Extensible Effects
An Alternative to Monad Transformers

Oleg Kiselyov Amr Sabry Cameron Swords

Haskell Symposium 2013
Boston, MA Sep 23, 2013

We design and implement a library that solves the long-standing problem of combining effects without imposing restrictions on their interactions (such as static ordering). Effects arise from interactions between a client and an effect handler (interpreter); interactions may vary throughout the program and dynamically adapt to execution conditions. Existing code that relies on monad transformers may be used with our library with minor changes, gaining efficiency over long monad stacks. In addition, our library has greater expressiveness, allowing for practical idioms that are inefficient, cumbersome, or outright impossible with monad transformers.

Our alternative to a monad transformer stack is a single monad, for the coroutine-like communication of a client with its handler. Its type reflects possible requests, i.e., possible effects of a computation. To support arbitrary effects and their combinations, requests are values of an extensible union type, which allows adding and, notably, subtracting summands. Extending and, upon handling, shrinking of the union of possible requests is reflected in its type, yielding a type-and-effect system for Haskell. The library is lightweight, generalizing the extensible exception handling to other effects and accurately tracking them in types.

Monad Transformers

Monad Transformers and Modular Interpreters

Sheng Liang and Paul Hudak and Mark Jones
POPL 1995

"[1] Very recently, Cartwright and Felleisen [3] have independently proposed a modular semantics emphasizing a *direct* semantics approach, which seems somewhat more complex than ours; the precise relationship between the approaches is, however, not yet clear."

Effects in Haskell immediately evoke monad transformers, which were proposed, essentially in the same form we know now, in this classic, well-cited paper by Liang, Hudak and Jones from 1995. It is deservingly a classic paper.

Here is the first page of the paper. If we scroll down to the bottom of it, we see a curious footnote, mentioning a paper that hardly anyone knows or remembers. There was an alternative to monad transformers! This whole talk at its heart is about this footnote. The relationship between the two approaches has become clear, and the alternative can be presented much simpler than it was originally.

EDLS I

Extensible Denotational Language Specifications

Robert Cartwright, Matthias Felleisen

Theor. Aspects of Computer Software, 1994

`http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.25.5941`

The schema critically relies on the distinction between a complete program and a nested program phrase. A complete program is thought of as an agent that interacts with the outside world, e.g., a file system, and that affects global resources, e.g., the store. A central authority administers these resources. The meaning of a program phrase is a *computation*, which may be a value or an *effect*. If the meaning of a program phrase is an effect, it is propagated to the central authority. The propagation process adds a function to the effect package such that the central authority can resume the suspended calculation. We refer to this component of an effect package as the *handle* since it provides access to the place of origin for the package.

Here is this paper presenting the alternative approach. Curiously, it was published roughly at the same time as the Monad Transformers paper, just a couple of months earlier. Research in effects seems to be filled with amazing coincidences.

The paper, especially the introduction, is very inspirational. I strongly recommend reading it. Let me show a few parts of the paper that directly inspired and are reflected in the work presented now.

Cartwright and Felleisen were interested in extensible denotational specifications and hence building extensible interpreters. They really did it, starting from the language with only two constructs, for looping and failure. They add boolean algebra, arithmetic, call-by-name or by-value, state, etc. – as extensions. You have probably read the highlighted paragraph. You see the word 'handle' (and then 'handler') that you'll hear again in this talk and in several ICFP talks. These term was coined in 1994.

Our effect library pretty much reflects the quoted Cartwright and Felleisen's description.

Main ideas

1. "An effect is most easily understood as an interaction between a sub-expression and a central authority that administers the global resources of a program."
2. "an effect can be viewed as a message to the central authority plus enough information to resume the suspended calculation"
   the program and its authority are like co-routines

3. the authority is part of the program, distributed throughout it: *bureaucracy*
   - modularity and extensibility
   - encapsulation of effects
4. *types* approximate possible effects, unordered collection: type-and-effect system

Let me summarize the main ideas. Cartwright and Felleisen propose to view effect as an interaction, communication with an authority in charge of resources. The interacting authority and the program may be viewed as coroutines.

The last two points are our departure from Cartwright and Felleisen. When the authority is part of user program and distributed throughout it, it is easier to add new effects (compared to the extension of the monolithic central authority) and it becomes possible to encapsulate effects, limiting them to parts of the program. Also unlike EDLS, we use types – as a conservative approximation of effects an expression may perform. We thus get a type and effect system.

If you understand these ideas, you can write your own library of extensible effects. Let me briefly describe how we did it.

Outline

- Limits of monad transformers
- Extensible effects, the library
- Extensible effects: beyond the limits of MTL
- Frequently Expressed Confusions
  - Isn't this just "Data types à la carte"?
  - Isn't this just a Free Monad?
  - Typeable is ugly! OverlappingInstances are ugly!
  - Extensible effects vs Algebraic effects?
  - OCaml polymorphic variants vs OpenUnion?
- Concluding remarks

Looking at alternatives is worthwhile in and of itself. Yet some may say, if Monad Transformers work so well, what's the problem then? We show a simple example, implemented with the monad transformer library (MTL): to remind of monad transformers and to demonstrate they do have a problem. We then demonstrate *one* implementation of the main ideas of the alternative approach, which we just saw. The alternative looks and feels very much like MTL, but goes beyond its limits.

Running example

Given

comp :: **Monad** m ⇒ m Int

Tasks

1. if the result is too big, throw TooBig exception
2. recover from that exception

Here is the specification of our running example, to illustrate monad transformers and their limitations, and extensible effects' overcoming them. Suppose we have a computation in some monad m producing an Int. First, we check the result against a threshold. If it's bigger, throw the exception TooBig. In the second part of the example, we should recover from the exception.

Running example in MTL 1

```
newtype TooBig = TooBig Int deriving Show

ex2 :: MonadError TooBig m ⇒ m Int → m Int
ex2 m = do
  v ← m
  if v > 5 then throwError (TooBig v) else return v
```

Here is the most elegant implementation with the Monad Transformer Library (MTL). We define the data type to use as an exception. We throw the TooBig value if the result of the argument computation m is bigger than our threshold, 5. The use of throwError induces the constraint MonadError: the monad m must be an error monad. The code is intuitive, elegant and clearly express our intention. Let's see how it works.

Running example in MTL 2

```
ex2 :: MonadError TooBig m ⇒ m Int → m Int
ex2 m = do
  v ← m
  if v > 5 then throwError (TooBig v) else return v

choose :: MonadPlus m ⇒ [a] → m a

runListT  :: ListT m a → m [a]
runErrorT :: ErrorT e m a → m (Either e a)
```

Fixing the order

```
runIdentity ∘ runListT ∘ runErrorT $ ex2 (choose [5,7,1])
```

Our guard ex2 should work with any computation. As a computation to guard, we take non-deterministic choice of an integer out of three: 5, 7 or 1. We need a MonadPlus monad to program such choice. We can now try running the example. But we need to choose a monad that has the properties of being an error monad and a non-determinism monad. In MTL, monads with desired properties are composed by layering of monad transformers responsible for a single property. For example, ListT applied to some m makes the monad that supports non-determinism (a member of the type class MonadPlus). The function runListT 'peels off' the layer, returning all choices in a list. ErrorT e adds supports for throwing exceptions of type e. We must fix the order of layers though. It is fixed by choosing the order of the run functions. Here, the order is: Identity transformed by ListT transformed by ErrorT. The order remain fixed throughout the computation.

Running example in MTL 2

```
ex2 :: MonadError TooBig m ⇒ m Int → m Int
ex2 m = do
  v ← m
  if v > 5 then throwError (TooBig v) else return v

choose :: MonadPlus m ⇒ [a] → m a

runListT  :: ListT m a → m [a]
runErrorT :: ErrorT e m a → m (Either e a)
```

Fixing the order

```
runIdentity ∘ runListT ∘ runErrorT $ ex2 (choose [5,7,1])
-- [Right 5]
```

The result is quite surprising. We see no shred of the TooBig exception, which was supposed to be thrown. Also, the choice 1 is gone. It is a puzzle indeed. It is an exercise to the audience to determine what is going on.

Running example in MTL 2

```
ex2 :: MonadError TooBig m ⇒ m Int → m Int
ex2 m = do
  v ← m
  if v > 5 then throwError (TooBig v) else return v

choose :: MonadPlus m ⇒ [a] → m a

runListT  :: ListT m a → m [a]
runErrorT :: ErrorT e m a → m (Either e a)
```

Fixing the order

```
runIdentity ∘ runListT ∘ runErrorT $ ex2 (choose [5,7,1])
-- [Right 5]

runIdentity ∘ runErrorT ∘ runListT $ ex2 (choose [5,7,1])
-- Left (TooBig 7)
```

What if we choose the different order of layers? This time, we see the exception thrown, as expected. So, the order of layers matters a great deal. Well, at least one order of layers has worked for us. But we are not done yet. There is the second part of the example: error recovery.

Running example in MTL 3

```
ex2 :: MonadError TooBig m ⇒ m Int → m Int
ex2 m = do
  v ← m
  if v > 5 then throwError (TooBig v) else return v

exRec :: MonadError TooBig m ⇒ m Int → m Int
exRec m = catchError m handler
 where handler (TooBig n) | n ≤ 7 = return n
       handler e = throwError e
```

The MonadError type class offers not only throwError to throw errors but also catchError to handle them. We can catch the TooBig exception and examine it. If the result is really not that big, not exceeding 7, we then continue the execution, recovering from the error. Otherwise, we re-throw the error. Let us see how it works, using the winning order of the layers.

Running example in MTL 3

```
ex2 :: MonadError TooBig m ⇒ m Int → m Int
ex2 m = do
  v ← m
  if v > 5 then throwError (TooBig v) else return v

exRec :: MonadError TooBig m ⇒ m Int → m Int
exRec m = catchError m handler
 where handler (TooBig n) | n ≤ 7 = return n
       handler e = throwError e

runIdentity ∘ runErrorT ∘ runListT $ exRec (ex2 (choose [5,7,1]))
-- Right [7]
```

Wanted: per world exception and recovery

Well, it doesn't work too well. The original computation had three non-deterministic choices. Two of them should not trigger any error, and one triggers the TooBig exception, which is supposed to be recovered from. So, the result should have three choices, but we got only one. The other two got lost. We would really wanted that exception did not interfere with choices, was limited to its possible world. We tried the two possible ordering of the monad transformer layers and could not get what we wanted, per-world exception and recovery. Simple variations lead to the similar disappointment.

Actually we are lucky: it is possible to implement our example, with per world exception and recovery, using MTL. But we need *two* error layers. To see how easy or intuitive it is, I encourage the audience to write that code.

MTL Problems

- Fixed order of layers throughout the computation: limited expressivity
- Complexity (quadratic) of lifting
- Performance degradation on large transformer stacks

So, monad transformers do have problems. The principal problem is the fixed order of monad transformer layers. The order matters, since it defines how effects, exceptions and non-determinism, interact. If the order is fixed throughout the whole computation, we cannot express computations that require more flexible interaction. We were lucky that we could after all implement our running example as intended, albeit unintuitively and with performance degradation. The paper outlines a different example, with coroutines and dynamic binding, the Reader monad. There does not seem to be any way to implement that example in its full form with MTL, by composing independent layers of effects. There are other problems of MTL, please see the paper for their discussion.

Alternative: Extensible Effects

- effect as a communication with an authority
  the program and its authority are like coroutines
- the authority is part of the program: bureaucracy
- types approximate possible effects

Let's recall what've learned from the Cartwright and Felleisen's paper, and from contemplating extensions to it. Let us see how we can implement them in Haskell.

Alternative: Extensible Effects

- effect as a communication with an authority
  the program and its authority are like coroutines

```
type Eff r a
instance Monad (Eff r)
run :: Eff Void w → w
send_req :: (Suspension a → Request) → Eff r a
```

- the authority is part of the program: bureaucracy
- types approximate possible effects

First, we need coroutines. Let's pick a monadic implementation of coroutines out of several available, and call it monad Eff r. We will come to this r parameter of the monad in a moment. A coroutine monad will have an operation to run it; let's call it run. Again, please disregard the void for a moment. A coroutine monad obviously has an operation for a coroutine to resume its caller, and suspend. Let's call this operation send_req. Its argument is a function that takes a suspension, the 'return address', and incorporates it into the request; send_req will then send the request and suspend. When the coroutine resumes, it receives the value of the type a and continues.

Alternative: Extensible Effects

- effect as a communication with an authority
  the program and its authority are like coroutines

```
type Eff r a
instance Monad (Eff r)
run :: Eff Void w → w
send_req :: (Suspension a → Request) → Eff r a
```

- the authority is part of the program: bureaucracy

```
data VE w r = Val w | E (Union r (VE w r)) -- cf. free monad
admin :: Eff r w → VE w r                  -- cf. try
handle_relay :: Union (req ▷ r) v →
    (v → Eff r a) → (req v → Eff r a) → Eff r a
```

- types approximate possible effects

The authority, bureaucracy, is part of the program. So, we should be able to program it as well. First, the bureaucrat needs to know if the supervised program fragment has finished, with the value w, or has sent a request. Recall, Cartwright and Felleisen wrote: "The meaning of a program phrase is a *computation*, which may be a value or an *effect*." The function admin, which should be provided by the coroutine library, does this. Recall that bureaucracy is distributed: each particular bureaucrat, or handler, is responsible only for one or a couple of specific requests. A program may perform several effects, that is, send several requests. Thus a bureaucrat has do determine if the request of the type it can handle. If not, the request should be relayed upstairs. The function handle_relay does the case analysis. It checks if the current request, contained in the union of possible request, is of the desired type `req`. If so, it calls the supplied handler (the last argument of the function). The function handle_relay is provided by the library of open unions.

Alternative: Extensible Effects

- effect as a communication with an authority
  the program and its authority are like coroutines

```
type Eff  r  a
instance Monad (Eff r)
run :: Eff  Void w → w
send_req :: (Functor req,  Member req r) ⇒
            (∀ w. (a → VE w r) →  req (VE w r)) → Eff  r  a
```

- the authority is part of the program: bureaucracy

```
data VE w r = Val w | E (Union r (VE w r)) -- cf. free monad
admin :: Eff  r  w → VE w r                -- cf. try
handle_relay  :: Union ( req ▷ r) v →
     (v → Eff  r  a) → (req v → Eff  r  a) → Eff  r  a
```

- types approximate possible effects
  handled effects are subtracted from the type
  assure: there is a handler for each request

Finally, we need types that tell which effects a program fragment may possibly do. Here is where that r parameter comes in. The coroutine monad is indexed by r, the type of possible requests. One may think of this type as a set of possible requests. Void denotes the empty set. Therefore, we can run only uneffectful computations, whose effects are all handled. The notation req ▷ r denotes a set with the element req added to r. Therefore, when the bureaucrat handles `req`, the computation no longer has that req (see the handle_relay.) In other words, handled effects are subtracted from the effect type.
Here is the full type of `send_req`: it shows that the request req we are sending has to be a member of the set r. That is, the whole program must have a bureaucrat that can handle the request.

That is all there is to it. We can re-implement the whole MTL, and we can do more.

Extensible Effects: Error effect

```
newtype Exc e v = Exc e

throwError :: (Member (Exc e) r) ⇒ e → Eff r a
throwError e = send_req (const $ Exc e)

runError :: Eff (Exc e ▷ r) a → Eff r (Either e a)
runError m = loop (admin m)
 where
 loop (Val x)   = return (Right x)
 loop (E u)     = handle_relay u loop (\(Exc e) → return (Left e))

catchError :: (Member (Exc e) r) ⇒
          Eff r a → (e → Eff r a) → Eff r a
```

No need for MonadError

Let's implement the effect of throwing an exception (of type `e`) in our framework. `Exc e v` is the type of the request, with `v` being the result to be produced by a suspension. Since the program, after requesting an exception, is not resumed, we ignore `v`. Throwing an exception makes the request. The interpreter, the bureaucrat, runError returns the result of the supervised computation tagged with Right. If the computation requested an exception, the exception value `e` is returned, without resuming the computation. If the computation made some other request, it is relayed upstairs. After the higher bureaucrat replied, we continue the supervision.

Our library implements this Error effect and other MTL effects.

We no longer need to define a type class per effect. In MTL, it hid the ordering of the layers. Now, the ordering is hidden anyway, by the constraint Member.

Back to the running example 1

```
newtype TooBig = TooBig Int deriving (Show, Typeable)

ex2 :: Member (Exc TooBig) r ⇒ Eff r Int → Eff r Int
ex2 m = do
  v ← m
  if v > 5 then throwError (TooBig v) else return v

exRec :: Member (Exc TooBig) r ⇒ Eff r Int → Eff r Int
exRec m = catchError m handler
 where handler (TooBig n) | n ≤ 7 = return n
       handler e = throwError e
```

Let's come back to our running example and re-implement it with extensible effects.

The code looks exactly the same as it was with MTL. Only type signatures differ. The type signatures are all inferred.

Back to the running example 2

```
ex2    :: Member (Exc TooBig) r ⇒ Eff r Int  → Eff  r  Int

ex2 (choose [5,7,1])
   :: (Member Choose r, Member (Exc TooBig) r) ⇒ Eff r Int
```

We see the encapsulation of effects. As we add handlers, the type becomes smaller as effects are handled.

Back to the running example 2

```
ex2    :: Member (Exc TooBig) r ⇒ Eff r Int  → Eff  r  Int

runErrBig :: Eff  (Exc TooBig ▷ r)  a → Eff  r  (Either  TooBig a)
runErrBig  m = runError m

ex2 (choose [5,7,1])
   :: (  Member Choose r, Member (Exc TooBig) r) ⇒ Eff r Int

runErrBig  $  ex2 (choose [5,7,1])
   ::   Member Choose r ⇒ Eff r (Either  TooBig Int)
```

Back to the running example 2

```
ex2    :: Member (Exc TooBig) r ⇒ Eff r Int  → Eff  r  Int
runErrBig :: Eff  (Exc TooBig ▷ r)  a → Eff  r  (Either  TooBig a)

makeChoice :: Eff  (Choose ▷ r)  a → Eff  r  [a]

ex2 (choose [5,7,1])
  :: (Member Choose r, Member (Exc TooBig) r) ⇒ Eff r Int

 runErrBig $ ex2 (choose [5,7,1])
  :: Member Choose r ⇒ Eff r (Either  TooBig Int)

 makeChoice ∘ runErrBig $ ex2 (choose [5,7,1])
  :: Eff  r  [Either  TooBig Int]

 run ∘ makeChoice ∘ runErrBig $ ex2 (choose [5,7,1])
[Right 5, Left  (TooBig 7),Right 1]
```

When all effects are handled, we can finally run the computation. Of three choices, one ended in error, the others normally. There are no surprises.

Back to the running example 3

```
exRec :: Member (Exc TooBig) r ⇒ Eff r Int → Eff r Int

run ∘ runErrBig ∘ makeChoice $ exRec (ex2 (choose [5,7,1]))
-- Right [5,7,1]

run ∘ makeChoice ∘ runErrBig $ exRec (ex2 (choose [5,7,1]))
-- [Right 5,Right 7,Right 1]
```

Quickly, error recovery. According to our thresholds, 7 is too big but not too big. So, when it comes to 7, an exception should be raised, and should be recovered from. That's exactly the result we get. There are no lost choices any more. If we switch the order of the handlers, the return type obviously changes, but the overall result stays the same. The exception is handled, no choices gone missing. There are no surprises.

Choice effect

```
data Choose v = ∀a. Choose [a] (a → v)
                deriving (Typeable)

choose :: Member Choose r ⇒ [a] → Eff r a
choose lst  = send (\k → inj $ Choose lst k)

mzero' :: Member Choose r ⇒ Eff r a
mzero' = choose []
mplus' m1 m2 = choose [m1,m2] >>= id
```

We have time to see the implementation of the Choice effect, with the choose request, which expresses the familiar MonadPlus operators mzero and mplus.

Choice effect: handler

```
makeChoice :: ∀ a r. Eff (Choose ▷ r) a → Eff r [a]
makeChoice m = loop (admin m)
 where
 loop (Val x)   = return [x]
 loop (E u)     = handle_relay u loop (\(Choose lst k) → handle lst

 handle :: [t] → (t → VE a (Choose ▷ r)) → Eff r [a]
 handle []  _   = return []
 handle [x] k = loop (k x)
 handle lst  k = fmap concat $ mapM (loop ∘ k) lst
```

The handler of choose requests implements DFS. One can see the similarity with the list monad: cf. concatMap

Choice effect: Soft cut (LogicT)

Non-deterministic if-then-else, aka Prolog's `*->`

$$\mathsf{ifte}\ \ \mathsf{t}\ \ \mathsf{th}\ \ \mathsf{el}\ \equiv\ (\mathsf{t}\ \gg\!=\ \mathsf{th})\ \text{`}\mathsf{mplus}\text{`}\ ((\,\mathsf{not}\ \ \mathsf{t}\,)\ \gg\ \mathsf{el}\,)$$

but t is evaluated only once

The choice effect implements not just the MonadPlus interface but also LogicT, in particular, soft cut. That is, `ifte t th el` is equivalent to `t >>= th` if `t` has at least one solution. If `t` fails, `ifte t th el` is the same as `el`. To implement soft cut, we should sort of look-ahead into the test `t` to see it succeeds at least once. Hence we need to convert a computation to its representation, so we can look into it.

Choice effect: Soft cut (LogicT)

Non-deterministic if-then-else, aka Prolog's `*->`

$$\mathsf{ifte}\ \ \mathsf{t}\ \ \mathsf{th}\ \ \mathsf{el}\ \equiv\ (\mathsf{t}\ \gg\!=\ \mathsf{th})\ \text{`}\mathsf{mplus}\text{`}\ ((\mathsf{not}\ \mathsf{t})\ \gg\ \mathsf{el}\,)$$

but t is evaluated only once

Monadic reification

```
admin :: Eff r w → VE w r
admin (Eff m) = m Val
```

But we had been doing it all along: admin. More technically, it is called reification.

Choice effect: Soft cut (LogicT)

Non-deterministic if-then-else, aka Prolog's `*->`

```
ifte t th el ≡ (t >>= th) `mplus` ((not t) >> el)
```

but t is evaluated only once

Monadic reification

```
admin :: Eff r w → VE w r
admin (Eff m) = m Val
```

Monadic reflection

```
reflect :: VE a r → Eff r a
reflect (Val x) = return x
reflect (E u) = Eff (\k → E $ fmap (loop k) u) where
 loop :: (a → VE w r) → VE a r → VE w r
 loop k (Val x) = k x
 loop k (E u)   = E $ fmap (loop k) u
```

Monadic reification and reflection are generic

For soft-cut, we also need the inverse operation: We have disassembled `t` and 'looked inside' and saw if it fails or not. If we found it does not fail, we have to put it back together, to implement `t >>= th`. We need so-called reflection. It is also implementable. The shown implementation is not efficient; that can be fixed. What interesting is that normally reification and reflection require separate implementation per each effect. With extensible effects, they are written generically.

Choices

- how to implement coroutines
- use OverlappingInstances and Typeable
- use closed type families instead of OverlappingInstances
- use neither OverlappingInstances nor Typeable
- allow or prohibit the duplication of effect descriptors in types
- allow or prohibit different types of monad readers
- ...

The design space is quite large. For example, there are several ways to implement coroutines and sending of messages; there are several ways to implement open unions. We explored a few options, shown on the slide; more remain.

In our implementation, Open unions it is a *true* union: union of `a` and `a` is `a`.

Frequently Expressed Confusions

- Isn't this just "Data types à la carte"?
- Isn't this just a Free Monad?
- Typeable is ugly! OverlappingInstances are ugly!
- Extensible effects vs Algebraic effects?
- OCaml polymorphic variants vs OpenUnion?

There has been a fair amount of discussion of extensible effects on various forums like reddit. Alas, not all of it has been well-informed. Quite a bit of misunderstanding was expressed. Let me try to clear it.

Isn't this just "Data types à la carte"?

Similarities

- Requests and their interpreters
- Extensibility: more requests
- Open Unions

Differences

- lack the encapsulation of effects
- lack effect inference

The biggest confusion is about extensible effects and Swierstra's data types à la carte. There are many similarities, to be sure. One of them is open unions – which have been implemented already in Liang et al. MTL paper. This is not surprising: any extensible interpreter has to have open unions for terms it interprets. The EDSL paper has open unions too.

Isn't this just "Data types à la carte"?

```
data Incr  t    = Incr  Int  t
data Recall  t  = Recall  (Int  → t)

incr  :: (Incr  :<: f  ) ⇒ Int  → Term f ()
incr  i  = inject  (Incr  i  (Pure ()))
recall  :: (Recall  :<: f  ) ⇒ Term f Int
recall  = inject  (Recall  Pure)
```

This is an example from the “Data types à la carte” paper, the implementation of a State effect with two operations: to increment the integer state and to get it. Defining and sending of the requests is essentially the same as with extensible effects.

Isn't this just "Data types à la carte"?

```
newtype Mem = Mem Int
class  Functor f  ⇒ Run f where
      runAlgebra :: f  (Mem → (a, Mem)) → (Mem → (a, Mem))

instance Run Incr where
  runAlgebra (Incr  k r) (Mem i ) = r (Mem (i + k ))
instance Run Recall where
  runAlgebra (Recall  r) (Mem i ) = r i  (Mem i )
instance (Run f ,  Run g) ⇒ Run (f :+: g) where
  runAlgebra (Inl  r) = runAlgebra r
  runAlgebra (Inr  r) = runAlgebra r
```

Lack of encapsulation

- Type class Run is global (and unnecessary)

This is the handler part: and it is very different. With extensible effects, we never had to define a type class for a handler, had we?

Isn't this just "Data types à la carte"?

```
newtype Mem = Mem Int
class Functor f ⇒ Run f where
    runAlgebra :: f (Mem → (a, Mem)) → (Mem → (a, Mem))

run :: Run f ⇒ Term f a → Mem → (a, Mem)
run = foldTerm (,) runAlgebra
```

Lack of encapsulation

- Type class Run is global (and unnecessary)
- run handles all effects rather than some of them
- Term f is extensible but not shrinkable

The main difference from extensible effects is in the signature of `run`: its return type is not a `Term f' a`. That is, `run` is a complete interpreter, rather than a handler of some requests, letting other handlers do their part. There is no effect encapsulation. Correspondingly, open unions in the "Data types à la carte" paper do not support the decomposition operation.

Isn't this just a Free Monad?

```
data VE w r = Val w | E (Union r (VE w r))
instance Functor (Union r) where
```

Free Monad is an accident

- The bind on VE w r is never used
- There are other implementations, without Functor

If we look carefully at the type VE, we see that VE w r with the flipped arguments is a free monad, over a functor Union r. This is somewhat of an accident. A handler needs to rethrow *any* request it does not understand. One way of implementing it is for the handler to forward the request to a handler upstairs, and then relay its reply. For the latter, the handler needs to send a reply to a request of an unknown form. The Functor constraint is enough to ensure that. In this design, the handler is able to do finalization for any request, even the ones it does not understand. However, the request relay may be implemented differently, without the functor constraint. In that case, the handler can do finalization only for known requests.

Typeable is ugly! OverlappingInstances are ugly!

```
module OpenUnion1 ... where

data Union r v where
  Union :: (Functor t, Typeable1 t) ⇒ Id (t v) → Union r v

class Member (t :: ∗ → ∗ ) r
instance Member t (t ▷ r)
instance Member t r ⇒ Member t (t' ▷ r)
```

However,

- other implementations of open unions use neither Typeable nor OverlappingInstances
- OverlappingInstances in Member: no closed type families until GHC 7.8
- These are all implementation details

You may be wondering what Typeable and OverlappingInstances I am talking about. They appear in one implementation of open unions, OpenUnion1, which I have not shown you. Here I have to reveal the details. I did not show you the implementation deliberately: there are other implementations of open unions, which use neither feature. Also, OverlappingInstances were the artifact: GHC 7.8 released the other day supports closed type families, and the overlapping instances are no longer needed. So the main message: there is nothing in extensible effects that needs these features. Let's not focus on implementation details.

Extensible effects vs Algebraic effects?

The general coroutine effect

```
data Yield a b v = Yield a (b → v) deriving (Typeable, Functor)

yield :: (Typeable a, Member (Yield a b) r) ⇒ a → Eff r b
yield x = send (inj ∘ Yield x)

data Y r a b = Done | Y a (b → Eff r (Y r a b))

runC :: Typeable a ⇒ Eff (Yield a b ▷ r) w → Eff r (Y r a b)
runC m = loop (admin m) where
 loop (Val x) = return Done
 loop (E u)   = handle_relay u loop $
                  \(Yield x k) → return (Y x (loop ∘ k))
```

General delimited continuation effects are not algebraic

Here is a simple implementations of generalized coroutines, which can be resumed more than once. Such coroutines are equivalent in expressive power to delimited control. Delimited control effects are not algebraic. So, extensible effects easily deals with non-algebraic effects.

OCaml polymorphic variants vs OpenUnion?

- Tags do not have to be declared, extensible type

```
# `ta 1
- : [> `ta of int ] = `ta 1
```

OCaml polymorphic variants vs OpenUnion?

- Tags do not have to be declared, extensible type
- Order does not matter

```
# fun x → if x then `ta 1 else `tb 2.0
- : bool → [> `ta of int | `tb of float ] = <fun>

# let f = fun x → if x then `tb 2.0 else `ta 1
val f : bool → [> `ta of int | `tb of float ] = <fun>
```

OCaml polymorphic variants vs OpenUnion?

- Tags do not have to be declared, extensible type
- Order does not matter
- Static check: all possibilities are handled

```
# let f = fun x → if x then `tb 2.0 else `ta 1
val f : bool → [> `ta of int | `tb of float ] = <fun>

# fun x → match f x with `ta x → x;;
Error: This pattern matches values of type [< `ta of 'a ]
       but a pattern was expected which matches values of type
         [> `ta of int | `tb of float ]
       The first variant type does not allow tag(s) `tb
```

OCaml polymorphic variants vs OpenUnion?

- Tags do not have to be declared, extensible type
- Order does not matter
- Static check: all possibilities are handled
- Alas: pattern-match doesn't remove variants
  No encapsulation

```
# let f = fun x → if x then `tb 2.0 else `ta 1
val f : bool → [> `ta of int | `tb of float ] = <fun>

# fun x → match f x with `ta (y:int) → None | y → Some y;;
- : bool → [> `ta of int | `tb of float ] option = <fun>
```

Conclusions

A framework for extensible effects

- modularity, extensibility, encapsulation of effects
- type-and-effect system
- handling effects individually or in groups, interleaving
- subsumes MTL

Effect is an interaction

Think in terms of desired effects rather than MTL layers

what effect we want to achieve rather than which monad transformer to use

We have presented an extension of Cartwright and Felleisen approach to modular effect handling, and described one of its implementation in Haskell. Our framework induces a type and effect system for Haskell. It subsumes MTL: it can be used quite like MTL but offers flexibility of effect handling and interaction, without fixed lifting.
The take-away is the view of effects as interaction, between an expression and the interpreter. And the other thought: when designing a program we should start thinking what effect we want to achieve rather than which monad transformer to use. Using ReaderT or StateT or something else is an implementation detail. Once we know what effect to achieve we can write a handler, or interpreter, to implement the desired operation on the World, obeying the desired equations. And we are done.
Defining a new class for each effect is possible but not needed at all. With monad transformers, a class per effect is meant to hide the ordering of transformer layers in a monad transformer stack. Effect libraries abstract over the implementation details out of the box. Crutches – extra classes – are unnecessary.
Some papers become popular, some other, just as inspirational, sink to obscurity. I'm happy have a chance to bring attention to Cartwright and Felleisen's paper, which should be read and remembered.