# Freer Monads, More Extensible Effects

Oleg Kiselyov
Tohoku University, Japan
oleg@okmij.org

Hiromi Ishii
University of Tsukuba, Japan
h-ishii@math.tsukuba.ac.jp

## Abstract

We present a rational reconstruction of extensible effects, the recently proposed alternative to monad transformers, as the confluence of efforts to make effectful computations compose. Free monads and then extensible effects emerge from the straightforward term representation of an effectful computation, as more and more boilerplate is abstracted away. The generalization process further leads to freer monads, constructed without the Functor constraint. The continuation exposed in freer monads can then be represented as an efficient type-aligned data structure. The end result is the algorithmically efficient extensible effects library, which is not only more comprehensible but also faster than earlier implementations.

As an illustration of the new library, we show three surprisingly simple applications: non-determinism with committed choice (LogicT), catching IO exceptions in the presence of other effects, and the semi-automatic management of file handles and other resources through monadic regions.

We extensively use and promote the new sort of 'laziness', which underlies the left Kan extension: instead of performing an operation, keep its operands and pretend it is done.

***Categories and Subject Descriptors*** D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Control primitives; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure

***Keywords*** free monad, Kan extension, effect handler, open union, type and effect system, effect interaction, coroutine

## 1. Introduction

That monads do not compose was recognized as a problem early on [29]. Two independently-written expressions using different side-effects (and hence monads) are difficult to combine in one program. Modifying a small part of a large program to use a new side-effect (e.g., adding debug output) sends ripples of changes throughout the code base. The very same difficulty of adding and combining effects has plagued denotational semantics [7]. In fact, monads, introduced by Moggi as a way to structure denotational semantics, inherited that problem. One can identify three approaches to solving it. The most popular is monad transformers [22], implemented in the widely used monad transformer library (MTL). They are based on Moggi's original idea of "monads with a hole", adding to it the lifting of monad operations through the transformer stack. The second approach combines monads through a quite complicated coproduct [24], whose simplification has lead to the free monad popularized in Data types à la carte [30]. The third, presented just before monad transformers, looked at effects as an interaction and introduced side-effect–request handlers [7]. That idea of effect handlers, generalizing exception handlers, was picked up in [4, 26], and developed into the language Eff. In Haskell, it was implemented as extensible effects [21] and [17].

We present, in §2, a unifying view: we derive the free monad and extensible effects by progressively abstracting the straightforward term representation of an effectful computation. Extensible effects emerge as the combination of the ideas of free monads and open union. The unifying, rational reconstruction is not only edifying: it pointed to the further generalization in §2.4: freer monads, free even from the Functor constraint. Freer (or, free-er, for emphasis) monad is an algebraic data type that is a monad by the very construction, just like list is a monoid by construction.

Besides intellectually satisfying, the freer monads are more economical with memory, avoiding rebuilding of the request data structure on each bind operation. Mainly, by exposing the continuation the freer monads made it easier to represent it differently, as a type-aligned sequence data structure [31], which improved the performance algorithmically. §3 describes the improved extensible effects library and §4 demonstrates its better performance on several benchmarks, in comparison with MTL and other effect handler libraries.

Our contribution thus is *rationally deriving* – telling the compelling story – of the freer monad, which supports the easy addition, composition and also subtraction (that is, encapsulation) of effects. It is so far the most efficient and expressive extensible monad. We demonstrate the expressivity on three applications, which were previously considered difficult for extensible effects or monads in general. §5 shows the exceptionally straightforward implementation of non-determinism with committed choice (the LogicT monad). §6 presents the surprisingly simple implementation of catching IO errors in monads other than IO. At last IO exceptions behave, with regard to other effects (State, in particular), just as non-IO exceptions. Finally, §7 is the ultimate demonstration of effect encapsulation: monadic regions, re-implementing and simplifying the transformer-based library of [19].

§8 describes the related work. The complete code is available at `http://okmij.org/ftp/Haskell/extensible/Eff1.hs`.

## 2. Derivation of Free-er Monad

In this section we derive the freer and extensible monads by progressively removing boilerplate from the term representation of effects. The result, however elegant, has poor performance, to be improved in §3.

### 2.1 Reader Effect

We start with the simplest side effect: dynamic binding, or Reader in the MTL terminology. Reader computations depend on a value supplied by the environment, that is, their context. A side-effect can be understood [7] as an interaction of an expression with its context. The possible requests can be specified as a data type, which in our case is[1]

```
data It  i  a = Pure a
              | Get (i  → It  i  a)
```

Such an algebraic modeling of possible operations was pioneered in Haskell by Hughes [14] and is now known in Haskell as 'operational' [1]. Hinze [13] gave the lucid demonstration of this technique, to derive backtracking monad transformers. (We also deal with non-determinism, in §5). The expression Pure e marks the computation e that makes no requests, silently working towards a value of the type a. The request Get k asks the context for the (current dynamically-bound) value of the type i. Having received the value i, the computation k i :: It i a continues, perhaps asking for more values from the context. One may hence call Get's argument k a continuation.

The simplest asking computation is

```
ask ::  It  i  i
ask = Get Pure
```

which immediately returns the received value. Bigger computations are built with the help of the monad bind (≫= ): It i is a monad.

```
instance  Monad (It i)  where
  −− return ::  a → It  i  a
  return  = Pure

  −− (≫= ) :: It  i  a → (a → It  i  b) → It  i  b
  Pure x ≫= k = k x
  Get k'  ≫= k = Get (k' ⋙  k)
```

The last clause in the definition of bind says that a computation that waits for an input and then continues as k', and after that, as k – is the computation that continues after waiting as the composition of k' and k. The operation (⋙ ), Kleisli composition, is the composition of effectful functions:

```
(⋙ ) ::  Monad m ⇒ (a → m b) → (b → m c) → (a → m c)
f  ⋙  g = (≫=  g) ∘ f
```

Here are two examples of bigger Reader computations

```
addGet ::  Int  → It  Int  Int
addGet x = ask ≫= \i → return  (i +x)

addN ::  Int  → It  Int  Int
addN n = foldl  (⋙ ) return  (replicate   n addGet) 0
```

The latter asks for n numbers and returns their sum.

The computations addGet and addN make requests to the context. We need to define how to reply, that is, how to "run" these computations. The following interpreter gives the same value i on each request: It i a is indeed interpreted as the Reader monad.

```
runReader ::  i  → It  i  a → a
runReader _ (Pure v)  = v
runReader x (Get k)  = runReader x (k x)
```

Unlike the MTL Reader, It i a may be treated differently: each request gets a new value, as if read from an input stream:

```
feedAll   ::   [i]  → It  i  a → a
feedAll   _ (Pure v)  = v
feedAll   []  _         = error  "end_of_stream"
feedAll   (h  :  t)  (Get k)  = feedAll  t  (k h)
```

In this interpretation, It i a is called an iteratee and feedAll an enumerator [18].

### 2.2 Reader/Writer Effect

Let us add another effect: rather than asking a context for a value, we tell the context. This is a Writer, or tracing effect.

```
data IT  i   o a = Pure a
               |  Get    (i  → IT  i   o a)
               |  Put o (()  → IT  i   o a)
```

The Put o k request tells the value o to the context. After the context acknowledges with ()[2] the computation continues as k (). The extended IT i o is also a monad:

```
instance  Monad (IT i o) where
  return  = Pure
  Pure x     ≫= k = k x
  Get k'     ≫= k = Get   (k'  ⋙  k)
  Put x k'   ≫= k = Put x (k'  ⋙  k)
```

Again, a computation that tells the context and continues as k' and then as k, really continues as k' ⋙ k.

In MTL's Writer monad, the told value must have a Monoid type. Our IT i o has no such constraints. If we write a Writer-like IT interpreter to accumulate the told values in a monoid, it will have the Monoid o constraint then:

```
runRdWriter ::  Monoid o ⇒ i  → IT  i   o a  → (a,o)
runRdWriter i  m  = loop mempty m
 where
   loop  acc  (Pure x)    = (x,acc)
   loop  acc  (Get k)     = loop acc  (k  i )
   loop  acc  (Put o k)  = loop (acc `mappend` o)  (k ())
```

There are other ways of interpreting IT i o a requests, for example, keeping the last told value, or writing the told value to stderr. Yet another interpreter, of IT s s computation takes the told value as the one to give when next asked, thus treating IT s s as a State computation.

The IT i o computation is an extension of It i. Alas, data types are not extensible. Therefore, we had to change the data type name and hence modify (the signatures of) addGet and addN, even if their code does not care about the new writer effect and remains essentially the same.

### 2.3 Free Monad

A data type describing an effectful computation such as It i a and IT i o a follows a common pattern: It is a recursive data type, with the Pure variant for the absence of any requests, and the variants for requests, usually containing the continuation that receives the reply (except for exceptions that do not expect any reply). The recursive occurrences of the data type are always as the return type of the continuations, that is, in covariant positions. This pattern, of pure and effectful parts and covariant recursive occurrences can be captured as

```
data Free  f  a  = Pure a
                 |  Impure (f  (Free  f  a))
```

where f is a (categorical) functor, that is, in f a, the type a occurs covariantly. The latter phrase means that if we can convert a value of type a into some other value of type b, we can also turn f a into f b. The Functor type class captures that meaning literally:

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

The concrete instantiations of f define the types of requests and replies, that is, the effect signature of a particular effectful computation. This splitting of a recursive data type such as It i into a non-recursive "structure component" and the recursive tying the knot Free f a was pioneered in [28] (who also used extensible effectful interpreters as one of the examples).

[1] The choice of the name It should become clear shortly.

[2] In Haskell, this () acknowledgment is not needed, but it fits our story better.

The monad instances for It i and IT i o also look very much alike. It is a shame to keep writing such instances for each new effect and each combination of effects. The Free f data type lets us capture the common pattern:

```
instance Functor f ⇒ Monad (Free f) where
  return = Pure
  Pure a    ≫= k = k a
  Impure f  ≫= k = Impure (fmap (≫= k) f)
```

Thus Free f for a functor f is a monad – the free monad. New effects will have new effect signatures f, but the single instance of **Monad** (Free f) will work for all of them, with no further re-writing.

As an example, the earlier IT i o computation may now be specified as

```
data ReaderWriter i o x = Get (i → x) | Put o (() → x)
instance Functor (ReaderWriter i o) where ...
type IT i o a = Free (ReaderWriter i o) a
```

The word "free" in free monad refers to the category theory's construction of the left adjoint of a forgetful operation [2]. In English, if we take a monad, say, State s with its return, bind, fmap, put and get operations and forget the first two, we can recover the monad as Free (State s), with prosthetic return and bind. In short, we get the **Monad** instance for free.

In general monads do not compose: if M1 a and M2 a are monads, M1 (M2 a) is generally not. Free monads however are a particular form of monads, defined via a functor. Functors do compose. We will exploit that fact after one more generalization.

### 2.4 Free-er Monads

Let us look more carefully at the Monad instance for Free f. The purpose of fmap there is to extend the continuation, embedded somewhere within (f (Free f a)), by (⋙)-composing it with the new k. The operation fmap lets us generically modify the embedded continuation, for any request signature.

Since the continuation argument is being handled so uniformly, it makes sense to take it out of the request signature and place it right into the fixed request data structure, as the second argument of Impure:

```
data FFree f a where
  Pure   :: a → FFree f a
  Impure :: f x → (x → FFree f a) → FFree f a
```

The remaining part of the request signature f x tells the type x of the reply, to be fed into the continuation. Different requests have their own reply types, hence x is existentially quantified. Our ReaderWriter effect gets then the following signature:

```
data FReaderWriter i o x where
  Get :: FReaderWriter i o i
  Put :: o → FReaderWriter i o ()
```

It is a GADT: the type variable x in FReaderWriter i o x is instantiated depending on the type of the request. For Get, the reply type is i, and for Put, it is unit. The IT i o a is now

```
type IT i o a = FFree (FReaderWriter i o) a
```

The monad instance for FFree f no longer needs the Functor or any other constraint on f:

```
instance Monad (FFree f) where ...
  Impure fx k' ≫= k = Impure fx (k' ⋙ k)
```

FFree f is more satisfying since it abstracts more of the common pattern of accumulating continuation, compared to Free. It is more general, not imposing any constraints on f – it is "freer". Continuing our example of State s from the end of §2.3, we can now forget not only return and bind but also the fmap operation, and still recover the state monad through FFree (State s) construction. We no longer have to bother defining the basic monad and functor operations in the first place: We now get not only the **Monad** instance but also the Functor and Applicative instances for free.

Freer monad is also more economical in terms of memory (and running time) because the continuation can now be accessed directly rather than via fmap, which has to rebuild the mapped data structure. The explicit continuation of FFree also makes it easier to change its representation, which we will do in §3.

Marcelo Fiore has suggested in private communication that the above FFree construction is the left Kan extension. To highlight this point we show another derivation of FFree. Recall, if f :: ∗ → ∗ is a functor, we can convert f x to f a whenever we can map x values to a values. If g :: ∗ → ∗ is not a functor, such a conversion is not possible. We can "cheat" however: although we cannot truly fmap h :: x → a over g x, we can keep its two operands as a pair, and assume the mapping as if it were performed:

```
data Lan (g :: ∗ → ∗) a where
  FMap :: (x → a) → g x → Lan g a
```

Any further mapping over Lan g a updates the original mapping, leaving g x intact. That is, Lan g is now a "formal" functor:

```
instance Functor (Lan g) where
  fmap h (FMap h' gx) = FMap (h ∘ h') gx
```

This Lan construction is the Left Kan extension. One may think of it as a free Haskell Functor – Functor by construction – just as a list is a free Monoid.

Let us see what Free (Lan g) is: substituting f in the type of (f (Free f a)) → Free f a of Free.Impure with Lan g gives us

```
∃ x. (x → (Free (Lan g) a)) → g x → Free (Lan g) a
```

which is the type of FFree.Impure. Hence

```
type FFRee g = Free (Lan g)
```

Incidentally, the type-aligned sequences, which we will use in §3, are essentially Free-er Applicative.

By analogy with the "free functor" Lan g we may also define a "free bifunctor"

```
data BiFree p a b where
  Bimap :: (a → b) → (c → d) → p a c → BiFree p b d
```

which is a generalization of the bifunctor used in [16, §6.3].

One last generalization step remains, to deliver the promised extensibility.

### 2.5 From Free(er) Monads to Extensible Effects

We have hinted in §2.3 that the form of free monads, built from functors, lends itself to composability since functors compose. This section demonstrates this composability on freer monads, built around left Kan extensions, which are functors by construction. There are two sides to composability: extensible monad type and modular interpreters. The latter part has been receiving less attention: for example, Data types à la carte [30] provides the former but not the latter.

A monad type is extensible if we can add a new effect without having to touch or even recompile the old code. The Free f or FFree f lets us do that: the monad type is indexed by the request signature f. Specifying this signature as an ordinary data type, such as ReaderWriter in §2.3 or GADT FReaderWriter in §2.4 is not extensible: an ordinary variant data type is a closed union, with the fixed number of variants. Open unions are relatively easy to construct, essentially by nesting the simplest union, the Either data type. The monad transformer paper [22] already showed such an implementation; Swierstra [30] used essentially the same.

We will use the open union that improves the previous implementations, including the one in [21]. It provides the (abstract) type Union (r :: [∗ → ∗]) x where the first argument r is a type-level list of effect labels, to be described shortly. The second argument is the response type, which depends on a particular request. The argument r lists all effects that are possible in a computation; a concrete Union r x value contains one request out of those listed in r.

It is crucial for extensibility to be able to talk about one effect without needing to list all others. For the sake of this effect polymorphism, our implementation provides a type class

```
class  Member t r where
  inj  ::  t v → Union r v
  prj  ::  Union r v → Maybe (t v)
```

that asserts that a label t occurs in the list r. If an effect is part of the union, its request can be injected and projected. We also offer another function, not present in [22, 30], to "orthogonally project" from the union,

```
decomp :: Union (t ': r) v → Either (Union r v) (t v)
```

obtaining either a request labeled t or a smaller union, without t. This function is needed for effect encapsulation. The earlier extensible effects library [21] provided a similar open union, implemented using overlapping instances and Typeable. The latter in particular attracted a large number of complaints. Deriving Typeable is indeed an extra step for a library aiming to encourage using many custom effects. For applications like monadic regions, Typeable was quite an obstacle, as we discuss in §7. The current implementation uses neither overlapping, nor Typeable. It also does not provide the no longer needed Functor instance.

The extensible freer monad, the monad of extensible effects, is hence FFree with the open union:

```
data FEFree r a where
  Pure    ::  a → FEFree r a
  Impure ::  Union r x → (x → FEFree r a) → FEFree r a
```

A request label defines a particular effect and its requests. For example, the Reader and Writer effects have the following labels:

```
data Reader i x where
  Get ::  Reader i i
data Writer o x where
  Put ::  o → Writer o ()
```

Informally, we split the monolithic FReaderWriter request signature into its components (to be combined in the open union). The simplest Reader computation, ask of §2.1, can now be written as

```
ask ::  Member (Reader i) r ⇒ Eff r i
ask = Impure (inj Get) return
```

The signature tells that ask is an Eff r i computation which includes the Reader i effect, without telling what other effects may be present. Unlike the old ask of §2.1, the new one can be used, *as it is*, without any adjustments to code or the signature, in programs with other effects. The new ask is thus extensible.

Making interpreters such as runRdWriter of §2.2 modular is just as important, and not always achieved in the past. We describe them §3.

### 2.6 Performance Problem of Free(er) Monads

Free (and freer) monads are certainly elegant and insightful, but poorly performing. Let us look again at the FFree f monad instance

```
instance  Monad (FFree f) where ...
  Impure fx k'  ≫= k = Impure fx (k'  ⋙  k)
```

The bind operation traverses its left argument but merely passes around the right argument. Therefore, the performance of left-associated binds, like the performance of left-associated list appends, will be poor – algorithmically poor. For example, the running time of addN n, implemented either as the It i monad or the FEFree [Reader i] monad, is quadratic in n. This is because addN happens to associate addGets on the left. For example, addN 3 evaluates to

```
(((return  ⋙  addGet) ⋙  addGet) ⋙  addGet) 0
```

which takes 3 evaluation steps to

```
((Impure (inj  Get) return ∘ (+0)) ≫= addGet) ≫= addGet
```

The two evaluations of bind then produce the final request

```
Impure (inj  Get) ((return ∘ (+0) ⋙ addGet x) ⋙  addGet)
```

The continuation, the second argument to Impure, is the addGet chain we started with, only one link shorter. Processing the reply from the context will again take time linear in the size of the chain. Overall, processing $n$ requests takes $O(n^2)$ time. We refer the reader to [31] for more illustration and discussion of this performance problem, and for the general solution: representing the continuation as an efficient data structure, a type-aligned sequence.

## 3. Final Result: Freer and Better Extensible Eff Monad

This section describes our current, improved and efficient library of extensible effects. Thanks to the Freer monad and the new open union it became easier, compared to the version presented two years ago [21], to define a new effect and to write a handler for it. There is no longer any need for Functor and Typeable instances. The performance has also improved, algorithmically; see §3.3. Before showing off the library in §3.2 we describe the last key improvement, representing the continuation as an efficient data structure.

### 3.1 Composed Continuation as a Data Structure

The new library is based on the FEFree monad derived in §2.5 (repeated here for reference):

```
data FEFree r a where
  Pure    ::  a → FEFree r a
  Impure ::  Union r x → (x → FEFree r a) → FEFree r a
```

differing in one final respect: Now that the request continuation x → FEFree r a is exposed, it can be represented in other ways than just a function. The motivation for a new representation comes from looking at the monad instance for FEFree f

```
instance  Monad (FFree f) where ...
  Impure fx k'  ≫= k = Impure fx (k'  ⋙  k)
```

which extends the request continuation k' with the new segment k. The lesson of [31] is to represent this *conceptual* sequence of extending the continuation with more and more segments as a concrete sequence. It would contain all the segments that should be functionally composed – without actually composing them! We shall see soon that the composing is not really needed: it was just a way of accumulating continuation segments, and not an efficient way at that. (Another motivation to look for a new representation of continuations is the performance problem of free(er) monads, described in §2.6).

We call the improved FEFree r monad Eff r, where r, as in §2.5, is the list of effect labels. The request continuation – which receives the reply x and works towards the final answer a – then has the type x → Eff r a. We define the convenient type abbreviation for such effectful functions, that is, functions mapping a to b that also do effects denoted by r.

```
type Arr r a b = a → Eff r b
```

The job of the monad bind is to accumulate the request continuation, by (⋙ )-composing it with further and further Arr r a b segments. Rather than really doing the composition, we assume it as performed, and merely accumulate the pieces being composed in a data structure. The left Kan extension used the same 'pretend the operation performed' trick. The data structure has to be heterogeneous, actually, type-aligned [31]: the Arr r a b being composed have different a and b types, and the result type of one function must match the argument type of the next. The type-aligned sequences enforce this invariant by construction. We chose the sequence FTCQueue of the following interface

```
type FTCQueue (m :: ∗ → ∗ ) a b
tsingleton  ::  (a → m b) → FTCQueue m a b
(▷ ) ::  FTCQueue m a x → (x → m b) → FTCQueue m a b
```

```
(⋈ ) :: FTCQueue m a x → FTCQueue m x b → FTCQueue m a b
data ViewL m a b where
  TOne ::  (a → m b) → ViewL m a b
  (:| )  ::  (a → m x) → (FTCQueue m x b) → ViewL m a b
tviewl  ::  FTCQueue m a b → ViewL m a b
```

FTCQueue m a b represents the composition of one or more functions of the general shape a → m b. The operation tsingleton constructs a one-element sequence, (▷) adds a new element at the right edge and (⋈ ) concatenates two sequences; tviewl removes the element from the left edge. All operations have constant or average constant running time. Our FTCQueue may be regarded as the minimalistic version of a more general fast type-aligned queue FastTCQueue: see [31] and type-aligned on Hackage. Thus the composition of functions (continuation segments) a → Eff r t1, t1 → Eff r t2, . . . , tn → Eff r b is represented as

```
type Arrs  r  a  b  = FTCQueue (Eff r) a b
```

and the Eff r monad has the following form

```
data Eff  r  a  where
  Pure    ::   a → Eff  r  a
  Impure ::  Union r x → Arrs  r  x a → Eff  r  a
```

A composition of functions is a function itself; likewise Arrs r a b is isomorphic to the single Arr r a b (or a → Eff r b). In one direction,

```
singleK  ::   Arr  r  a  b → Arrs  r  a  b
singleK  = tsingleton
```

the conversion builds the sequence with one element. In the other direction,

```
qApp ::  Arrs  r  b w → b → Eff  r  w
qApp q x = case tviewl  q  of
    TOne k → k x
    k :|  t  → bind'  (k x)  t
  where bind'  ::   Eff  r  a → Arrs  r  a  b → Eff  r  b
          bind'  (Pure y)  k        = qApp k y
          bind'  (Impure u q)  k = Impure u (q ⋈ k)
```

The qApp operation applies the argument x to a composition of functions denoted by the sequence Arrs r a b. To be precise, it applies x to the head of the sequence k and 'tacks in' the tail t (if any) as it was. That is the performance advantage of the new representation for continuation. The bind' operation is like monad bind (≫= ) but with the continuation represented as the sequence Arrs r a b rather than the a → Eff r b function. If the application k x runs in constant time, the whole qApp q x takes on average constant time.

Finally, in the monad instance of Eff r

```
instance  Monad (Eff r) where
  return  = Pure
  Pure x         ≫= k = k x
  Impure u q  ≫= k = Impure u (q ▷ k)
```

the bind operation grows the sequence Arrs r x a of continuations by appending another segment, k, which takes constant time.

### 3.2 Library Showcase: Defining and Interpreting Effects

We now demonstrate the extensible effects library: writing and composing effectful computations with the Eff monad. We re-do the reader and writer example §2.1, §2.2 to show that now adding the writer does not have to change the earlier code.

An effect is defined first by listing its requests and the corresponding reply types. For the Reader i effect, the request merely asks for a reply of the type i.

```
data Reader  i  x  where
  Get ::   Reader  i  i
```

The simplest client that returns the received reply is hence

```
ask  ::   Member (Reader i) r ⇒ Eff  r  i
ask  = Impure (inj  Get)  (tsingleton   Pure)
```

Recall, tsingleton creates the singleton sequence. The following library function makes the sending of requests even easier:

```
send  ::   Member t r ⇒ t  v → Eff  r  v
send t  = Impure (inj  t)  (tsingleton   Pure)
```

The other Reader computations addGet and addN of §2.1 are expressed in terms of ask and monad operations; their code is hence unchanged. Here they are, for the ease of reference:

```
addGet ::  Member (Reader Int) r ⇒ Int  → Eff  r  Int
addGet x = ask  ≫= \i → return  (i +x)

addN ::  Member (Reader Int) r ⇒ Int  → Eff  r  Int
addN n = foldl  (≫ ) return  (replicate   n addGet) 0
```

Their types however become more general: addN n has the Reader effect *and* can be used in computations that do other effects.

Interpreters of Reader requests now have to keep in mind there may be other request types, for other interpreters to deal with. Here is the new version of runReader from §2.1:

```
runReader ::  i  → Eff  (Reader i  ':  r)  a → Eff  r  a
runReader i  m = loop m where
 loop  (Pure x)         = return x
 loop  (Impure u q)  = case decomp u of
           Right  Get → loop $  qApp q  i
           Left     u    → Impure u (tsingleton   (qComp q loop))
```

The type signature says that runReader i receives the Eff computation with the Reader i effect, and returns the Eff computation without. The Reader i effect is thus handled, or encapsulated. The code indeed replies to the Get request – leaving other requests for other interpreters, see the Left u case. After that other interpreter replies, the program resumes and may make further Get requests. That is why we append the reader interpreter loop to the reply continuation q, using the function qComp:

```
qComp :: Arrs  r  a  b → (Eff  r  b → Eff  r'  c)  → Arr  r'  a  c
qComp g  h = h ∘ qApp g
```

The result continuation has the different list of effect labels r' since some of the effects will be handled by the interpreter h.

The common request handling code is factored out in the following function provided by the library:

```
handle_relay   ::   (a → Eff  r  w) →
                          (∀ v.  t  v → Arr  r  v w → Eff  r  w) →
                          Eff  (t  ':  r)  a → Eff  r  w
handle_relay   ret   _  (Pure x)         = ret  x
handle_relay   ret   h  (Impure u q)  = case decomp  u of
  Right  x  → h  x  k
  Left    u → Impure u (tsingleton   k)
 where k = qComp q (handle_relay ret h)
```

The first two arguments of handle_relay are like return and bind. The reader interpreter can be thus written simply as

```
runReader i  = handle_relay  return   (\Get k → k  i )
```

The last part of handle_relay's signature, Eff (t ': r) a → Eff r w, shows that the label t of the handled effect must be at the top of the list of effect labels r. Whereas effectful functions like addN above or rdwr below regard r truly as a set of effect labels, with no particular order, handlers impose the order. This fact is noticeable already in the interface of Union in §2.5: in the signatures of inj and prj, effects are represented by the type variable r, with a Member constraint. On the other hand, decomp takes the collection of effects to be specifically a list, with the projected effect t at its head. In our experience so far, this imposition of order by the handlers has not been a problem. It is theoretically unsatisfying. Although we could avoid it by playing with Constraint types, the required type annotations made the result impractical. Unfortunately, there does not seem to be any convenient way in Haskell to discharge one type class constraint by submitting the corresponding dictionary. (Implicit parameters do come very close.)

To run the Eff computation after all effects have been handled by the corresponding interpreters, the library provides

```
run ::   Eff  '[]   a → a
run  (Pure x)  = x
```

The Impure case is unreachable since Union '[] a has no (non-bottom) values. Thus we run addGet 1 as

```
run ∘ runReader 10 $ addGet 1
```

Let us add the writer effect, of telling the context the value of type o:

```
data Writer o x where
  Put :: o → Writer o ()

tell :: Member (Writer o) r ⇒ o → Eff r ()
tell o = send $ Put o
```

The type of tell lets it be combined in any effectful computation with the Writer o effect. Here is a sample combined reader-writer computation

```
−− rdwr :: (Member (Reader Int) r, Member (Writer String) r)
−− ⇒ Eff r Int
rdwr = do{ tell "begin"; r ← addN 10; tell "end"; return r }
```

whose inferred type is shown in the comments. Because the type of addN is polymorphic in r, we could use addN as it was in a computation with more effects (and similarly, for tell).

In §2.2, the interpreter for Reader-Writer computations was the monolithic runRdWriter, which handled both types of requests. Now we can interpret only the Writer requests

```
runWriter :: Eff (Writer o ': r) a → Eff r (a,[o])
runWriter =
  handle_relay (\x → return (x,[]))
     (\(Put o) k → k () ≫= \(x,l) → return (x, o:l))
```

and literally compose it with the previously written runReader. The sample reader-writer computation rdwr is thus run as

```
(run ∘ runReader 10 ∘ runWriter) rdwr
```

Since the reader and writer effects commute, the order of the interpreters can be switched without affecting the result.

One may write other reader and writer interpreters, for example, handling Reader and Writer requests together; the value last told becomes the value to give on the next Reader request. We thus implement State, by decomposing it into the reading and mutating parts. It becomes easier to tell, just from their inferred type, which parts of the computation mutate the state.

```
runStateR :: Eff (Writer s ': Reader s ': r) w → s → Eff r (w,s)
runStateR m s = loop s m where
  loop :: s → Eff (Writer s ': Reader s ': r) w → Eff r (w,s)
  loop s (Pure x) = return (x, s)
  loop s (Impure u q) = case decomp u of
    Right (Put o) → k o ()
    Left  u  → case decomp u of
      Right Get → k s s
      Left u → Impure u (tsingleton (k s))
   where k s = qComp q (loop s)
```

### 3.3 Improved Performance

This section re-analyzes the performance of the freer monad after changing the representation of the request continuation, on the problematic example from §2.6. As before, addN 3 evaluates to

```
(((return ⋙ addGet) ⋙ addGet) ⋙ addGet) 0
```

and then to

```
((Impure (inj Get) [return ∘ (+0)]) ≫= addGet) ≫= addGet
```

The two evaluations of bind produce the request

```
Impure (inj Get) [return (+0), addGet, addGet]
```

(where we used the list notation for the type-aligned sequence for clarity). So far, the process and its result seem similar to that for the non-optimized monad in §2.5. The fact that the continuation of the Get request is now represented as an efficient sequence makes the difference. When a runReader interpreter replies, say, with the value v1, it does the following operations that eventually produce a new request. For emphasis we denote as t the tail of the request continuation (in our example, t is the singleton sequence [addGet]):

```
  qApp (return (+0) : addGet : t) v1
↝ return v1 `bind'` (addGet : t)
↝ addGet v1 `bind'` t
↝ Impure (inj Get) (return (+v1) : t)
```

The above reduction sequence has dealt only with the two head elements of the entire continuation of the original request. The tail t was merely passed around and not even looked at. Furthermore, all FTCQueue operations involving t such as concatenation, etc., were constant-time. Therefore, the entire sequence of reductions above runs in time independent of the length of t. The run-time of the entire addN n computation is thus linear in n. Compared with the previous version §2.5, we obtain the algorithmic improvement in performance, from quadratic to linear. The key to the performance is the ability to look at and remove initial segments from the accumulated request continuation. If the continuation is represented as a composition of functions, we cannot 'uncompose' them – but we can deconstruct a data structure.

## 4. Performance Evaluation

This section reports on several micro-benchmarks used to evaluate the performance of extensible effects (EE) relative to monad transformer library MTL, Kammar's et al. "Handlers in action (HIA)" [17] and the old version of EE presented in [21].

The benchmark code was compiled with GHC 7.8.3 with the flag −threaded −O −rtsopts. We ran the benchmark on an Intel Core i7 (2.8GHz) laptop with 16GB of RAM. The Criterion framework was used to report the run-time.

### 4.1 Deep-monad-stack Benchmarks

First, we ran two benchmark computations with many effects (deep monad stacks). These benchmarks do a simple stateful computation with many Reader layers under or over the target State layer. The core State computation is as follows:

```
benchS ns = foldM f 1 ns where
  f acc x | x `mod` 5 == 0 = do
                          s ← get
                          put $! (s+1:: Integer)
                          return $! max acc x
  f acc x = return $! max acc x
```

Strictness annotations are to avoid space leaks.

Figure 1 shows the results. If we add the extra Reader layers *under* the State (the top of Fig.1), EE runs in constant time, while the MTL version takes linear time in the number of layers. Our EE is about 12% faster than HIA, and 40% faster than the old EE. If the State layer is at the bottom of the monad stack (the bottom of Fig.1) the run-time of HIA and EE versions is linear in the number of layers, whereas MTL and the old EE are quadratic. The results confirm the analyzes of performance in §2.6 and §3.3: the EE library presented in this paper indeed algorithmically improves the performance over the old version – as well over MTL for deep monad stacks. The overhead of MTL can indeed be severe for deep stacks. We also see that EE is competitive with HIA.

#### 4.1.1 Monad Stack Depth and Memory Consumption

We also evaluated the memory efficiency of the two deep-monad-stack benchmarks by taking the memory profile, using GHC with RTS options −N2 −prof −p −N2 −p −hm. Figure 2 shows the result.

Adding Reader layers under the State layer (the top of Fig.2) affects the memory consumption of effect libraries (EE and HIA) very little. The memory use does increase linearly with the number of layers, but by such a small amount that it is very difficult to see in the figure. In contrast, the amount of allocated memory for MTL is quadratic in the number of layers, and is quite large compared to the effect libraries. If we add Reader over the State layer (the

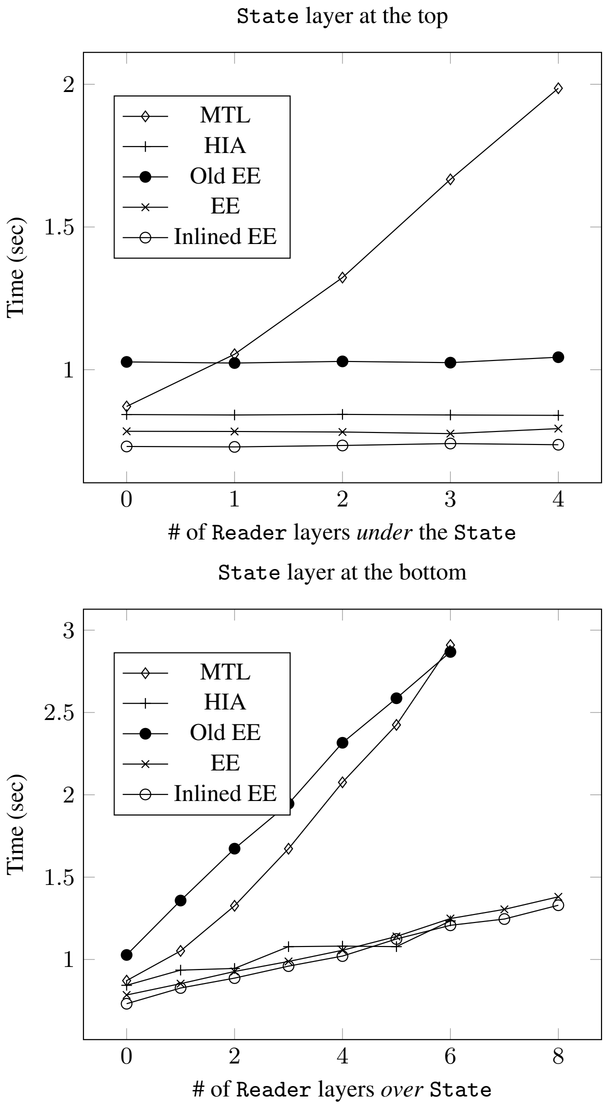

**Figure 1.** Runtime in seconds for MTL, HIA, Old EE, EE and Inlined EE. $x$-axis corresponds to the number of `Reader` layers *under* (top) or *over* (bottom) the target `State` layer.

bottom of Fig.2), the linear increase in allocated memory for effect libraries becomes quite more noticeable. The MTL memory use is again quadratic in the number of layers. The results confirm our expectation of the memory efficiency of the EE library presented in this paper.

### 4.2 Single-effect Benchmark

We have just seen that EE can overcome the overhead of handling very many effects. To see how EE and MTL compare for a single effect, we ran a simpler benchmark, with the single State or the single Error effect (table 1).

**Table 1.** A simple benchmark with a single layer (msec).

| | pure | MTL | HIA | Old EE | EE | Inlined EE |
|---|---|---|---|---|---|---|
| `State` | - | 15.2 | 7.16 | 840 | 579 | 488 |
| `Error` | 46.4 | 218 | 648 | 644 | 204 | 216 |

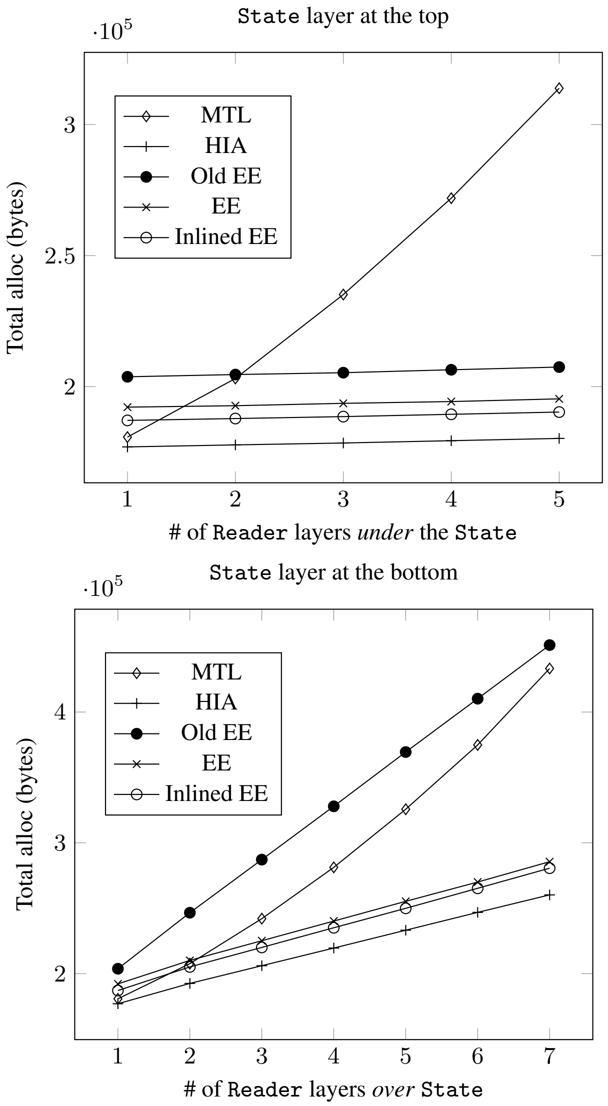

**Figure 2.** Total allocation in $10^5$ bytes for MTL, HIA, Old EE, EE and Inlined EE. $x$-axis corresponds to the number of `Reader` layers *under* (top) or *over* (bottom) the target `State` layer.

The single State benchmark counts down from $10,000,000$ to 0, using the State monad. The EE version is much slower than the MTL and HIA, 30 and 60 times correspondingly. This is because the State monad enjoys the preferential treatment by GHC, with dedicated optimization passes. Likewise, GHC is very good at optimizing simple CPS code employed in simple instances of HIA. Thus for the single State effect, our EE approach is not so suitable. The new library is still noticeably faster than the original EE version two years ago.

In contrast, for the Error monad EE and MTL have almost the same performance and notably, three times, faster than HIA and the old approach. The Error benchmark takes the product of $10,000,000$ copies of 1 and 0, raising a exception when the zero factor is found.

Thus for the single or few-layered monadic computations, EE can compete with individual single specialized monads in general, but for some monads, like State, it runs much more slowly.

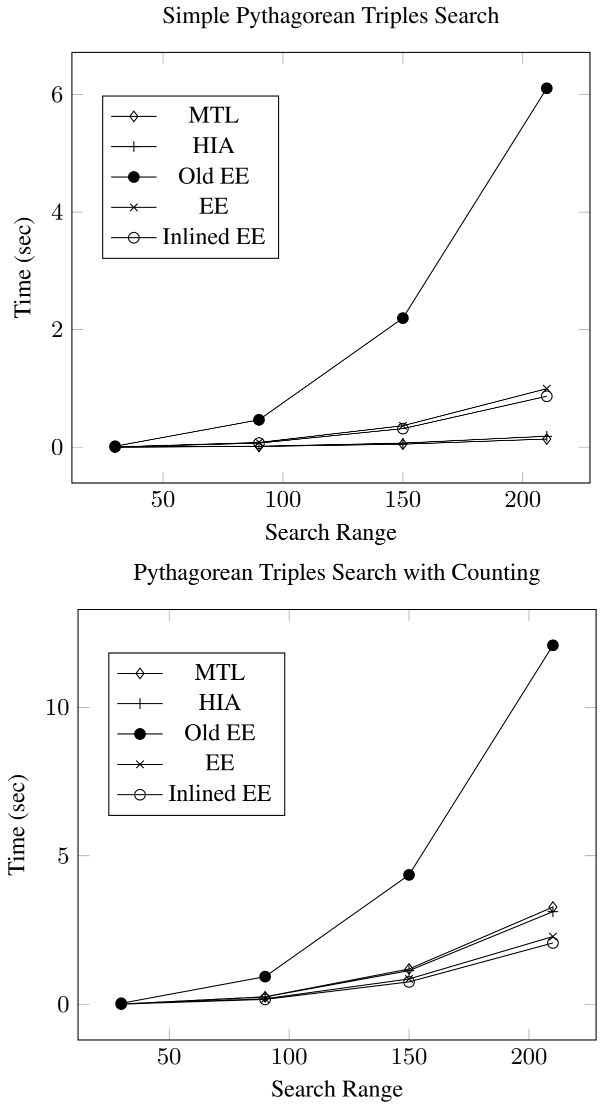

**Figure 3.** Runtime in seconds for MTL, HIA, Old EE, EE and Inlined EE. $x$-axis corresponds to the search range for Pythagorean triples.

### 4.3 Non-determinism Benchmarks

We have run another series of benchmarks, for the non-determinism effect, to be discussed in detail in §5.

The first benchmark (the top of Fig.3) searches for Pythagorean triples up to the given bound with non-deterministic brute-force:

```
iota k n = if k > n then mzero else return k `mplus` iota (k+1) n

pyth1 :: MonadPlus m ⇒ Int → m (Int, Int, Int)
pyth1 upbound = do
  x ← iota 1 upbound
  y ← iota 1 upbound
  z ← iota 1 upbound
  if x∗ x +y∗ y == z∗ z then return (x,y,z) else mzero
```

For the MTL version, we use the continuation monad transformer ContT. The result shows that our EE is much faster than old EE, but slightly slower than MTL and HIA. We should stress that our EE library, unlike HIA and MTL, implements the more general LogicT effect: non-determinism with committed choice.

The next benchmark adds to the previous one counting of the all attempted choices, using the State effect. The result at the bottom of Figure 3 shows that our EE approach is faster than the other alternatives.

The results confirm the good performance of EE, also for more complicated computations with many layers of effects.

### 4.4 Comparison with “Fusion for Free”

Very recently, Wu and Schrijvers [32] introduced the “Fusion for Free” approach for algebraic event handlers. Since their implementation is not yet published as a library, we will briefly compare performance in a qualitative manner. Specifically, we ran the benchmarks count_1 and count_2 from [32] for MTL, EE and Inlined EE. The result is shown in Table 2.

**Table 2.** Runtime in milliseconds for Counting benchmarks from Wu and Schrijvers [32]

| | MTL | EE | Inlined |
|---|---|---|---|
| count_1 $10^3$ | 0.000694 | 0.0592 | 0.0488 |
| $10^4$ | 0.00692 | 0.593 | 0.489 |
| $10^5$ | 0.0689 | 5.87 | 4.81 |
| count_2 $10^3$ | 0.202 | 0.306 | 0.287 |
| (Writer $10^4$ | 4.84 | 6.40 | 6.51 |
| bottom) $10^5$ | 54.4 | 84.7 | 80.8 |
| count_2 $10^3$ | 0.0859 | 0.345 | 0.316 |
| (Writer $10^4$ | 2.85 | 6.57 | 6.67 |
| top) $10^5$ | 37.3 | 85.2 | 84.2 |

Here, count_1 is the single State-effect benchmark, counting down in the State monad, similar to our benchmark in §4.2. This is the singular most unfavorable case for EE compared to MTL, since GHC has several optimizations that benefit the MTL State monad. The table shows that in all the cases, the run-time increases with the count not just linearly but proportionally. This qualitatively reproduces the behavior reported by Wu and Schrijvers in [32].

The next count_2 benchmark counts down using State, and also logs every intermediate value in the Writer monad. Wu and Schrijvers did not indicate which layer is on top, so we ran both cases, which proved to make little difference for EE (in contrast to MTL, however). The run-times again seem linear, but not proportional. In [32], the run-time of “Fusion” is proportional. Although the qualitative behavior again seems similar, quantitative comparison is clearly needed. We defer it to future work, when the code for [32] becomes available.

### 4.5 Inlining of Key Functions

The key functions of the EE library such as handle described in §3, contain a recursive reference but not a recursive invocation. These functions are hence safe to inline. To see if it makes any difference we added the INLINEABLE pragma for these functions. The pragma had almost no effect. The performance has improved slightly only when we inlined tviewl into qApp by hand (these functions are defined in different modules).

## 5. Non-determinism with Committed Choice

Non-determinism, with its inherent balancing of several continuations, may seem impossible to express as a freer monad, which explicitly deals with a single continuation. This section shows that not only the Eff monad can represent non-deterministic choice, but also that the representation preserves the sharing of continuations, lost in the standard free monad approach.

Free monad models non-determinism with the following request functor:

```
data Ndet x = MZero | MPlus x x
instance Functor Ndet where ...
```

MZero, like an exception, requests abandoning the current line of computation as unsuccessful; MPlus asks the context to choose between the two Ndet computations. This request signature comes straight from the interface for non-deterministic computations in Haskell: MonadPlus or Alternative:

```
instance MonadPlus (Free Ndet) where
  mzero        = Impure MZero
  mplus m1 m2 = Impure $ MPlus m1 m2
```

The MPlus constructor has two continuation arguments. How are we going to separate them into the single continuation argument of FFree? Let us consider the non-deterministic choice in context:

```
   (mplus m1 m2 ≫= k1) ≫= k2
   {The bind of the Free monad}
⇝ Impure (fmap (≫= k1) (MPlus m1 m2) ≫= k2
   {The fmap from the derived Functor instance }
⇝ Impure (MPlus (m1 ≫= k1) (m2 ≫= k1)) ≫= k2
   {Repeating for k2}
⇝ Impure (MPlus ((m1 ≫= k1) ≫= k2) ((m2 ≫= k1) ≫= k2))
```

That is, the two continuations collected by MPlus in fact have the common k1, k2 suffix. That suffix, albeit common, is not shared: although the two MPlus continuations share the common segments, they are independently composed. It is this common suffix that the freer monad will factor out and share.

After the common continuation suffix is separated out, what remains of MPlus is the request to the context to pick and return one of the two choices. There is no need to include the choices themselves in the request then. Hence in the Eff framework, the non-determinism effect has the following signature:

```
data NdetEff a where
  MZero :: NdetEff a
  MPlus :: NdetEff Bool

instance Member NdetEff r ⇒ MonadPlus (Eff r) where
  mzero        = send MZero
  mplus m1 m2 = send MPlus ≫= \x → if x then m1 else m2
```

To complete the implementation, we add an interpreter, such as the following, mapping the NdetEff-effect non-determinism to Alternative:

```
makeChoiceA :: Alternative f ⇒
               Eff (NdetEff ': r) a → Eff r (f a)
makeChoiceA = handle_relay (return ∘ pure) $ \m k → case m of
    MZero → return empty
    MPlus → liftM2 (<|>) (k True) (k False)
```

One may recognize in this code the "flip oracle" of [9, §3], which non-deterministically returns a boolean value. Just as Danvy and Filinski's code, we are capturing the context of mplus, represented as k above, and plugging first True and then False into the very same context.

The advantage of NdetEff over Alternative is not only the ability to mix NdetEff with other (non-applicative) effects, for example, State. It also supports the so-called "committed choice" [25], such as logical "if-then-else" (called "soft-cut" in Prolog):

```
ifte :: Member NdetEff r ⇒
        Eff r a → (a → Eff r b) → Eff r b → Eff r b
```

Declaratively, ifte t th el is equivalent to t ≫= th if the non-deterministic computation t succeeds at least once. Otherwise, ifte t th el is equivalent to el. The difference between ifte t th el and the seemingly equivalent (t ≫= th) \`mplus\`el is that in the latter el is a valid choice even if t succeeds. In the former, el is chosen if and only if t is the total failure. One of the examples of ifte is the many parser combinator with 'maximal munch': many p should keep applying the argument parser p for as long as it succeeds. The following is another, easier to explain albeit more contrived, example: computing primes

```
test_ifte   = do
  n ← gen
  ifte  (do d ← gen
             guard $ d <n && n `mod` d == 0)
    (\_ → mzero)
    (return n)
 where gen = msum ∘ fmap return $ [2..30]
msum :: MonadPlus m ⇒ [m a] → m a -- choose one from a list
```

Here gen non-deterministically produces a candidate prime and a candidate divisor. The prime candidate is accepted if all attempts to divide it fail. For example,

```
test_ifte_run  :: [Int]
test_ifte_run  = run ∘ makeChoiceA $ test_ifte
-- [2,3,5,7,11,13,17,19,23,29]
```

gives the result shown in the comment.

We actually implement not just ifte but the general

```
msplit :: Member NdetEff r ⇒
          Eff r a → Eff r (Maybe (a, Eff r a))
```

which expresses all other committed choice operations [20]. One may think of msplit as "inspecting" the argument computation, to see if it can succeed. If a computation gives an answer, it is returned along with the computation that may produce further answers. The implementation is so straightforward and small that it can be listed in its entirety:

```
msplit = loop [] where
 loop jq (Pure x)       = return (Just (x, msum jq))
 loop jq (Impure u q)  = case prj u of
  -- The current choice fails  (requested abort)
  Just MZero → case jq of
                   -- check if there are other choices
                   []      → return Nothing
                   (j:jq) → loop jq j
  Just MPlus → loop ((qApp q False):jq) (qApp q True)
  _  → Impure u (tsingleton  k) where k = qComp q (loop jq)
```

In words, msplit t intercepts the NdetEff requests of t. If t asks to choose, MPlus, one choice is pursued immediately and the other is saved in the work list jq of possible choices. The function finishes when the watched computation succeeds (the worklist is the collection of the remaining choices then) or when all possible choices failed.

We have demonstrated the most straightforward Eff implementation of not just non-determinism but non-determinism with committed choice (or, LogicT) [20].

## 6. Catching IO Exceptions

Handling IO errors in the presence of other effects abounds in subtleties. It was also thought to be a challenge for the Eff library. Not only has Eff met the challenge, it improves on MTL. With extensible effects, the state of the computation at the point of an exception is available to the handler. In MTL, an exception handler only has access to the state that existed at the point where it was installed (that is, catch was entered). Any further changes, up to the point of the exception, are lost.

Capturing IO errors in general MonadIO computations (not just the bare IO monad) has been a fairly frequently requested feature, going back to 2003[3]. An early approach[4] has been improved

[3] http://www.haskell.org/pipermail/glasgow-haskell-users/2003-September/005660.html http://haskell.org/pipermail/libraries/2003-February/000774.html

[4] http://okmij.org/ftp/Haskell/misc.html#catch-MonadIO

and polished through many packages (such as MonadCatchIO) and eventually de facto standardized in exceptions . The solution, although very useful in many circumstances is not without problems. For example, consider the following computation with the Writer and IO effects

```
do tell  "begin"; r ←faultyFn; tell  "end"; return r
`catch` (\e → return ∘ show $ (e:: SomeException))
```

where faultyFn throws an IO or a user-defined dynamic exception. With MTL, any Writer updates that happened after catch up to the point of the exception are lost. That is, after the above code finishes the accumulated trace has neither “end” nor “begin”. Such a transactional semantics is useful – but not when the Writer is meant to accumulate the debug trace. Alas, MTL does not give us the easy choice.

To understand the MTL behavior, recall that its WriterT String IO a monad is IO (a,String): it is the computation that produces the value a along with the contribution to the writer string. The catch is implemented as (see liftCatch in mtl).

```
catch h m = m `IO.catch` \e → h e
```

When an IO exception is raised, the value produced by m, including its Writer contribution, is lost. MTL's liftCatch for the State monad has the similar behavior of discarding the state accumulated since the catch is entered. In general, effect interaction in MTL depends on the order of the transformer layers; the IO monad is not a transformer however and must always be at the bottom of the stack.

If we execute the same code with the extensible-effect IO error handling [5] the trace accumulated by the writer of course has no "end" but it does have "begin". Here is the whole code for catching IO exceptions

```
catchDynE :: ∀ e a r.
             (MemberU2 Lift (Lift IO) r, Exc.Exception e) ⇒
             Eff r a → (e → Eff r a) → Eff r a
catchDynE m eh = interpose return h m
 where
   h :: Lift IO v → Arr r v a → Eff r a
   h (Lift em) k = lift (Exc.try em) ≫= \x → case x of
         Right x → k x
         Left  e → eh e
```

In the extensible effects library, IO computations are requested with the Lift IO effect

```
newtype Lift m a = Lift (m a)
```

whose interpreter

```
runLift :: Monad m ⇒ Eff '[Lift m] w → m w
```

is necessarily the last one, which is signified by the special MemberU2 Lift (Lift IO) r constraint. The library function interpose is a version of handle_relay that does not consider an effect handled although it does reply to its requests: interpose may also ‘re-throw’ effect's request. The function catchDynE intercepts IO requests to wrap them into the Exception.try to reify possible exceptions. Therefore, IO errors are instantly caught and do not immediately discard their continuation. The effect handlers in scope and their state are thus preserved.

We can also easily implement transactional behavior: an exception rolling-back the state to what it was when the exception handler was installed; see the source code for details.

## 7. Regions

Monadic Regions were introduced by Fluet and Morrisett [10] as a surprisingly simple version of the type-safe region memory management system. It may be thought of as a nested ST monad while also allowing reference cells allocated in a parent region to be used, relatively hassle-free, in any child region. Lightweight monadic regions [19] is the Haskell implementation of the extended version of Fluet and Morrisett's system, which was applied to IO resources such as file handles rather than memory cells, and is simpler to use. Lightweight regions statically ensure that every accessible file handle is open, while providing timely closing. The original Monadic Regions used an atomic monad, indexed by a unique region name; the lightweight version was built by iterating an ST-like monad transformer. Extensible effects, with its atomic Eff monad indexed by effects tempted one to re-implement lightweight regions closer to Fluet and Morrisett's original style while still avoiding the inconvenience of passing around parent-child–relationship witnesses. This challenge was set as future work in [21].

Implementing monadic regions with extensible effects was certainly a challenge. To ensure that an allocated resource such as a memory cell or a file handle do not escape from their region, Monadic Regions – like the ST s monad  – mark the types of the computation and its resources with a quantified (or rigid, in GHC parlance) type variable. Defining Typeable instances for such types was the first challenge. More worrisome, any type-level programming with types that include rigid variables never meant to be instantiated is fragile. Sometimes, incoherent instances [5] are needed, which is a rather worrisome extension that we are keen to avoid. Finally, lightweight monadic regions, although based on monad transformers, intentionally prohibited any lifting and hence the addition of other effects. Exceptions and non-determinism are clearly incompatible with the region discipline. On the other hand, State and Reader are benign and should be allowed.

All these challenges have been met[6]. Below we describe the salient points of the implementation.

Since the new version of extensible effects no longer uses Typeable, the first challenge disappears. The second one was difficult indeed. The most straightforward realization of Fluet and Morrisett's idea is to provide a RegionEff s effect indexed by the rigid type variable s taken to be the name of the region. File handles allocated within the region will be marked by that region's name:

```
newtype SHandle s = SHandle Handle
data RegionEff s a where
  RENew :: FilePath → IOMode → RegionEff s (SHandle s)
```

The data constructors are private and not exported. (The actual implementation is a bit more complex because it supports bequeathing of file handles to an ancestor region, see [19] for more discussion.)

The operation to allocate the new file handle will send a RENew request and obtain the handle marked with the region's name.

```
newSHandle :: Member (RegionEff s) r ⇒ −− simplified
              FilePath → IOMode → Eff r (SHandle s)
newSHandle fname fmode = send (RENew fname fmode)
```

The list of constraints is a bit simplified, omitting the type-level computation that scans the list of effect labels r and finds the name of the closest, that is, innermost, region. The interpreter of the requests

```
newRgn :: (∀ s. Eff (RegionEff s ': r) a) → Eff r a
```

like runST, has higher-rank type: informally, it allocates a fresh rigid type variable s, the fresh name for the region. The interpreter keeps the list of handles it was asked to allocate, closing all of them upon normal or exceptional exit. An operation using the handle has the type

```
shGetLine :: Member (RegionEff s) r ⇒
       SHandle s → Eff r String
```

that enforces that the region named s owning the handle is active: its name is among the current effect labels r. Incidentally, the signature

[5] http://okmij.org/ftp/Haskell/extensible/EffDynCatch.hs

[6] http://okmij.org/ftp/Haskell/extensible/EffRegion.hs
http://okmij.org/ftp/Haskell/extensible/EffRegionTest.hs

*automatically* allows the handle allocated in any ancestor region to be used in a child region.

The outlined implementation indeed works, save for two subtleties. It is indeed tempting to think of the rigid type variable s as the name for the region RegionEff s. Alas, the ever-present Member (RegionEff s) r constraint, checking that the RegionEff s effect is part of the current effect list r, cannot distinguish two types RegionEff s1 and RegionEff s2 that differ only in the rigid type variable. Although these variables will never be instantiated and hence never can be the same, the constraint-solving part of GHC does not know or understand this fact. Therefore, we have to give regions another name, a type-level numeral, which the constraint-solver can distinguish. Therefore, the signatures of newSHandle and newRgn (but not shGetline, etc) are slightly more complex than shown.

The second subtlety is allowing other effects besides RegionEff. Since all possible effects of a Eff r computation are listed in r, we merely need to look through the list to check if the effect is known to be benign. The implementation provides such SafeForRegion constraint, treating Reader and State as safe[7]. Exc SomeException is also allowed since newRgn specifically listens for this request.

The rest of the implementation is straightforward. It passes the old Lightweight Regions regression tests with minimal modifications.

## 8. Related Work

The library of extensible effects reported in this paper is the simplification and improvement of the library presented two years ago [21]. Eff was a co-density-transformed free monad – which was not made clear in that paper. The co-density transform is regarded as an optimization – alas, it does not work for modular interpreters, which have to reflect the continuation when relaying a request to another handler. The incompatibility of reflection with the co-density optimization was described in detail in [31]. We now use the simpler and quite better performing Freer monad with type-aligned sequences. The new Eff also uses the new implementation of open unions without the objectionable features: Typeable and overlapping instances. The applications described in §§5,6,7 are also new, for extensible effects.

One of the most common questions about Extensible Effects is their relation to Swierstra's well-known "Data types à la carte" [30]. Similarities are indeed striking: free monads, open unions, 'modular' monads leading to a type-and-effect system. Although the à la carte approach provides extensible monad type, it does not provide modular interpreters with modular effects and hence effect encapsulation. Related to the lack of compositionality are problems with type inference, requiring cumbersome and what should be unnecessary annotations. (The ambiguity in the definition of the subsumption relation on collection of types, which caused the inference problems, has been rooted out in the novel approach by [3].) See `http://okmij.org/ftp/Haskell/extensible/extensible-a-la-carte.html` for a detailed comparison with the old Eff library. The present paper moves past the free monad to freer monad.

In comparison with monad transformers, the interaction of effects in Eff depends not on the statically fixed order of transformers but on the order of effect interpreters and can even be adjusted dynamically (by interpreters that listen to and intercept other requests). See [21] for more extensive comparison with MTL. That paper relates Eff with other effect systems known at that time. In the following we compare Eff with the systems introduced since.

The effect system of Idris [6] is an implementation of algebraic effects in the dependently-typed setting. The paper [6] introduces a domain-specific language – a notation – for describing effectful computations and demonstrates the easy combination of effects. The handlers are specifies as instances of a type class. The effect order is globally fixed and effects are interpreted essentially at the top level; there is no encapsulation of effects. The paper makes an excellent case that effect handlers provide a more flexible and cleaner alternative to monad transformers. We disagree about limitations: as we show in our implementation, the effect approach is more, rather than less expressive than monad transformers.

The closely related to our work is Kammar's et al. "Handlers in action" [17]. Whereas our library manages sets of effects using both type-level constraints and type-level lists, Kammar et al. rely only on type-class constraints. Constraints truly represent an unordered set. Using constraints exclusively however requires all effect handler definitions be top-level since Haskell does not support local type class instances. Handlers in Action rely on Template Haskell to avoid much of the inconvenience of type-class encoding and provide a pliable user interface. The provided library has excellent performance, which can also been seen from our benchmarks in §4. The use of Template Haskell however significantly hinders the practical use of Handlers in Action. The present paper demonstrates that many of Kammar's et al. benefits can be attained in a simple to develop and to use library, staying entirely within Haskell.

The freer or freer-like monads have already appeared before, yet connecting all the dots took long time. The origins of freer monads can be traced to the pioneering work of Hughes [14], Claessen [8] and Hinze [13], who introduced and explained the term representation of effectful monadic computations. That representation was fully developed in the monad construction toolkit Unimo [23]:

```
data Unimo r a =
      Unit a
    | Effect (r (Unimo r) a)
    | ∀ b. Bind (Unimo r b) (b → Unimo r a)
```

It is quite close to FFree of §2.4, in particular, the Bind constructor whose second argument accumulates the continuation. Dedicating a variant Effect for effect requests proved to be a drawback, requiring the interpreter of Unimo r a monad to deal with two separate but very similar cases: Effect e and Bind (Effect e) k. One can think of free monads as eliminating this boilerplate – and throwing away the explicit continuation argument in the process. Our FFree brings the explicit continuation back. Unimo aimed to provide extensibility by emulating monad transformers. The Operational tutorial [1] introduces

```
data Program instr a where
  Then   :: instr  a → (a → Program instr b) → Program instr b
  Return :: a → Program instr a
```

which is exactly like our FFree. The tutorial correctly observed that Program instr is a monad (although without proof). Alas the paper mis-characterized Program as a GADT (it is not: it is a mere existential data type in GADT notation) and has not made the connection with the free monad. It is this connection that proves that Program instr really is a monad. More recently, Kammar's et al. also came within an inch of the freer monad: [17, Figure 5] contains the following definition

```
data Comp h a where
  Ret :: a → Comp h a
  Do :: (h `Handles` op ) e ⇒
     op e u → (Return (op e u ) → Comp h a ) → Comp h a
```

where Return is a type family and Handles is a three-parameter type class. It is very, very similar to FFree of §2.4, but with con-

[7] Since the user may write their own interpreter, they may well treat Reader as an exception, which is not safe. We may prevent such a behavior by not exporting the data constructor for the Reader request.

straints. The very similar data type, also with the constraints, appears in [27] as the data type NM ctx t a for constrained monad computations. Handlers in Action did not seem to have recognized that removing all the constraints gives a new algebraic data structure that is a monad by construction. The paper describes Comp in the traditional way: “the monad Comp h, which is simply a free monad over the functor defined by those operations op that are handled by h (i.e. such that (h ‘Handles ‘ op ) e is defined for some type e)”. FFree in §2.4 requires no functors and has no constraints or preconditions; it is a monad, period.

The papers [1] and [17] have noted the performance problem of free monads and attempted to overcome it with some sort of continuation passing – which works only up to the (reflection) point, as explained in [31]. The latter paper, which introduced type-aligned sequences, also applied them to speed up free monads. The implementation was quite complex, with the mutually recursive FreeMonad and FreeMonadView. It tried hard to fit the type-aligned sequences into the traditional 2free monads, rather than overcoming them. The main lesson of that paper – representing a conceptual sequence of binds as an efficient data structure – is expressed most clear in the new Eff monad in §3.

The recent ‘Handlers in scope’ [33] gives the more traditional introduction of extensible effects based on Data types à la carte. It also introduces the notion of a handler scope and two ways to support it. The underlying idea seems to be to run an effectful computation at the place of its handler, so to speak. The detailed investigation of the notion of scope deserves its own paper.

Compared to the right Kan extensions, left Kan extensions seem to have found so far fewer applications in functional programming. A notable application is Johann and Ghani’s [16], which used a specific form of left Kan extension (only with the equality GADTs) to develop the initial algebra semantics for GADTs.

## 9. Conclusions

We have rationally reconstructed the simplified and more efficient version of the extensible effects library and illustrated it with three new challenging applications: non-determinism, handling IO errors in the presence of other effects, and monadic regions. The new library is based on the freer monad, a more general and more efficient version of the traditional free monads. To improve efficiency we systematically applied the lesson of the left Kan extension: instead of performing an operation, record the operands in the data structure and pretend it done.

The ambition is for Eff to be the only monad in Haskell. Rather than defining new monads programmers will be defining new effects, that is, effect interpreters.

## Acknowledgments

We are very grateful to Marcelo Fiore and Shin-ya Katsumata for many helpful discussions, especially with regard to Kan extensions. We thank anonymous reviewers for many comments.

This work was partially supported by JSPS KAKENHI Grants 22300005, 25540001, 15H02681.

## References

[1] H. Apfelmus. The Operational monad tutorial. *The Monad.Reader*, 15:37–56, 2010.

[2] S. Awodey. *Category Theory*, volume 49 of *Oxford Logic Guides*. Oxford University Press, 2006.

[3] P. Bahr. Composing and decomposing data types: a closed type families implementation of data types à la carte. In *WGP@ICFP*, pages 71–82. ACM Press, 31 Aug. 2014.

[4] A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. arXiv:1203.1539 [cs.PL], 2012.

[5] J.-P. Bernardy and N. Pouillard. Names for free: polymorphic views of names and binders. In Haskell [11], pages 13–24.

[6] E. Brady. Programming and reasoning with algebraic effects and dependent types. In ICFP [15], pages 133–144. .

[7] R. Cartwright and M. Felleisen. Extensible denotational language specifications. In M. Hagiya and J. C. Mitchell, editors, *Theor. Aspects of Comp. Soft.*, number 789 in LNCS, pages 244–272. Springer, 1994.

[8] K. Claessen. Parallel parsing processes. *J. Functional Programming*, 14(6):741–757, 2004.

[9] O. Danvy and A. Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160. ACM Press, 27–29 June 1990.

[10] M. Fluet and J. G. Morrisett. Monadic regions. *J. Functional Programming*, 16(4–5):485–545, 2006.

[11] Haskell. *Haskell Symposium*, 2013. ACM.

[12] Haskell. *Haskell Symposium*, 2014. ACM.

[13] R. Hinze. Deriving backtracking monad transformers. In *ICFP ’00*, pages 186–197. ACM Press, 2000.

[14] J. Hughes. The design of a pretty-printing library. In *First Intl. Spring School on Adv. Functional Programming Techniques*, pages 53–96, London, UK, UK, 1995. Springer-Verlag.

[15] ICFP. *ICFP ’13*, 2013. ACM Press.

[16] P. Johann and N. Ghani. Foundations for structured programming with GADTs. In *POPL ’08*, pages 297–308. ACM Press, 2008.

[17] O. Kammar, S. Lindley, and N. Oury. Handlers in action. In ICFP [15], pages 145–158.

[18] O. Kiselyov. Iteratees. In *Proc. of the 11th International Symposium on Functional and Logic Programming*, pages 166–181, 2012.

[19] O. Kiselyov and C.-c. Shan. Lightweight monadic regions. In *Proc. 2008 Haskell Symposium*, pages 1–12. ACM Press, 2008.

[20] O. Kiselyov, C.-c. Shan, D. P. Friedman, and A. Sabry. Backtracking, interleaving, and terminating monad transformers (functional pearl). In *ICFP ’05*, pages 192–203. ACM Press, Sept. 2005.

[21] O. Kiselyov, A. Sabry, and C. Swords. Extensible effects: an alternative to monad transformers. In Haskell [11], pages 59–70. .

[22] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *POPL ’95*, pages 333–343. ACM Press, 1995.

[23] C.-k. Lin. Programming monads operationally with Unimo. In *ICFP ’06*, pages 274–285. ACM Press, 2006.

[24] C. Lüth and N. Ghani. Composing monads using coproducts. In *ICFP ’02*, pages 133–144. ACM Press, 2002.

[25] L. Naish. Pruning in logic programming. Technical Report 95/16, Department of Computer Science, University of Melbourne, 1995.

[26] G. Plotkin and M. Pretnar. Handlers of algebraic effects. In *ESOP ’09*, pages 80–94, Berlin, Heidelberg, 2009. Springer-Verlag.

[27] N. Sculthorpe, J. Bracker, G. Giorgidze, and A. Gill. The constrained-monad problem. In ICFP [15], pages 287–298.

[28] T. Sheard and E. Pašalić. Two-level types and parameterized modules. *J. Functional Programming*, 14(5):547–587, Sept. 2004.

[29] G. L. Steele, Jr. Building interpreters by composing monads. In *POPL ’94*, pages 472–492. ACM Press, 1994.

[30] W. Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, July 2008.

[31] A. van der Ploeg and O. Kiselyov. Reflection without remorse: revealing a hidden sequence to speed up monadic reflection. In Haskell [12], pages 133–144. .

[32] N. Wu and T. Schrijvers. Fusion for free: Efficient algebraic effect handlers. In *MPC 2015*, 2015. URL `/Research/papers/mpc2015.pdf`.

[33] N. Wu, T. Schrijvers, and R. Hinze. Effect handlers in scope. In Haskell [12], pages 1–12.