GADT that *enforces* the normal form is unnecessary. A valid alternative is to define a GADT that allows arbitrary nesting of Binds and occurrences of Returns (such as the JS data type in Section 1.3). We would then normalize that GADT, or normalize during deconstruction of the GADT (as is done by Unimo [24] and RMonad [35]). However, we consider it clearer, and less error-prone, to have the GADT enforce the normal form (as is done by Operational [2]).

## 3. Generalizing using Constraint Kinds

Each deep embedding in Section 2 was specialized to a particular example. In this section we use the recent *constraint kinds* GHC extension to generalize to a single deep embedding that can be instantiated to any of the examples. This generalized solution is also available in our Constrained-Normal library [34].

### 3.1 Constraint Kinds

Constraint kinds were implemented by Bolingbroke [3], motivated by a desire to support the constraint synonyms and constraint families proposed by Orchard and Schrijvers [29]. The core idea is to add a new *literal kind* to Haskell called Constraint, which is the kind of a fully applied type class. For example, the Ord and Monad classes would be assigned the following kinds:

```
Ord   :: ∗ → Constraint
Monad :: (∗ → ∗) → Constraint
```

The most significant benefit of this extension is the ability to abstract over constraints: data types and type classes can take class constraints as parameters.

The extension also adds syntax for an empty constraint and constraint synonyms; however, as constraint synonyms cannot be partially applied, they are less expressive than defining empty type classes with polymorphic instances. For example, we will later need to instantiate a parameter of kind (∗ → Constraint) such that it imposes no constraints on the type; so we encode this using the following empty type class:

```
class Unconstrained (a :: ∗) where
instance Unconstrained a where
```

### 3.2 Constrained Normal Monads

We now define what we call a *constrained normal monad*: a GADT representing a normalized monadic computation with constrained existential types. This requires parameterizing the GADT on a class constraint $c$ and an underlying type of primitive operations $t$:

```
data NM :: (∗ → Constraint) → (∗ → ∗) → ∗ → ∗ where
  Return :: a                              → NM c t a
  Bind   :: c x ⇒ t x → (x → NM c t a) → NM c t a
```

This GADT generalizes SetM, VecM and JSM from Section 2, and has a Monad instance defined in the same way.

To accompany NM, we provide a generalized lifting function:

```
liftNM :: c a ⇒ t a → NM c t a
liftNM ta = Bind ta Return
```

Constructing monadic computations is now no harder than when using the specialized solutions in Section 2, for example:

```
s1 :: NM Unconstrained Set (Int, Char)
s1 = do n ← liftNM (fromList [3, 2, 1, 2])
        c ← liftNM (fromList ['a', 'b'])
        return (n, c)
```

A generalized lowering function is slightly more complicated. Recall that lowerSet and lowerVec made use of operations specific to their underlying type, such as returnVec and bindVec. Thus, our generalization has to take those functions as arguments:

```
lowerNM :: ∀ a c t. (a → t a) →
  (∀ x. c x ⇒ t x → (x → t a) → t a) → NM c t a → t a
lowerNM ret bind = lowerNM′
  where lowerNM′ :: NM c t a → t a
        lowerNM′ (Return a)  = ret a
        lowerNM′ (Bind tx k) = bind tx (lowerNM′ ∘ k)
```

The type signatures used here require the *scoped type variables* [31] and *rank-2 types* [32] GHC extensions. Notice that the type of *ret* shares its type parameter ($a$) with the parameter of the NM argument. Thus *ret* need only be applicable at that one specific type, which is sufficient because normalization ensures that there will only be one Return constructor, and it will have that type. The second type parameter of *bind* is that same type $a$, but its first type parameter can be *any type that type satisfies the constraint* $c$. This restriction is precisely what allows lowerNM to take constrained functions such as bindVec as arguments. For example:

```
lowerVec :: Eq a ⇒ NM Finite Vec a → Vec a
lowerVec = lowerNM returnVec bindVec

lowerSet :: Ord a ⇒ NM Unconstrained Set a → Set a
lowerSet = lowerNM returnSet bindSet
```

Actually, the inferred type of lowerNM is more general than the type signature we assigned it. Its definition is a fold over the GADT, which could return values of types other than the primitive operation. Renaming lowerNM to foldNM, the inferred type is:

```
foldNM :: ∀ a c r t. (a → r) →
  (∀ x. c x ⇒ t x → (x → r) → r) → NM c t a → r
```

This generalization is useful because it can be used to define interpretations of a monadic computation that differ from the underlying type. For example, we could define compileJSM in terms of foldNM, but not in terms of lowerNM.

A problem arises when we try to use multiple interpretations that impose distinct constraints. Consider the type of the following pretty printer for set computations (its definition is straightforward, but unimportant to the discussion):

```
prettySet :: Show a ⇒ NM Show Set a → String
```

If we try to apply prettySet to $s_1$ (from Section 3.2), type checking will fail because Show does not match Unconstrained. We can work around this by assigning $s_1$ a more polymorphic type, thereby allowing it to be interpreted by both lowerSet and prettySet:

```
s1 :: (c Char, c Int) ⇒ NM c Set (Int, Char)
```

However, while this works in this case, there are limitations to this approach. We postpone demonstrating these limitations, and describing an approach to overcoming them, until Section 6.

# 4. A Survey of Solutions

The constrained-monad problem is well known, and a variety of techniques have been implemented or proposed to address it. In this section we survey the existing solutions, and compare them to each other and the normalized deep-embedding approach.

## 4.1 Restricted Data Types

An early solution to the constrained-monad problem was Hughes' *restricted data types* [12] proposed language extension. The essence of the idea was to extend Haskell to support data types with attached constraints that scope over the entire type. Hughes's proposed syntax is best seen by example:

```
data Ord a ⇒ RSet a = ...
```

Note however that this is intended to be semantically stronger than the existing meaning of that same syntax in Haskell (which has since been deprecated). Furthermore, the proposal included introducing a keyword **wft** (well-formed type) that would allow the constraints of a data type to be referenced. For example, a context

```
wft (RSet a) ⇒ ...
```

would be semantically equivalent to

```
Ord a ⇒ ...
```

Type classes could then refer to these attached constraints in their methods. For example:

```
class Monad (m :: * → *) where
  return :: wft (m a) ⇒ a → m a
  (>>=)  :: (wft (m a), wft (m b)) ⇒ m a → (a → m b) → m b
```

Instances could then be declared for restricted data types such as RSet, as any attached constraints are brought into scope.

Note that the keyword **wft** is crucial to this approach. Without the keyword, a restricted data type is analogous to a GADT with the constraint on each constructor, except that it is necessary to pattern match on the constructor to bring the constraint into scope. However, this GADT approach is insufficiently constraining: for example, an instance for sets would effectively require return to map *any* type $a$ to a pair of Set $a$ and an Ord $a$ class dictionary, which is impossible to define. The **wft** keyword is needed to allow the Ord constraint to scope over the argument type $a$.

## 4.2 Restricted Type Classes

Hughes [12] also suggested an alternative approach: defining *restricted type classes* that take a class constraint as a parameter. For example, a *restricted monad* could be defined as follows:

```
class RMonad (c :: * → Constraint) (m :: * → *) where
  return :: c a ⇒ a → m a
  (>>=)  :: (c a, c b) ⇒ m a → (a → m b) → m b
```

This was prior to the advent of constraint kinds so was not possible at the time, but several simulations of restricted type classes were encoded using various work-arounds [12, 18, 19, 35].

An alternative formulation of restricted type classes uses an associated type function [4] to map the data type to a constraint, rather than taking a constraint as a parameter [3, 29]. Such a function can be given a default definition as Unconstrained, which is convenient when declaring instances for data types that do not require a constraint on their operations. For example:

```
class RMonad (m :: * → *) where
  type Con m :: * → Constraint
  type Con m = Unconstrained

  return :: Con m a ⇒ a → m a
  (>>=)  :: (Con m a, Con m b) ⇒ m a → (a → m b) → m b

instance RMonad Set where
  type Con Set = Ord

  return = returnSet
  (>>=)  = bindSet

instance RMonad [] where
  return a   = [a]
  ma >>= k = concatMap k ma
```

The restricted-type-class solutions do not require any modification to the data types involved, but they do require either the existing type classes to be modified, or new types classes to be added. In the former case existing code will break, in the latter case the new code will not be compatible with the existing, unrestricted, classes. Thus, for this to be practical, a class author must anticipate the need for constraints when defining the class [12].

## 4.3 Normalizing using Deep Embeddings

The idea of separating the monadic structure of a computation from its interpretation was pioneered by the Unimo framework [24], and then later used by the MonadPrompt [15] and Operational [2] libraries. The same idea was used by Swierstra [38] to embed IO computations, albeit formulated somewhat differently using free monads (see Section 5.2). Normalization is not an essential part of separating structure from interpretation, but it does have the advantage of enforcing the monad laws: without normalization it is possible to define an interpretation that exhibits different behavior for computations that are equivalent according to the monad laws.

The first use of normalization to overcome the constrained-monad problem of which we are aware was in the RMonad library [35]. The central feature of this library is a restricted-monad class, RMonad. This class is similar to the restricted monads described in Section 4.2, except that it is implemented using a data family [20], with the constructors of the family containing the constraints. We believe this implementation choice was made because constraint kinds were not then available; the implementation could now be simplified using the associated-type-function approach. The RMonad library also provides a (non-normalized) deep embedding over the RMonad class, with an accompanying Monad instance. The library provides a function to interpret that deep embedding, which normalizes the structure and lowers it to the underlying restricted monad. This is essentially the same technique as presented in Section 3, using normalization to ensure that the necessary constraints hold whenever the (restricted) $\gg\!=$ of the restricted monad is applied. The differences are that an RMonad instance is required, and that the only interpretation of the embedding is as that underlying RMonad.

### 4.3.1 Using the Operational Library

While Unimo, MonadPrompt and Operational do not explicitly handle constraints, it is possible to leverage their deep embedding and normalization functionality when addressing the constrained-monad problem. We will demonstrate this by encoding the technique from Section 3 using Operational, and we note that Unimo and MonadPrompt could use a similar encoding.

The core of Operational is a deep embedding of a normalized monadic computation, which is essentially our NM data type without the constraint parameter[3]:

```
data Program :: (* → *) → * → * where
  Return :: a                            → Program t a
  Bind   :: t x → (x → Program t a) → Program t a
```

For Program to be able to handle constraints, we have to embed the desired constraint into an underlying type. This can be achieved using a GADT:

```
data FinVec :: * → * where
  FV :: Finite a ⇒ Vec a → FinVec a
```

When defining an interpretation, we can pattern match on the GADT to bring the constraint into scope, for example (where lowerProg is defined similarly to lowerNM):

```
lowerVec :: Eq a ⇒ Program FinVec a → Vec a
lowerVec = lowerProg returnVec (λ(FV vx) k → bindVec vx k)
```

This is okay if the user always wants to work with specific pairs of primitive operations and constraints, but does not allow for situations where the user wants to write code that treats one as polymorphic and the other as specialized. However, this can be addressed by abstracting over the constraint and primitive operations:

```
data Box :: (* → Constraint) → (* → *) → * → * where
  Box :: c a ⇒ t a → Box c t a

lowerVec :: Eq a ⇒ Program (Box Finite Vec) a → Vec a
lowerVec = lowerProg returnVec (λ(Box vx) k → bindVec vx k)
```

[3] We simplify slightly, as the Operational implementation uses monad transformers and views, but our encoding is valid using the actual library.

We could now, for example, assign the following polymorphic type to the $s_1$ computation:

```
s1 :: (c Char, c Int) ⇒ Program (Box c Set) (Int, Char)
```

### 4.3.2 Constraining the Primitive Operations

If the type of primitive operations is a GADT, then as an alternative to using the Box type from Section 4.3.1, the desired constraint can instead be placed within *each constructor* of the GADT that contains polymorphic types, for example:

```
type JSM (a :: *) = Program JS a
data JS :: * → * where
  If :: Sunroof a ⇒ JSBool → JSM a → JSM a → JS a
  ...
```

Instead of pattern matching on the Box GADT, we can now introduce the constraint by performing a case analysis on the primitive operation. The disadvantages of this approach are that primitive operations must not be an abstract data type, that syntactic clutter is introduced in the form of repeated constraints, and that additional case analyses may sometimes be required. However, a significant advantage is that it becomes possible to constrain existential types that occur within the primitive-operation GADT, or to have different constraints on different constructors. For example, the Prompt and Alert constructors can be generalized to a single Call constructor that is polymorphic in its argument and result type, provided those types have Sunroof instances:

```
data JS :: * → * where
  Call :: (Sunroof a, Sunroof b) ⇒ JSFunction a b → a → JS b
  ...
```

Indeed, this is how Sunroof is actually implemented, using Operational for the monadic deep embedding and normalization [6].

## 4.4 Normalizing using Continuations

Unimo, Operational, RMonad, and our constrained normal monads are all similar in that they normalize the structure of a monadic computation by capturing that structure as a deep embedding. However, an alternative means of normalizing a monadic computation is to use *continuations*.

Consider the following types:

```
type ContT (r :: *) (t :: * → *) (a :: *) =       (a → t r) → t r
type CodT           (t :: * → *) (a :: *) = ∀ r. (a → t r) → t r
```

These are known as the *continuation monad transformer* [23] and the *codensity monad transformer* [14, 16]. Note however that both ContT $r$ $t$ and CodT $t$ form a monad, regardless of whether the underlying type $t$ is a monad[4]:

```
instance Monad (CodT t) where
  return :: a → CodT t a
  return a = λh → h a
  (>>=) :: CodT t a → (a → CodT t b) → CodT t b
  ca >>= k = λh → ca (λa → (k a) h)
```

A Monad instance for ContT is declared in the same way: ContT is just a special case of CodT that fixes the result type $r$.

Primitive operations can be lifted into (or lowered from) the codensity monad by providing a $\gg\!=$-like or return-like function, respectively:

```
liftCodT :: (∀ r. t a → (a → t r) → t r) → t a → CodT t a
liftCodT bind ta = bind ta

lowerCodT :: (a → t a) → CodT t a → t a
lowerCodT ret ca = ca ret
```

[4] As with Vec, we elide the detail that ContT and CodT must be **newtype**s for this to be valid Haskell.

A consequence of these definitions is that any monadic computation constructed in the CodT monad will construct a normalized computation with respect to the underlying ≫=-like and return-like functions. A useful analogy for what happens, suggested by Voigtländer [40], is that it is like using *difference lists* [11] to right-associate nested applications of string concatenation. Alternatively, observe that each primitive operation always appears as the first argument to the underlying ≫=-like function, as that function is partially applied to the primitive during lifting. And there is always exactly one use of the return-like function, which is when it is used as the final continuation during lowering.

This infrastructure can be used to address the constrained-monad problem, for example:

```
liftVec :: Finite a ⇒ Vec a → CodT Vec a
liftVec = liftCodT bindVec

lowerVec :: Eq a ⇒ CodT Vec a → Vec a
lowerVec = lowerCodT returnVec
```

During lifting the first type parameter of bindVec is exposed, so it can be constrained with Finite. During lowering, the type parameter of returnVec is exposed, so it can be constrained with Eq.

However, what is not exposed is the second type parameter to bindVec, which is the universally quantified $r$ hidden inside CodT. For the vector example that is not a problem, but if we try to lift a Set in a similar manner then the definition will not type check because $r$ is required to satisfy an Ord constraint. Perhaps then we should use ContT instead of CodT, as that exposes a fixed result type that we could constrain? While that would work, it would be unnecessarily restrictive. We don't need $r$ to be a specific type, merely for it to satisfy Ord. Therefore, we define what we shall call the *restricted codensity transformer*:

```
type RCodT (c :: ∗ → Constraint) (t :: ∗ → ∗) (a :: ∗)
  = ∀ r. c r ⇒ (a → t r) → t r
```

This type expresses exactly what we need: the result type can be any type that satisfies $c$. This gives rise to lifting and lowering functions with the following types:

```
liftRCodT  :: (∀ r. c r ⇒ t a → (a → t r) → t r) →
              t a → RCodT c t a
lowerRCodT :: c a ⇒ (a → t a) → RCodT c t a → t a
```

Thus, the shallow constrained-monad problem can be overcome using the restricted codensity transformer with comparable ease to using a normalized deep embedding. However, the two approaches are not equivalent, as becomes evident when we consider interpretations other than lowering to the underlying type. The deep-embedding approach allows multiple interpretations to be given to a monadic computation, whereas the codensity approach only allows a single interpretation. That single interpretation is essentially "hard-wired" into the computation, as the ≫=-like operation is partially applied to each primitive operation when it is lifted.

There is a work-around though. Rather than using the primitive operations directly, we can first construct a deep embedding of a monadic computation over the primitive-operation type, and then use that deep embedding as the underlying type of the restricted codensity monad. Regardless of how return and ≫= are then used, the result after lowering will be a normalized deep embedding. That deep embedding can then be interpreted in whatever way is desired. This is essentially the approach taken by Persson et al. [30] to overcome the monad-reification problem in the Syntactic library, though they build their own specialized type around the continuation monad, rather than defining the general-purpose restricted codensity transformer as we have here.

In our opinion, using the codensity technique for the deep constrained-monad problem in this manner is more complicated than using the normalized-deep-embedding approach, as it requires two levels of structure rather than one: the codensity transformer to perform the normalization, and then the deep embedding to allow multiple interpretations. Whereas a normalized deep embedding provides both with a single structure. Finally, we observe that this two-level approach could be employed with any solution to the constrained-monad problem that only permits one interpretation, such as the RMonad library.

### 4.5 Normalization and Efficiency

That continuations can be used to overcome the constrained-monad problem has been an obscure piece of functional-programming "folklore" for several years [26], but, to our knowledge, the only published use of the technique was when Persson et al. [30] used it in Syntactic. There have been other uses of continuations to normalize monadic computations, but with the aim of improving efficiency. For example, Voigtländer [40] uses the codensity transformer to improve the efficiency of the tree-substitution monad. This is possible because the monad laws only guarantee the equivalence of the *semantics* of two monadic computations; the laws do not guarantee the equivalence of operational behavior.

However, normalization is not always beneficial to performance. For example, normalizing a set-monad computation defers elimination of duplicate elements until the end, rather than eliminating duplicates in intermediate results [35]. That is, the performance of a normalized set monad will typically be no better than converting to a list, using the list monad, and then converting back to a set again. Note that this change in operational behavior is a consequence of the normalization, and applies regardless of whether a deep embedding or the codensity transformer is used to achieve that normalization. Conversely, using a restricted monad does not cause a change in operational behavior, as no normalization of structure occurs (but nor are the monad laws enforced).

## 5. The Constrained-Type-Class Problem

The focus of this paper has been monads; a choice we made because of the widespread use of monads in functional programming. However, the essence of the problem — that some data types cannot be made instances of some type classes because of the presence of class constraints on their operations — is not specific to monads, nor are the techniques for overcoming it. In this section we demonstrate that the solutions to the constrained-monad problem are more widely applicable, by applying them to several related type classes: Functor, Applicative and MonadPlus.

### 5.1 The Constrained-Functor Problem

Consider the Functor type class (Figure 3). As with the Monad class methods, sometimes the only mapping function that exists for a data type imposes constraints on its type parameters, thereby preventing a Functor instance from being declared. For example:

```
mapSet :: Ord b              ⇒ (a → b) → Set a → Set b
mapVec :: (Finite a, Eq b)   ⇒ (a → b) → Vec a → Vec b
```

We have the same options for addressing this problem as we had for monads. Thus, if we are prepared to use a new type class, then we could define a *restricted functor* class:

```
class RFunctor (c :: ∗ → Constraint) (f :: ∗ → ∗) where
  fmap :: (c a, c b) ⇒ (a → b) → f a → f b
```

We could also use the variant that has an associated type function that maps to a constraint, rather than a constraint parameter.

On the other hand, if we want to use the standard Functor class, then we can take the normalization approach. The normal form for functors is fairly simple: a single fmap applied to a single primitive operation. This ensures that all existential types within the computation appear as parameters on the (single) primitive operation.

```
class Functor (f :: * → *) where
  fmap :: (a → b) → f a → f b
```

- fmap id ≡ id (identity)
- fmap g ∘ fmap h ≡ fmap (g ∘ h) (composition)

**Figure 3.** Functors and the functor laws.

Normalization consists of applying the functor composition law to fuse together all uses of fmap.

Taking the deep-embedding approach to normalization, we can define *constrained normal functors* as follows:

```
data NF :: (* → Constraint) → (* → *) → * → * where
  FMap :: c x ⇒ (x → a) → t x → NF c t a

instance Functor (NF c t) where
  fmap :: (a → b) → NF c t a → NF c t b
  fmap g (FMap h tx) = FMap (g ∘ h) tx   -- composition law

liftNF :: c a ⇒ t a → NF c t a
liftNF ta = FMap id ta   -- identity law

lowerNF :: (∀ x. c x ⇒ (x → a) → t x → t a) → NF c t a → t a
lowerNF fmp (FMap g tx) = fmp g tx
```

Notice the similarities to constrained normal monads (Section 3.2): we define the normal form as a GADT, declare a Functor instance by using laws to convert to that normal form, define a lifting function by using an identity law, and define a lowering function that takes interpretations of the constructors as arguments.

We can also take the codensity approach to normalization, but instead of the codensity transformer, we need to use the related *Yoneda functor transformer* [21], which generates a functor (but not a monad) for any $t :: * \to *$:

```
type Yoneda (t :: * → *) (a :: *) = ∀ r. (a → r) → t r

instance Functor (Yoneda t) where
  fmap :: (a → b) → Yoneda t a → Yoneda t b
  fmap g ya = λh → ya (h ∘ g)
```

As with codensity, we introduce a restricted version of this transformer by adding a constraint parameter:

```
type RYoneda (c :: * → Constraint) (t :: * → *) (a :: *)
  = ∀ r. c r ⇒ (a → r) → t r
```

We can then define lifting and lowering functions as follows:

```
liftRYoneda :: (∀ r. c r ⇒ (a → r) → t a → t r) →
               t a → RYoneda c t a
liftRYoneda fmp ta = λh → fmp h ta

lowerRYoneda :: c a ⇒ RYoneda c t a → t a
lowerRYoneda ya = ya id
```

Notice that liftRYoneda takes an fmap-like function as an argument — this means that the interpretation of fmap is "hard-wired" during construction in a similar manner to the interpretation of ≫= when using the codensity transformer for monads.

### 5.2 Aside: Free Monads

Every functor induces a monad, known as the *free monad* [38, 40] of that functor:

```
data Free (f :: * → *) (a :: *) = Pure a | Impure (f (Free f a))

instance Functor f ⇒ Monad (Free f) where
  return :: a → Free f a
  return = Pure

  (≫=) :: Free f a → (a → Free f b) → Free f b
  (Pure a)     ≫= k = k a
  (Impure ffa) ≫= k = Impure (fmap (≫=k) ffa)
```

The data type Free is a deep embedding of a monadic computation, and thus can be given multiple interpretations [38]. Yet this embedding is not as deep as the NM embedding, as it always uses the fmap of the underlying functor, rather than allowing it to be given multiple interpretations. We find it helpful to think of this as constructing a monad from fmap, return and join, where fmap is shallowly embedded and return and join are deeply embedded.

Aside from the lack of constraint parameter, the key distinction between the two embeddings is that, to induce a monad, Free requires the underlying type to be a Functor, whereas NM does not. However, we showed in Section 5.1 that it is possible to generate a functor for any $t{::}* \to *$, by using a deep embedding. This provides an alternative way to construct a deep embedding of a monadic computation: take the free monad over the deep embedding of a functor. Doing so results in a type isomorphic to a deep embedding of a normalized monad. This can be seen informally as follows:

```
Free (NF c t) a      ≈ NM c t a
Pure a               ≈ Return a
Impure (FMap k tx)   ≈ Bind tx k
```

```
class Applicative (f :: * → *) where
  pure :: a → f a
  (⊛)  :: f (a → b) → f a → f b
```

- pure id ⊛ fa ≡ fa (identity)
- ((pure (∘) ⊛ fa) ⊛ fb) ⊛ fc ≡ fa ⊛ (fb ⊛ fc) (composition)
- pure g ⊛ pure a ≡ pure (g a) (homomorphism)
- fg ⊛ pure a ≡ pure (λg → g a) ⊛ fg (interchange)

**Figure 4.** Applicative functors and the applicative-functor laws.

### 5.3 The Constrained-Applicative-Functor Problem

Applicative functors [25] (Figure 4) are a structure that lies between functors and monads, and the usual problem arises if we want to declare an instance for a data type with constrained operations. Defining a class of *restricted applicative functors* is as straightforward as defining restricted monads or restricted functors, so we will just discuss the normalization approach.

The normal form for applicative functors [7] consists of a left-nested sequence of ⊛s (pronounced "apply") terminating in a pure (Figure 5). Our deep embedding of this normal form is as follows:

```
data NAF :: (* → Constraint) → (* → *) → * → * where
  Pure :: a                                → NAF c t a
  Ap   :: c x ⇒ NAF c t (x → a) → t x → NAF c t a
```

Defining an Applicative instance and lifting function require applying the laws of the structure (Figure 4), as usual:

```
instance Applicative (NAF c t) where
  pure :: a → NAF c t a
  pure = Pure

  (⊛) :: NAF c t (a → b) → NAF c t a → NAF c t b
  (Pure g) ⊛ (Pure a) = Pure (g a)                  -- homomorphism
  n1 ⊛ (Pure a)      = Pure (λg → g a) ⊛ n1         -- interchange
  n1 ⊛ (Ap n2 tx)    = Ap (Pure (∘) ⊛ n1 ⊛ n2) tx   -- composition

liftNAF :: c a ⇒ t a → NAF c t a
liftNAF ta = Ap (Pure id) ta   -- identity
```

While in many ways similar, there is an important difference between the normal forms for monads and applicative functors. For monads, the top-level type parameter is propagated along the spine of the normal form, giving the same result type at each node (type $a$ in Figure 2). Whereas for applicative functors the type of each node differs, as the function type is progressively saturated with type arguments (see Figure 5). Furthermore, the type of Pure differs from the top-level type parameter. Consequently, an interpretation of applicative functors has to be able to handle these existential types. This is not an obstacle, but does lead to heavy use of rank-2 types when assigning a type to those interpretations. For example:

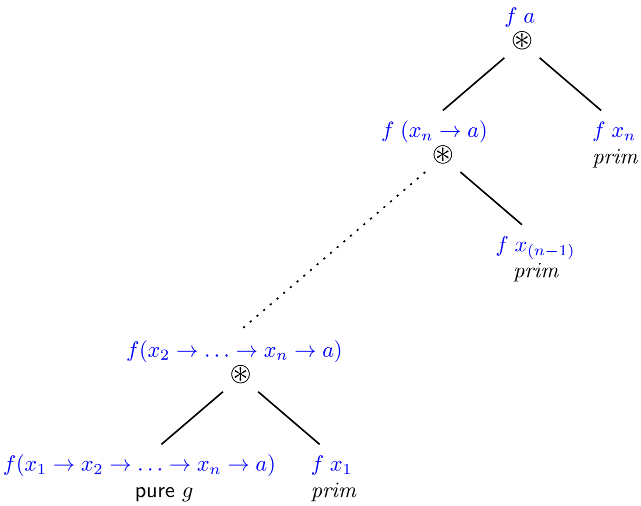


**Figure 5.** A normal form for applicative computations.

```
foldNAF :: ∀ a c r t. (∀ x. x → r x) →
    (∀ y z. c y ⇒ r (y → z) → t y → r z) → NAF c t a → r a
foldNAF pur app = foldNAF′
  where
    foldNAF′ :: ∀ b. NAF c t b → r b
    foldNAF′ (Pure b)   = pur b
    foldNAF′ (Ap n tx) = app (foldNAF′ n) tx
```

The reader may now expect us to define a transformer similar to Codensity and Yoneda, but that generates an applicative functor. However, we were unable to find such a transformer, as we will now explain.

Initially we did not know what type the transformer should have, so we started with the lifting function. We wanted the lifting function to partially apply ⊛, such that the primitive appears as its second argument, which gave us the following type:

```
liftAFT :: (∀ r. t (a → r) → t a → t r) → t a → AFT t a
liftAFT app ta = λtg → app tg ta
```

From this, we inferred a type for the transformer:

```
type AFT t a = ∀ r. t (a → r) → t r
```

It was then straightforward to define a lowering function:

```
lowerAFT :: (∀ x. x → t x) → AFT t a → t a
lowerAFT pur ra = ra (pur id)
```

However, we were unable to define an Applicative instance for AFT $t$, and have come to the conclusion that it is only possible to define such an instance *if the underlying type $t$ is itself an Applicative*. As the problem we are trying to overcome is that the underlying type is *not* an Applicative, this is no use to us.

We do not claim that there is no transformer type that could solve this problem for applicative functors. Rather, we observe that we found it difficult to apply the Codensity/Yoneda approach to other structures, because it is not obvious what the transformer type should be. Arguably, this problem requires no more inventiveness than deciding on a normal form for the structure (which is needed for the deep-embedding approach). However, our experience has been that it is easier to devise a normal form than to devise a suitable transformer type.

### 5.4 The Constrained-MonadPlus Problem

Now consider the MonadPlus type class (Figure 6). Defining a restricted version of this type class is straightforward as usual, so we will just discuss the normalization approach. We represent the MonadPlus normal form using two mutually recursive data types:

```
class Monad m ⇒ MonadPlus (m :: ∗ → ∗) where
  mzero :: m a
  mplus :: m a → m a → m a
```

- mplus mzero $a \equiv a$ (left unit)
- mplus $a$ mzero $\equiv a$ (right unit)
- mplus (mplus $a$ $b$) $c \equiv$ mplus $a$ (mplus $b$ $c$) (associativity)
- mzero $\gg\!=$ $k \equiv$ mzero (left zero)
- mplus $a$ $b \gg\!= k \equiv$ mplus $(a \gg\!= k)$ $(b \gg\!= k)$ (left distribution)

**Figure 6.** MonadPlus and the MonadPlus laws.

```
type NMP c t a = MZero | MPlus (NMP′ c t a) (NMP c t a)
data NMP′ :: (∗ → Constraint) → (∗ → ∗) → ∗ → ∗ where
  Return :: a                                    → NMP′ c t a
  Bind   :: c x ⇒ t x → (x → NMP c t a) → NMP′ c t a
```

The first type, NMP, represents a normal form for the monoidal mplus/mzero operations. That normal form is just the free monoid (i.e. a list) over the second type, NMP′. The normal form for NMP′ is the usual monadic normal form, except that the result of the second argument to Bind is the NMP type, rather than NMP′.

Defining a MonadPlus instance for NMP proceeds in a similar manner to the monad and applicative functor cases, so we direct the reader to our Constrained-Normal library [34] for the details. The key step is the application of the left-distribution law (Figure 6) when an MPlus appears as the first argument to ≫=. Likewise, the accompanying lifting function involves applying the right-identity law (and the right-unit law), and the folding function takes interpretations for the four constructors as arguments.

### 5.5 Discussion

The same idea underlies the normalization technique for all of the structures we have considered, and for each structure we performed the same sequence of steps. First, we identify a normal form that contains no existential types except those that appear on primitive operations. Second, we define a deep embedding of that normal form as a GADT. The GADT takes a class constraint as a parameter, and places that constraint on any existential types within the GADT. Third, we declare the structure's class instance for the GADT; this instance normalizes the computation by applying the algebraic laws of the structure, which typically involves fusing pure computations and thereby eliminating intermediate existential types. Fourth, we define a function to lift a primitive operation into the normal form, which (for the structures we have considered) involved applying one or more identity laws, and required the type parameter of the primitive operation to satisfy the class constraint. Finally, we define a fold for the computation, which takes interpretations for the operations of the structure as arguments.

In sections 4.3.1 and 4.3.2 we discussed variant formulations of a deep embedding for monads that place the class constraint either on the constructors of the primitive operations, or in a Box GADT that can be used as a wrapper around the primitive operations. Both of those techniques generalize to the other structures.

The structures we considered in this section are all specializations or generalizations of monads. That is not a limitation of the technique; these are just well-known structures that we chose as a starting point. We are also investigating the Category and Arrow [13] type classes. Our initial results show that Category is straightforward to normalize, as it has a monoidal structure, but that there may not exist an Arrow normal form that ensures that all existential types appear as type parameters on primitive operations. However, an intermediate structure consisting of a Category with an arr operation (but not a first) does have a suitable normal form: a sequence of alternating pure functions and primitives. We are also interested in investigating recursive structures such as MonadFix and ArrowLoop, but this remains as future work.

Note that for some structures, full normalization is not required to eliminate all unconstrained existential types (and hence to allow the desired class instance). For example, normalizing the *monoidal* structure of the MonadPlus class is unnecessary to infer any constraints, as the mplus and mzero operations do not introduce any existential types; normalizing the *monadic* structure is sufficient. Indeed, the RMonad library [35] takes the approach of only normalizing the monadic structure, not the monoidal structure. However, an advantage of fully normalizing is that the monoid laws are enforced by construction, which is why we chose to do so.

## 6. Interpretations with Distinct Constraints

In Section 3.2 we demonstrated that it is possible to define multiple interpretations for a constrained normal monad (lowerSet and prettySet), even if those interpretations impose different constraints. However, that approach does not allow both interpretations to be applied to the same computation *if they impose distinct constraints*. Even if the computation is defined to be polymorphic in its constraint parameter, two copies of the computation have to be constructed, one for each interpretation. This is a more general problem that afflicts any data type that has a constraint parameter, but fortunately Jones et al. [17] have recently developed a technique for addressing it. In this section we demonstrate the problem, and then show how Jones et al.'s technique can be applied to overcome it. While we only consider our constrained normal monads, we note that this technique is also compatible with using a Box wrapper and the Operational library (Section 4.3.1), with placing the constraints on the constructors of the primitive operations (Section 4.3.2), and with the other type classes discussed in Section 5.

### 6.1 Distinct Constraint Parameters are Incompatible

Imagine that we wish to display all elements of all Finite types that appear within a vector computation. We could achieve this by defining a function of the following type:

```
showFin :: NM FiniteShow Vec a → String
class (Finite a, Show a) ⇒ FiniteShow a where
instance (Finite a, Show a) ⇒ FiniteShow a where
```

The definition of showFin is unimportant to the discussion; what matters is that it imposes a Show constraint on the existential types. The need for an auxiliary class representing the intersection of the Finite and Show constraints is irritating, but not a major concern.

Now let us try to both display and evaluate a computation:

```
-- does not type check
showAndLower :: Eq a ⇒ NM FiniteShow Vec a → (String, Vec a)
showAndLower v = (showFin v, lowerVec v)
```

This does not type check because lowerVec requires its argument to have exactly the type NM Finite Vec $a$, and the constraints Finite and FiniteShow are distinct. To overcome this, we would have to either define a variant of lowerVec that uses FiniteShow instead of Finite, or modify the type of the existing lowerVec function. That is, we must either duplicate or modify existing code, both of which are bad for modularity.

In general, whenever we want to apply two (or more) interpretations that impose distinct constraints to the same computation, we have to duplicate or modify those interpretations to use a new type class representing the intersection of all the required constraints. Note that while in this example one constraint is strictly stronger than the other, in general different interpretations can have disjoint constraints. We want a way to combine existing interpreters that have distinct constraints without modifying those interpreters.

### 6.2 A List of Existential Types

Jones et al.'s key idea is that, rather than parameterizing a GADT on a constraint, we can parameterize it on *the set of types within that GADT*. We can then constrain those types when we interpret the GADT, rather than during construction. We will demonstrate this technique by applying it to our constrained normal monads. We begin by replacing the constraint parameter with a list of types, making use of the *data kinds* [43] GHC extension:

```
data NM :: [∗] → (∗ → ∗) → ∗ → ∗ where
  Return :: a                                      → NM xs t a
  Bind   :: Elem x xs ⇒ t x → (x → NM xs t a) → NM xs t a
```

Instead of the constraint parameter, the context of the Bind constructor now uses the Elem type class, which represents type-level list membership:

```
class Elem (a :: ∗) (as :: [∗]) where ...
```

The intent is that Elem $a$ $as$ should only be satisfied if the type $a$ is an element of the list of types $as$, and thus the $xs$ parameter of NM limits the existential types that can appear on Bind constructors.

The definitions of the liftNM and foldNM functions remain unchanged from Section 3.2, but their types are modified to use Elem rather than a constraint parameter:

```
liftNM  :: Elem a xs ⇒ t a → NM xs t a
foldNM :: ∀ a r t xs. (a → r) →
  (∀ x. Elem x xs ⇒ t x → (x → r) → r) → NM xs t a → r
```

To use this list of types to constrain the existential types within the GADT, we need another type class:

```
class All (c :: ∗ → Constraint) (as :: [∗]) where ...
```

The intent is that All $c$ $as$ should only be satisfied if $c$ holds for every type in the list $as$. We can then use All to constrain the existential types when defining an interpretation, for example:

```
lowerVec :: (Eq a, All Finite xs) ⇒ NM xs Vec a → Vec a
showFin  :: (All Finite xs, All Show xs) ⇒ NM xs Vec a → String
```

Our goal, combining multiple interpretations with different constraints, is now straightforward:

```
showAndLower :: (Eq a, All Finite xs, All Show xs) ⇒
                NM xs Vec a → (String, Vec a)
showAndLower v = (showFin v, lowerVec v)
```

An explanation of the methods of the Elem and All classes, and how they are used, is beyond the scope of this paper, so we direct the reader to the original paper by Jones et al. [17].

## 7. Conclusions

In this paper we surveyed a variety of solutions to the constrained-monad problem, some of which require modifying the data type (normalization, restricted data types), and some of which require modifying the type class (restricted type classes). Some solutions are only proposals, as they require modifications to the Haskell language (restricted data types). Some solutions are better suited to the shallow version of the problem than the deep version (restricted monads, continuations), and some were straightforward to apply to other structures (normalized deep embeddings, restricted type classes). Our solution — using normalized deep embeddings with explicit constraints on the existential types — is pragmatic, simple to understand and implement, and useful in practice.

A valid concern about our technique is whether the benefits of being able to define a monad instance for a data type with constrained operations outweigh the cost of applying the technique. We expect the answer will vary between use cases. However, we observe that if a data type is provided abstractly by a library, then it is possible for all of the work to be done internally by the library implementer. For example, we have constructed an alternative to the `Data.Set` module, and made it available as the Set-Monad library [9]. This library provides an abstract type Set, with the same interface as `Data.Set`. However, it performs normalization internally, and thus is able to provide Monad and other instances for its Set type. That is, the Set type it exposes corresponds to the SetM type described in Section 2.2.

We think that awareness of the ability to reify computations expressed using structures containing existential types will have a significant impact on future EDSL designs. Both Syntactic-based EDSLs [30] and our own Sunroof [6] would be considerably weaker without this ability. Referring to our experience in Sunroof, the ability to perform monadic-bind reification has led to a useful compiler that directly supports multiple threading models and concurrency objects such as mutable variables and channels. We anticipate others using monadic-bind reification to build other effect-based EDSLs.

## Acknowledgments

We thank Andrew Farmer and Nicolas Frisby for suggesting the normal form and normalization algorithm for applicative functors. We also thank Heinrich Apfelmus and Ryan Ingram for observing that we could use the Operational and MonadPrompt libraries to perform the monadic normalization, respectively. Finally, we thank Ed Komp, Philip Hölzenspies and the anonymous reviewers for feedback on earlier versions of this paper, and Edward Kmett for his helpful comments on restricted monads. This material is based upon work supported by the National Science Foundation under Grant No. 1117569.

## References

[1] S. Adams. Efficient sets — a balancing act. *Journal of Functional Programming*, 3(4):553–561, 1993.

[2] H. Apfelmus. The Operational monad tutorial. *The Monad.Reader*, 15:37–55, 2010.

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

[4] M. M. T. Chakravarty, G. Keller, and S. Peyton Jones. Associated type synonyms. In *International Conference on Functional Programming*, pages 241–253. ACM, 2005.

[5] C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(3):455–481, 2003.

[6] A. Farmer and A. Gill. Haskell DSLs for interactive web services. In *Cross-model Language Design and Implementation*, 2012.

[7] J. Gibbons and B. C. dos Santos Oliveira. The essence of the iterator pattern. *Journal of Functional Programming*, 19(3–4):377–402, 2009.

[8] A. Gill. Type-safe observable sharing in Haskell. In *Haskell Symposium*, pages 117–128. ACM, 2009.

[9] G. Giorgidze, 2012. URL `http://hackage.haskell.org/package/set-monad`.

[10] P. Hudak. Modular domain specific languages and tools. In *International Conference on Software Reuse*, pages 134–142. IEEE Computer Society, 1998.

[11] J. Hughes. A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, 22(3):141–144, 1986.

[12] J. Hughes. Restricted data types in Haskell. In *Haskell Workshop*, 1999.

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

[14] G. Hutton, M. Jaskelioff, and A. Gill. Factorising folds for faster functions. *Journal of Functional Programming*, 20(3&4):353–373, 2010.

[15] R. Ingram and B. Felgenhauer, 2008. URL `http://hackage.haskell.org/package/MonadPrompt`.

[16] M. Jaskelioff. Monatron: An extensible monad transformer library. In *Implementation and Application of Functional Languages 2008*, volume 5836 of *LNCS*, pages 233–248. Springer, 2011.

[17] W. Jones, T. Field, and T. Allwood. Deconstraining DSLs. In *International Conference on Functional Programming*, pages 299–310. ACM, 2012.

[18] E. Kidd. How to make Data.Set a monad, 2007. URL `http://www.randomhacks.net/articles/2007/03/15/data-set-monad-haskell-macros`.

[19] O. Kiselyov. Restricted monads, 2006. URL `http://okmij.org/ftp/Haskell/types.html#restricted-datatypes`.

[20] O. Kiselyov, S. Peyton Jones, and C. Shan. Fun with type functions. In *Reflections on the Work of C.A.R. Hoare*, chapter 14, pages 301–331. Springer, 2010.

[21] E. A. Kmett, 2013. URL `http://hackage.haskell.org/package/kan-extensions`.

[22] K. Läufer and M. Odersky. Polymorphic type inference and abstract data types. *Transactions on Programming Languages and Systems*, 16 (5):1411–1430, 1994.

[23] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Principles of Programming Languages*, pages 333–343. ACM, 1995.

[24] C. Lin. Programming monads operationally with Unimo. In *International Conference on Functional Programming*, pages 274–285. ACM, 2006.

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

[26] M. Mitrofanov. A problem defining a monad instance, 2009. URL `http://www.haskell.org/pipermail/haskell-cafe/2009-November/068761.html`.

[27] E. Moggi. Computational lambda-calculus and monads. In *Logic in Computer Science*, pages 14–23. IEEE Press, 1989.

[28] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[29] D. Orchard and T. Schrijvers. Haskell type constraints unleashed. In *International Conference on Functional and Logic Programming*, pages 56–71. Springer, 2010.

[30] A. Persson, E. Axelsson, and J. Svenningsson. Generic monadic constructs for embedded languages. In *Implementation and Application of Functional Languages 2011*, volume 7257 of *LNCS*, pages 85–99. Springer, 2012.

[31] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *International Conference on Functional Programming*, pages 50–61. ACM, 2006.

[32] S. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(1):1–82, 2007.

[33] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Programming Language Design and Implementation*, pages 199–208. ACM, 1988.

[34] N. Sculthorpe, 2013. URL `http://hackage.haskell.org/package/constrained-normal`.

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

[36] M. Spivey. A functional theory of exceptions. *Science of Computer Programming*, 14(1):25–42, 1990.

[37] J. Svenningsson and E. Axelsson. Combining deep and shallow embedding for EDSL. In *Trends in Functional Programming 2012*, volume 7829 of *LNCS*, pages 21–36. Springer, 2013.

[38] W. Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008.

[39] J. Vizzotto, T. Altenkirch, and A. Sabry. Structuring quantum effects: Superoperators as arrows. *Mathematical Structures in Computer Science*, 16(3):453–468, 2006.

[40] J. Voigtländer. Asymptotic improvement of computations over free monads. In *Mathematics of Program Construction*, pages 388–403. Springer, 2008.

[41] P. Wadler. Comprehending monads. In *LISP and Functional Programming*, pages 61–78. ACM, 1990.

[42] P. Wadler. The essence of functional programming. In *Principles of Programming Languages*, pages 1–14. ACM, 1992.

[43] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. In *Types in Language Design and Implementation*, pages 53–66. ACM, 2012.