Extensible Effects

An Alternative to Monad Transformers

Oleg Kiselyov
oleg@okmij.org

Amr Sabry
Indiana University, USA
sabry@indiana.edu

Cameron Swords
Indiana University, USA
cswords@indiana.edu

Abstract

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.

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 monad, monad transformer, effect handler, open union, type and effect system, effect interaction, coroutine

1. Introduction

From the early days of the introduction of monads to the world of functional programming [22], it was understood that monads, generally, do *not* compose [14, 29, 34]. A variety of 'monad compositions' were investigated based on Moggi's idea of "monad morphisms" [23]. Several initial designs [4–6, 29, 30] were extended by Liang's et al. [18] in what has become the current state of the art for Haskell: the "monad transformers library" (MTL) (`mtl-2.1.2`).

Monad transformers is a framework to let the programmer assemble monads through a number of transformers, producing a combined effect consisting of "layers" of elementary monadic effects. As effectful operations may combine in different ways, at each layering the programmer manually expresses how the operations of the underlying, or base, monad 'lift' through the current transformer. (This idea is also investigated from a different perspective and in more depth by Filinski [7, 8].) In the standard implementation of monad transformers, each layer has overhead that adds up over long stacks (see §4 for discussion). In addition, the layering is determined statically and cannot easily be altered dynamically in different parts of the program. Most importantly, some practical situations require the effects to be *interleaved*, wherein no complete static layering of one effect over the other provides the desired semantics.

Thus, despite its popularity, the framework of monad transformers is fundamentally limited. (We review, with examples, these limitations in §5.) Alternative approaches to monad transformers exist [12, 19]; the most relevant to our work is the "extensible denotational language specifications" (EDLS) approach of Cartwright and Felleisen [3] that was developed at around the same time as the original paper on monad transformers [18]. The basic idea is to model an effect as an *interaction*: a program fragment that wishes to update a mutable variable, throw an exception, or write to a file sends a 'request' to an 'authority.' The request describes the action to perform and contains the 'return address,' a continuation, to resume the requester. In the original EDLS framework, this authority, i.e., the interpreter of requests as transformations on resources, is not part of the user program (just as the the operating system kernel is not part of the user's process, and the interpreter of IO actions in Haskell is not part of the user program). This global external authority is in charge of all the resources (files, memory, etc.): it interprets a request, and may either execute it on the requester's behalf and then continue the requester, passing to it the result – or decline to answer, thus aborting the requester. The fundamental benefit of this approach is that the order of composing effects, when it is irrelevant, does not need to be specified at all, and there is no need to commit to a single, statically determined order of effect composition. The limitations are: (i) the global external authority is difficult to extend, (ii) effects are not encapsulated, and (iii) effects of a computation are not reflected in its type.

Partially inspired by free monads [32] and the term algebra approach of Hughes and Hinze [9, 10], we address the three problems above:

- We replace the central, inflexible authority by a distributed, automatically extensible, "bureaucracy" that is part of the user program. Following the work on algebraic handlers [1, 25], we call each partial authority (that controls *some* resources and interprets *some* requests) a *handler*. Each such handler is both the authority for its client part of the program and a client itself:

Haskell '13, September 23–24, 2013, Boston, MA, USA.

ACM 978-1-4503-2383-3/13/09. . . $15.00.
http://dx.doi.org/10.1145/2503778.2503791

if a handler receives a request it does not understand, it relays the request to an "upstream" handler.

- Second, and more importantly, we develop an expressive type-and-effect system that keeps track of which effects are currently active in a computation. This system maintains an open union (a type-indexed coproduct of functors) containing an unordered collection of current effects. The action of each handler is reflected in the type by removing the effects that have been handled, and thus the type system can guarantee that a full program does not contain "dangling effects."

Our primary contributions are:

- a user-level effect framework modeled after the MTL syntax that allows effects to be combined in any order and even *interleaved* in ways that are impossible in the standard monad transformer approach; the system is realized as a Haskell library using common extensions: developers may immediately start using it with minimal syntactic changes to existing programs;
- a detailed analysis of the expressiveness problems of monad transformers that cause subtle bugs in real programs;
- a novel implementation of an extensible, constant-time, open union type facility that sits at the center of the effect handler system;
- a system built around the new open union facility that provides mechanisms for adding, using, and removing effects dynamically across programs, yielding an effect system far more expressive than the current Haskell approach based on the MTL.

To summarize, the resulting design improves on both monads transformers and EDLS, facilitating more flexible interactions of effects that are accurately tracked by the type system. The complete Haskell implementation along with illustrations and warm-up examples can be found at `http://okmij.org/ftp/Haskell/extensible/`. We shall refer to this code throughout the paper, quoting relevant excerpts.

The remainder of the paper is structured as follows. We begin with a high-level "tour" of our extensible effects framework that introduces its programmer-level interface with a variety of small examples (§2). The next section (§3) provides the full semantics and explains the key components of the implementation. Then, §4 confirms that our design can simulate the full MTL and §5 uses several advanced examples to illustrate that our design goes beyond the MTL in expressiveness. We conclude after a comparison with related work.

2. A Tour of the Extensible Effects Framework

We start with a few examples to give a feel for our library of extensible effects and demonstrate its use. (The complete code `Eff.hs` accompanies the paper.) The library is implemented on top of a tiny core described in §3.4 and is designed to look like the MTL [24, Chap. 18] familiar to every Haskell programmer. This MTL-like interface is presented in Fig. 1. The two main points of interest are that (i) all effectful computations are represented by a monad Eff r, and that (ii) the type parameter r is an *open union* of individual effects whose components must be Typeable. This r may intuitively be thought of as the *set* of effects that computations may perform. We shall see later that effects are represented by requests, hence the name r.

There are two basic ways of indicating that an effect m is part of this open union r.[1] The first is via a type constraint (Member m r); this indicates that the effect m is an element of the set r. The second is via an explicit pattern (m ⊳r') that decomposes the union r into the effect m and the remaining effects r'; this is analogous to $\{m\} \cup r'$. We may think of Void as $\emptyset$, i.e., a computation of type (Eff Void a) is pure. In contrast, a computation of type (Eff (Reader Int ⊳Reader Bool ⊳Void) a) may access *two* environments: one of type Int and one of type Bool.

```
instance Monad (Eff r)

-- Pure computations
data Void
run :: Eff Void w → w

-- Reader (or environment) effect
type Reader e
ask       :: (Typeable e, Member (Reader e) r) ⇒ Eff r e
local     :: (Typeable e, Member (Reader e) r) ⇒
             (e → e) → Eff r w → Eff r w
runReader :: Typeable e ⇒
             Eff (Reader e ⊳ r) w → e → Eff r w

-- Exceptions
type Exc e
throwError :: (Typeable e, Member (Exc e) r) ⇒ e → Eff r a
catchError :: (Typeable e, Member (Exc e) r) ⇒
              Eff r w → (e → Eff r w) → Eff r w
runError   :: Typeable e ⇒
              Eff (Exc e ⊳ r) w → Eff r (Either e w)

-- State
type State s
get      :: (Typeable s, Member (State s) r) ⇒ Eff r s
put      :: (Typeable s, Member (State s) r) ⇒ s →
Eff r ()
runState :: Typeable s ⇒
            Eff (State s ⊳ r) w → s → Eff r (w,s)

-- Non-determinism
type Choose
choose     :: Member Choose r ⇒ [w] → Eff r w
makeChoice :: Eff (Choose ⊳ r) w → Eff r [w]

-- Tracing
type Trace
trace    :: Member Trace r ⇒ String → Eff r ()
runTrace :: Eff (Trace ⊳ Void) w → IO w

-- Built-in effects (e.g., IO)
type Lift m
lift    :: (Typeable1 m, MemberU2 Lift (Lift m) r) ⇒
           m w → Eff r w
runLift :: (Monad m, Typeable1 m) ⇒
           Eff (Lift m ⊳ Void) w → m w
```

Figure 1. The interface of the library of extensible effects

As a small example, consider the computation t1 below[2]:

```
t1 :: Member (Reader Int) r ⇒ Eff r Int
t1 = do v ← ask
        return (v +1:: Int)
```

Given the type annotations on the otherwise polymorphic numeric constant, the *inferred* type of t1 indicates that t1 returns an Int after potentially manipulating an environment containing an Int. The fact that r includes, at least, the Reader Int effect is expressed by the constraint (Member (Reader Int) r). The operation ask

[1] There are more advanced ways like MemberU2 t (t m) r described below in more detail.

[2] Recall that numeric literals are polymorphic. A type annotation indicates that we are asking specifically the Reader Int effect layer, out of many possible Reader layers. Instead of annotating 1 we could have annotated the binding v. The annotations can be avoided altogether, at the expense of flexibility, see §4.

(from the Reader monad) inquires about the current value in the environment and t1 returns the incremented value.

We can execute the computation t1 by providing it with an initial value for the environment, e.g., runReader t1 (10 :: Int). The inferred type for this expression is (Eff r Int) without further constraints: the Reader Int effect has been handled and hence ‘subtracted’ from the open union. In this particular case, there are no remaining effects, and the pure computation can be run to produce a plain Int. In contrast, run t1 gives a type error.

```
t1r = run $ runReader t1 (10:: Int )
-- 11

t1rr ' = run t1
--   No instance for (Member (Reader Int) Void)
--     arising from a use of ` t1'
```

The types thus constitute an effect system that ensures that all effects must be handled.

At the expression level, the code for the computation t1 is identical to an MTL-based implementation. At the type level, the constraint (Member (Reader Int) r) looks quite like the MonadReader constraint. In fact, we may even define the MonadReader instance for the Eff r monad. However, Eff r is more general; as described above, r may contain an arbitrary number of Reader effects:[3]

```
t2 :: (Member (Reader Int) r, Member (Reader Float) r) ⇒
      Eff r Float
t2 = do
   v1 ← ask
   v2 ← ask
   return $ fromIntegral (v1 + (1:: Int )) + (v2 + (2:: Float ))
```

Each occurrence of ask obtains the value from its own environment, which it finds by type (here Int vs Float). We cannot write t2 as it is with monad transformers: with two Reader effects, monad transformers compel us to choose the order and then use the explicit lift. In contrast, the inferred signature for t2 indicates no ordering of the two Reader effects.[4] The order is determined only when we run the computation:

```
t2r = run $ runReader (runReader t2 (10:: Int )) (20:: Float )
-- 33.0
```

One may swap the two occurrences of runReader to handle the effects in the opposite order (in this particular case, without affecting the result).

In the presence of control effects such as exceptions, the order of effect handling does matter. The computation incr below increments the Int state using the MTL-like operators get and put and tes1 combine this state manipulation with exceptions:

```
incr :: Member (State Int) r ⇒ Eff r ()
incr = get ≫= put ∘ (+ (1:: Int))

tes1 :: (Member (State Int) r, Member (Exc String) r) ⇒
        Eff r a
tes1 = do incr; throwError "exc"
```

The inferred signature for tes1 shows that the two effects ‘combine’ in no particular order: to run tes1 we must handle both the State and Exc effects; we must choose which to handle first. Unlike our previous example, the choice matters (as indicated by the types):

```
ter1 :: (Either String String , Int )
ter1 = run $ runState (runError tes1) (1:: Int )
-- (Left "exc",2)

ter2 :: Either String (String , Int )
ter2 = run $ runError (runState tes1 (1:: Int ))
-- Left "exc"
```

Whereas ter1 preserves the state accumulated at the point of exception, ter2 discards it; the first semantics models the conventional “hard-wired” interactions of state and exceptions in many languages (e.g., Scheme, ML, Java, etc.) while the second semantics models a situation in which exceptions indicate failed “transactions” and hence requiring the state to snap back to its original value.

Here is another example of multiple effects: a higher-order function that applies an effectful function f to each element of a list, printing the debugging trace:

```
mapMdebug:: (Show a, Member Trace r) ⇒
             (a → Eff r b) → [a] → Eff r [b]
mapMdebug f [] = return []
mapMdebug f (h:t) = do
 trace $ "mapMdebug:␣" ++ show h
 h' ← f h
 t' ← mapMdebug f t
 return (h': t')

add :: Monad m ⇒ m Int → m Int → m Int
add = liftM2 (+)

tMd = runTrace $ runReader (mapMdebug f [1..5]) (10:: Int)
  where f x = ask `add` return x
```

The inferred type shows that the Trace effect is added to whatever effects f may produce; the example tMd uses f with an environment effect. We see the composability of effects: two independently developed pieces of code and their effect types are seamlessly used in a context that tracks both effects.

Our framework can be used with an existing monadic library (either user-developed or built-in, such as IO). For example,

```
tl1 :: (MemberU2 Lift (Lift IO) r, Member (Reader Int) r) ⇒
Eff r ()
tl1 = ask ≫= \x → lift ∘ print $ (x+1:: Int)
```

combines Reader and IO effects in *some* order, as clearly seen from the inferred type of tl1. Thus an effect of some monad m is notated as (Lift m) in our framework. Since arbitrary monads do not generally compose, there may be at most one Lift effect in a given computation, which is what the constraint MemberU2 Lift (Lift m) r on lift indicates, see Fig. 1. Therefore, we could have implemented the debugging mapMdebug in the previous example with the (Lift IO) effect instead of Trace, replacing the trace line in mapMdebug with lift (print h). The new mapMdebug can be used as before, combining the effects of the mapping function f with IO.

This briefly-described framework is also fully extensible: the interface of Fig. 1 is implemented as a *user* library that the programmer may extend at will. We will examine more advanced examples after we explain our approach in detail and contrast it with the limitations of monad transformers.

3. The Extensible Effects Framework

We now present an implementation of the the interface in Fig. 1 and explain the key design decisions. It is organized as follows: §3.1 introduces the basic concepts underpinning our approach by demonstrating a single effect, Reader Int. Then, §3.2 generalizes this approach to handle a fixed but arbitrary effect and §3.4 takes this one step further, generalizing the approach to handle many, arbitrary effects (accomplished via open unions, described in §3.3). The following sections demonstrate our framework with more examples and contrast it with monad transformers in expressiveness and efficiency.

3.1 Reader Effect as an Interaction with a Coroutine

As a soft introduction to our approach, we implement the single, simplest effect: obtaining a (dynamically-bound) Int value from

[3] The MonadReader’s version, however, requires fewer type annotations on polymorphic numerals, see §4 for discussion.

[4] The order of type class constraints in a type signature is insignificant.

the environment. We view effects as arising from communication between a client and an effect handler, or *authority*. This client-handler communication adheres to a generic client-server communication model, and thus may be easily modeled as a coroutine: a computation sends a request and suspends, waiting for a reply; a handler waits for a request, handles what it can, and resumes the client. We use the continuation monad to implement such coroutines:

```
newtype Eff a = Eff{runEff :: ∀ w. (a → VE w) → VE w}
instance Monad Eff where
  return x = Eff $ \k → k x
  m ≫= f  = Eff $ \k → runEff m (\v → runEff (f v) k)

data VE w = Val w | E (Int → VE w)

ask :: Eff Int
ask = Eff (\k → E k)

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

runReader :: Eff w → Int → w
runReader m e = loop (admin m) where
 loop :: VE w → w
 loop (Val x)  = x
 loop (E k)    = loop (k e)
```

The type Eff a is the type of computations that perform control effects instantiated to answer types (VE w) for polymorphic w (short for Value-Effect, indicating the two types that make up the signature). The answer type shows that a computation may produce a value (alternative (Val w)) or send a request to read the Int environment. This request, when resumed, continues the computation, which then recursively produces another answer of type (VE w) (or diverges). The function admin launches a coroutine with an initial continuation expecting a value, which, unless the computation diverges, must be the ultimate result.

The handler runReader launches the coroutine and checks its status. If the coroutine sends an answer, the result is returned. If the coroutine sends a request asking for the current value of the environment, that value e is given in reply. The operation ask sends a request that retrieves the current value from the environment as follows: it obtains the current continuation (the 'return address') and incorporates it into the request, constructing the Int → VE w function that will be invoked by runReader to produce the final answer.

The remaining Reader operation is local, which runs a computation in a changed environment (cf. local of the Reader monad).

```
local :: (Int → Int) → Eff w → Eff w
local f m = do
  e0 ← ask
  let e = f e0
  let loop (Val x) = return x
      loop (E k)   = loop (k e)
  loop (admin m)
```

On one hand, local must handle Reader requests, similar to runReader, and on the other, local must send Reader requests to obtain the environment value to modify. As a result, the type of local, unlike the type of runReader, does not promise to remove the Reader effect[5].

3.2 Coroutines for an Arbitrary Effect

We now extend our framework to handle other effects. For example, we may model boolean exceptions: we send the exception value as a request without specifying the return address (since no resumption is expected). The status type for such exception-throwing coroutines can be expressed as:

```
data VEex w = Val w | E Bool
```

If we instead wish to non-deterministically choose an element from a given list, we send the request that includes the list and the return address expecting one element in reply:

```
data VEch w = Val w | ∀a. E [a] (a → VEch w)
```

Examining the status type for coroutines servicing Reader, choice, and Exc requests, we observe that the status always includes the Val w alternative for normal termination and some form of E alternative carrying a request. The request typically includes the return address of the form (t → VE effect w) where t, the expected reply type, depends on the request and the result type, (VE effect w), is the status type of the coroutine. Abstracting this approach, the general type for coroutine status is revealed:

```
data VE w r = Val w | E (r (VE w r))
```

The type variable r :: ∗ → ∗ describes a particular request. For example, the Reader requests in §3.1 instantiate r with (Reader e):

```
newtype Reader e v = Reader (e → v)
```

It may be surprising that the 'type' of the effect is actually a type constructor of kind ∗ → ∗, constructing the type of the request from the status type of the coroutine. However, this follows directly from the recursive nature of the request type—an open recursive type. This type, described in the next section, will allow us to compose arbitrary effects.

Using this rich type, we easily generalize our monad coroutine library in §3.1 to arbitrary requests:

```
newtype Eff r a = Eff{runEff :: ∀ w. (a → VE w r) → VE w r}
instance Monad (Eff r)

send :: (∀ w. (a → VE w r) → r (VE w r)) → Eff r a
send f = Eff $ \k → E (f k)

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

The coroutine monad is indexed by the type of requests r that the coroutine may send. The function send dispatches these requests and waits for a reply. It obtains the suspension k of the current computation (a return address of type a → VE w r), passes k to the user-specified request builder f obtaining the request body (of the type r (VE w r)), incorporates it into the request E, and delivers it to the waiting admin. The coroutine library, along with open unions (see §3.3), provide the entire groundwork for our effect system: the rest of the code, implementing various effects (monads), may all be written by the user. We demonstrate two such effects below (and more in §5.4):

The first example monad is similar to the Identity monad: it describes pure computations that contain no effects and send no requests. We designate Void as the type of "no request." This type is not populated and thus no requests are possible:

```
data Void v −− no constructors

run :: Eff Void w → w
run m = case admin m of Val x → x
```

The function run serves as the handler for pure computations, returning their results. The type of run indicates that no effects are possible and no requests are expected; only pure computations can be run.

The effect of reading an environment from §3.1 is reimplemented with the generalized coroutine library as follows:

```
newtype Reader e v = Reader (e → v)
ask :: Eff (Reader e) e
ask = send Reader
```

[5] Here local can easily be written in terms of runReader. In the full library, §3.4, the two functions have to be implemented separately because runReader, as a full handler, forces the Reader effect layer to be the topmost so to remove it, whereas local does not.

```
runReader :: ∀ e w. Eff (Reader e) w → e → Eff Void w
runReader m e = loop (admin m) where
 loop :: VE w (Reader e) → Eff Void w
 loop (Val x)           = return x
 loop (E (Reader k)) = loop (k e)
```

The signature of runReader indicates that it takes a computation that may send (Reader e) requests and completely handles them. The result is the pure computation with nothing left unhandled.

Thus defined, Eff Reader can be used exactly like the Reader monad defined in MTL:

```
t1 :: Eff (Reader Int) Int
t1 = ask `add` return (1:: Int)
```

The inferred type of t1 betrays it as an effectful computation. The type checker prevents running it—the computation may send requests, which must be handled first:

```
t1r :: Eff Void Int
t1r = runReader t1 10
```

The inferred type indicates that t1r is pure, and so run t1r is well-typed and its evaluation produces the final result, 11.

3.3 Open Unions

With our general, single-effect system, we now turn our focus to including more effects in a single computation. Recall that, to perform an effect r, the computation sends a request of that type to the handler. The type of such computation, Eff r a, is indexed by the type r of possible requests. Thus a computation that performs requests r1 and r2 may send requests of type r1 or r2. Therefore the request itself is a disjoint union, or sum, of r1 and r2. If a programmer can add new request types at will, this sum must be extensible: an open union. Our open union should be a type-indexed co-product [15, 28]: projecting a value not reflected in the union type is guaranteed to fail and thus should be statically rejected.

The open unions we designed are abstract: the user of the extensible effects framework sees the following interface:

```
type Union r :: ∗ → ∗ −− abstract

infixr 1 ▷
data ((a :: ∗ → ∗ ) ▷ b)

class Member (t :: ∗ → ∗ ) r

inj :: (Functor t, Member t r) ⇒ t v → Union r v
prj :: (Functor t, Member t r) ⇒ Union r v → Maybe (t v)
decomp :: Union (t ▷ r) v → Either (Union r v) (t v)
```

The framework employs open unions for requests (whose types have the kind ∗ → ∗). The open union is annotated with the set r of request types that may be in this union. These sets are constructed as follows: Void stands for the empty set, and t ▷r inserts t in the set r. We also provide a type-level assertion (a type class with no members) Member t r that can be used to assert that the set r contains the request t without revealing the structure of r.

The three functions inj, prj, and decompose have the following explanation. The injection, inj, takes a request of type t and adds it to the union r. The constraint Member t r ensures that t participates in the union. (The Functor constraint is explained in §3.4.) The projection prj does the opposite. Given a value of type Union (t▷r) that may have a summand of the type t, the orthogonal decomposition decomp determines if the value has that request type t. If it does, it is returned. Otherwise, the union value is cast to a more restrictive Union r type without t—we have just determined the value is not of type t. Thus decomp projects Union r into two orthogonal "spaces:" one for the particular type t and the other for r without t. This operation sets our open unions apart from previous designs [18, 32]: we can, not only extend unions, but also shrink them. The decomposition also distinguishes our open unions from the extensible polymorphic variants of OCaml. The operation inj is used when sending a request, and prj and decomp when handling a request (as demonstrated in §3.4).

The internals of the implementation are not visible to users. There are several possible implementation approaches that would be indistinguishable by the user: we provide an implementation similar to the one for HList [15]. Here is a brief summary[6]:

```
data Union r v where
  Union :: (Functor t, Typeable1 t) ⇒ Id (t v) → Union r v
newtype Id x = Id x          −− for the sake of gcast1
instance Functor (Union r) where ...

inj :: (Functor t, Typeable1 t, Member t r) ⇒
       t v → Union r v
inj x = Union (Id x)

prj :: (Functor t, Typeable1 t, Member t r) ⇒
       Union r v → Maybe (t v)
prj (Union v) | Just (Id x) ← gcast1 v = Just x
prj _ = Nothing

decomp :: Typeable1 t ⇒
          Union (t ▷ r) v → Either (Union r v) (t v)
decomp (Union v) | Just (Id x) ← gcast1 v = Right x
decomp (Union v) = Left (Union v)

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

The implementation of the type Union r v is essentially Dynamic, and thus the constraint Typeable1 t is added to inj and prj signatures. The type parameter r, the set of requests participating in the union, is a phantom parameter. This union implementation is not directly accessible to the users: data constructor Union is not exported. The operations inj, prj and decomp are the only way for the users to deal with the open unions. Since r is phantom, the type class (Member t r) indeed does not need any members; it is a compile-time constraint with no run-time footprint. Member is also a closed class – the two instances shown here are the entirety – and the users of open unions never make instances of the class. The class Member here permits duplicates, which are harmless, allowing nesting of handlers for the same request. A request will be handled by the dynamically closest handler. With three more lines of code, see [15], duplicates can be disallowed.

Since Member is a compile-time–only constraint and gcast1 only needs to compare two Typeable.TypeRep elements, the running-time of inj and prj is constant (with respect to the size of the union). In contrast, the previously developed libraries of open unions [18, 32] have linear-time projections and injections.

3.4 The Full Library of Extensible Effects

We now present a full library of extensible effects. Its core is built upon the Eff monad and open unions; defining effects and their interactions is all done by the users. Since effects such as Reader, exceptions, non-determinism, etc. are common, we implement them ourselves by way of example (see Fig. 2) and we provide a few helpers for convenience.

The full library is similar to the example in §3.1, extended with open unions to support multiple effects. The open union is clearly indicated in the data type VE w r: the type of the sent request is Union r, describing the status of the handled computation.

We start by looking at the Reader effect. The request type Reader is identical to that of §3.1. Its sender, the operation to ask for the value of the current environment, now injects the request value into the union. This is reflected in the type of ask as compared

[6] See the file OpenUnion1.hs in the accompanying code for the complete implementation.

with the single-effect framework. The Reader handler uses the helper handle_relay to deal with arbitrary requests. In general a request goes from one handler to the next until the appropriate handler is found. The pattern of analyzing a request, finding if its type is known to the handler, and re-sending unknown requests is so common that we incorporate it into a function handle_relay. This helper function is used throughout the library. Its variant, interpose, which does not 'shrink' the request type upon relay, is used in handlers that are also senders of the same request, like local.

Sending and receiving requests, running pure code

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

admin :: Eff r w → VE w r
send  :: (∀ w. (a → VE w r) → Union r (VE w r)) → Eff r a

run :: Eff Void w → w
run m = case admin m of Val x → x
```

Helpers to relay unrecognized requests

```
handle_relay :: Typeable1 t ⇒
      Union (t ▷ r) v → (v → Eff r a) →
      (t v → Eff r a) → Eff r a
handle_relay u loop h = case decomp u of
  Right x → h x
  Left u  → send (\k → fmap k u) ≫= loop

interpose :: (Typeable1 t, Functor t, Member t r) ⇒
      Union r v → (v → Eff r a) →
      (t v → Eff r a) → Eff r a
interpose u loop h = case prj u of
  Just x → h x
  _      → send (\k → fmap k u) ≫= loop
```

Reader effect

```
newtype Reader e v = Reader (e → v)
    deriving (Typeable, Functor)

ask :: (Typeable e, Member (Reader e) r) ⇒ Eff r e
ask = send (inj ∘ Reader)

runReader :: Typeable e ⇒
    Eff (Reader e ▷ r) w → e → Eff r w
runReader m e = loop (admin m) where
 loop (Val x) = return x
 loop (E u) =
  handle_relay u loop (\(Reader k) → loop (k e))

local :: (Typeable e, Member (Reader e) r) ⇒
      (e → e) → Eff r a → Eff r a
```

Exceptions

```
newtype Exc e v = Exc e
    deriving (Functor, Typeable)

throwError :: (Typeable e, Member (Exc e) r) ⇒ e → Eff r a
throwError e = send (\_ → inj $ Exc e)

runError :: Typeable e ⇒ Eff (Exc e ▷ r) a →
Eff r (Either e a)

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

Non-determinism

```
data Choose v = ∀a. Choose [a] (a → v)
choose :: Member Choose r ⇒ [a] → Eff r a
makeChoice :: Eff (Choose ▷ r) a → Eff r [a]
```

Tracing (for debugging)

```
data Trace v = Trace String (() → v)
trace    :: Member Trace r ⇒ String → Eff r ()
runTrace :: Eff (Trace ▷ Void) w → IO w
```

Figure 2. The library of extensible effects

The desugared version of the Reader handler is as follows:

```
runReader :: Typeable e ⇒
             Eff (Reader e ▷ r) w → e → Eff r w
runReader m e = loop (admin m) where
 loop (Val x) = return x
 loop (E u)   = case decomp u of
   Right (Reader k) → loop (k e)
   Left u → send (\k → fmap k u) ≫= loop
```

As in §3.1, the return type shows that all Reader e requests are fully handled; the runReader computation may have other requests, though, represented by r. The runReader handler, as before, obtains the status of the client computation using admin and analyzes it, handling three possible cases:

1. If the client completed, its result is returned.
2. If the client sent a request, we check if it is a Reader request. If so, the client is resumed with the current value of the dynamic environment. The client may then terminate or send another request, hence we loop.
3. If the request is not a Reader request, we re-send it. That other request, u, must have contained the return address, the suspension of the type t → VE w (Reader e ▷r). When re-sending the request, runReader obtains its own suspension k, which has the type VE w (Reader e ▷r) → VE w' r. We must somehow compose the two suspensions, to obtain t → VE w' r. When runReader's handler resumes it, the runReader's client is resumed.

The problem is how to compose k with the suspension that must be somewhere in the request u, given that we have no idea what u is (not even its concrete type). Recall that all requests are described by type constructors of kind $* \to *$. The full type of the request is obtained by applying the type constructor to the status type of the required computation. In our case, the full type of the unknown other request u is Union r (VE w (Reader e ▷r)). The composition with runReader's own suspension should produce the request of the type Union r (VE w r). The problem is solved if Union r is a functor so that we can use fmap k u since k has exactly the right type VE w (Reader e ▷r) → VE w r. The type constructor Union r is a functor of each request type, this demonstrates the need for the Functor t constraint on inj and prj functions in §3.3. Since Union r is a functor, VE· r is a free monad induced by the functor. The monad Eff r then looks like a combination of the codensity monad and the free monad.

Other effects and their handlers follow the Reader pattern. The exception effect Exc is simpler to handle since the sender is not expecting to be resumed. The exception recovery catchError is quite like local; the exception handler may re-throw the exception. The non-determinism effect Choose is also familiar. We take choose to be a primitive operation (the sender of Choose effects), for non-deterministically choosing a value from a given list. The familiar mplus and mzero are trivially expressed in terms of choose. The handler makeChoice for Choose effects produces a list, of the results of all successful choices. Presently the handler is simplistic, using depth-first search, like that of the List monad. Programmer may write their own handlers with more sophisticated search strategies. A computation that sends Choose requests can be handled with a variety of handlers.

4. Simulating the Full MTL

As demonstrated in §2 and the interface described in §3.4, our framework expresses code that is commonly written with monad transformers and the MTL. We have already provided several examples of Reader, State and exception effects. This section shows how our framework expresses two common and advantageous uses of MTL: adding a Reader etc. effect to an arbitrary monad ('lifting') and 'classes' of MTL effects such as MonadReader. The accompanying code has more examples of expressing MTL idioms. We see once again that code written with monad transformers may be easily translated to our framework with minimal changes.

4.1 Type Classes for Monadic Effects

Liang et al [18] introduced type classes providing individually-tailored methods as primitive operations associated with each monadic effect. For example, (MonadReader e m) is a type class for monads m with the Reader effect, which access an environment of type e, providing primitive Reader operations ask and local. MTL is based on this style of effect type classes, and with good reason: the benefit is that the code that uses ask and local may be polymorphic over the monad m, requiring only the (MonadReader e m) constraint. Our framework provides a similar flexibility. Even further, classes like MonadReader may be implemented as concrete, stand-alone monads by hiding the Open Union (and thus flexibility).

The type class (MonadReader e m) has a functional dependency constraining m to uniquely determine the type of the environment e. A MonadReader may have only one layer of Reader effects. The advantage is that fewer type annotations are needed. For example, in the expression local (+1) (liftM (+2) ask) both ask and local refer to the same monad. In MTL, these expressions refer to the same environment (here a numeric type).

Our interface for ask and local (see Fig. 1), however, does not restrict the number of Reader effects. Therefore local (+1) and ask in the above code may refer to numerals of different types (one Int and another Float, for example). If we intend the two operations to deal with the same environment, we must add annotations (e.g., annotating both 1 and 2 in the above code to be Int). In short programs, we must generally annotate all polymorphic literals such as numerals (long programs usually have enough context to infer the correct type).

Thus MonadReader in MTL is less expressive (insisting on a single Reader effect for a given monad) but more convenient (requiring few annotations). The interface in Fig. 1 makes the opposite trade-off. Still, extensible effects can implement the less general but more convenient MonadReader interface (and similar MonadState and MonadError): simply import our Eff library and define the Eff r monad as an instance of MonadReader.

```
import qualified Eff as E
instance (MemberU Reader (Reader e) r, Typeable e) ⇒
    MonadReader e (Eff r) where
    ask   = E.ask
    local = E.local
```

The constraint MemberU Reader (Reader e) r requires that the open union of effects r must have the unique Reader effect with environment type e. The user of our framework thus has the choice of which of the two Reader interfaces to use. The file ExtMTL.hs in the accompanying code demonstrates many examples of this approach to MTL monad classes in our framework. They have exactly the same look and feel as their MTL counterparts, and fewer annotations are required than our examples in Eff.hs.

4.2 Lifting from Arbitrary Monads

A valuable and prevalent feature of monad transformers is adding effects to arbitrary monads, either developed by the user or built-in (such as IO, ST, STM). For example, with MTL, ReaderT Int IO is a monad that combines IO operations with accessing an Int environment. IO actions in such a transformed monad have to be 'lifted' (prefixed with lift or liftIO), rendering such a transformation less than transparent and requiring systematic changes across entire programs. (Furthermore, IO operations like catch for catching exceptions may be impossible to use in the transformed monad, in general.)

Extensible effects have exactly the same feature with the same limitations. We may add Reader, State, generator, and other effects to an arbitrary monad m. As with MTL, actions m a will have to be lifted (and operations that take m a actions as arguments will require ad hoc work-arounds or may be impossible to use—as was the case with monad transformers in the Lüth et al [19] approach.)

Although lifting looks identical to MTL, it means a different thing. The operation lift m sends the action m for execution to the Lift handler:

```
data Lift m v = ∀a. Lift (m a) (a → v)

lift :: (Typeable1 m, MemberU2 Lift (Lift m) r) ⇒
        m a → Eff r a
lift m = send (inj ∘ Lift m)
```

The handler receives the request, extracts and executes the action and sends its result back:

```
runLift :: (Monad m, Typeable1 m) ⇒
           Eff (Lift m ▷ Void) w → m w
runLift m = loop (admin m) where
 loop (Val x) = return x
 loop (E u)   = case prj u of
                  Just (Lift m k) → m ≫= loop ∘ k
                  −− Nothing cannot occur
```

Thus in our approach, the effects of a monad m are treated just as any other effects in our framework. There is an important difference though: since arbitrary monads generally do not compose there may be at most one Lift layer. The type of runLift makes it a "terminal handler" for computations that have Lift m and no other effects. The constraint MemberU2 Lift (Lift m) r on lift ensures the uniqueness of the Lift layer. We saw the examples of lifting from the IO monad in §2.

4.3 Efficiency

Each layer of monad transformers adds overhead. For example, a built-with-MTL monad ReaderT Int (StateT Float) Identity that combines Int environment and Float state has the type, after desugaring, Int → (Float → (Identity a, Float)). Therefore each return must build two closures and each bind has to apply them. Exchanging the order of the two layers changes the type, yet the overhead of two closures remains. Chances are, however, that only a small part of the overall computation accesses the environment or the state. Nevertheless, the entire computation has to pay the overhead of building and applying closures. Everyone pays for the needs of the few. More transform layers further increase this overhead. Our framework of extensible effects propagates requests for effects through a chain of handlers. The overhead depends on the number of other handlers between the requester and its handler, but not on the total number of handlers. The benchmarks Benchmarks.hs confirm that adding more handlers increases the overhead in the worst case, or does not affect the performance in the best case. With MTL, adding more layers always increases overhead (and the increase is larger than that in our framework).

5. Beyond the MTL

So far it may appear that our framework is just a version of MTL with more convenient lifting. Our framework goes beyond MTL, as we shall see in this section. It begins by showing where MTL falls short. Although these limitations are rarely talked about, they are very real: common programming patterns exhibiting interleaving of effects are sometimes complex and inefficient (§5.1), or even impossible to express (§5.2) when using monad transformers. Our framework deals with these situations in an efficient, straightforward way.

5.1 Inflexible Semantics of Monad Transformers

This section demonstrates a simple example of raising and handling a single exception that, surprisingly, can only be implemented with two ErrorT monad transformer layers, imposing extra run-time overhead on the entire computation.

In the absence of other effects, an exception aborts all intermediate computations up to the dynamically closest handler. The situation is however more subtle when exceptions are used in the presence of other effects. As illustrated in §2, there are two reasonable semantics for the combination of exceptions and state: either the assignments are maintained during the error handling or not. An even more subtle situation occurs in the presence of non-deterministic computations: should the exceptions be confined to each branch of the non-deterministic computation or can an exception be used to abort an entire collection of non-deterministic choices? We will study this latter situation in detail.

Concretely, we consider a computation of type m Int for some arbitrary monad m and add an exception effect as follows: if the underlying computation returns a value greater than 5, an exception is thrown. The intended semantics is that the exception aborts the rest of the m-computation. This will already prove non-trivial but possible. The next step will then be to add a handler that catches the exception, analyzes it, and either recovers by resuming the normal control flow or re-throwing the exception. Hopefully we can implement both the exception throwing and handling generically, assuming as little as possible about the monad m.

The first part of the example is implemented in MTL in the straightforward way

```
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
```

imposing no structure on the monad m other than it must support throwing TooBig exceptions. We test the guard ex2 by applying it to a non-deterministic computation of choosing an integer from the list, where:

```
choose :: MonadPlus m ⇒ [a] → m a
choose = msum ∘ map return
```

To run the test ex2 (choose [5,7,1]) we have to pick the monad m that satisfies the MonadError and MonadPlus constraints. MTL lets us build such a monad by composing monad transformers each responsible for an individual effect. In our case, by applying layers ErrorT TooBig and ListT to a base monad, Identity[7].

The layers ErrorT and ListT can be composed in two different orders, with different behaviors. One composition order gives the computation type (ErrorT TooBig (ListT Identity)) a. If we expand the type abbreviations and elide Identity, we obtain [Either TooBig a] – the type of non-deterministic computations in which each choice can either produce a value or the TooBig exception. The exception is hence confined to a non-deterministic choice. Our example calls for the exception to abandon the computation completely. We should use then the opposite order of monad layers, ListT (ErrorT TooBig Identity) a, which desugars to Either TooBig [a] – the type of computations that either produce the TooBig exceptions or the list of non-deterministic choices. This order of monad transformers arises from the composition of the run functions of Identity, ErrorT and ListT in that order:

```
ex2_1 = runIdentity ∘ runErrorT ∘ runListT $ ex2 (choose [5,7,1])
-- Left (TooBig 7)
```

The result, shown in the comments, is as desired.

The second part of our example calls for catching the exception and recovering from it in some cases:

```
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
```

This wrapper checks if the argument computation ends in a TooBig exception, but the value was not really too large. If so, we recover; otherwise, the exception is re-thrown. Adding this wrapper to the previous example ex2_1

```
ex2r_1 = runIdentity ∘ runErrorT ∘ runListT $
           exRec (ex2 (choose [5,7,1]))
-- Right [7]
```

gives a surprising and undesirable result. Our intention was to recover from the exception! The computation choose [5,7,1] makes three non-deterministic choices: the first ex2 (return 5) throws no exceptions, and so the overall result is Right [5]; in the second choice, exRec (ex2 (return 7)), an exception is thrown but it is caught, so we expect Right [7]; the last choice gives Right [1]. Collecting the choices, we expected the result to be Right [5,7,1]. That is, we expected the choice operator to be lifted 'up'. Obviously, that is not what happened: the exception is recovered from, but all other choices got lost.

The reason for our failure is that part 1 of the example needs such order of ErrorT and ListT layers so that the exception abandons non-deterministic choices. When we recover from the exception, the choices have already been irrecoverably lost. For the recovery to act as if the exception never happened, preserving the choices, the opposite order of ErrorT and ListT layers is needed. A single program needs two different orders of monad transform layers.

The example can be implemented with MTL, rather counter-intuitively, using the type:

```
ErrorT TooBig (ListT (ErrorT TooBig Identity ))
```

with two ErrorT layers. (See `transf.hs` in the accompanying code for details). The outer ErrorT TooBig corresponds to an exception confined within a non-deterministic choice, where it does not affect other choices and can be recovered from. If the exception is not recovered at the end, it is re-thrown to the inner ErrorT layer, which abandons all the choices. The following function does the re-throwing:

```
runErrorRelay :: MonadError e m ⇒ ErrorT e m a → m a
runErrorRelay m = runErrorT m ≫= check
 where check (Right x) = return x
       check (Left e)  = throwError e
```

That is not the end of our trouble: the following code

```
ex22_1 = runIdentity ∘ runErrorT ∘ runListT ∘ runErrorRelay $
           ex2 (choose [5,7,1])
-- Right [5]
```

[7] We will be using ListT for non-determinism even though it is not strictly speaking a monad transformer since ListT m is a monad only when m is a commutative monad. However, ListT is simple and is part of the MTL canon, and for the purposes of our examples, ListT is a good enough approximation of a monad transformer.

produces the answer that is very hard to account for. The surprising behavior is a consequence of the interaction of transformer layers in MTL (ErrorT is also declared to be MonadPlus), which is hard-wired into MTL and cannot be changed. To prevent the unwanted interaction of exceptions and non-determinism that is built into MTL, we have to re-write ex2 to use explicit lifting

```
ex1 :: Monad m ⇒ m Int → ErrorT TooBig m Int
ex1 m = do
  v ← lift  m
  if  v > 5 then throwError (TooBig v)
     else return v
```

and arrange to bypass the inner ErrorT TooBig layer. Finally we achieve our goal: throwing an exception that terminates the (non-deterministic) computation, and fully recovering from it.

```
ex4_1 = runIdentity ∘ runErrorT ∘ runListT ∘ runErrorRelay $
          ex1 (choose [5,7,1])
-- Left (TooBig 7)

ex4_21 = runIdentity ∘ runErrorT ∘ runListT ∘ runErrorRelay $
          exRec (ex1 (choose [5,7,1]))
-- Right [5,7,1]
```

The inevitable duplication of the ErrorT layers is not only inelegant but also inefficient. The computation to guard, choose [5,7,1], has the type ListT (ErrorT TooBig) Int or Either TooBig [Int]. It uses no exceptions yet each use of return x, which is Right [x], has to tack on the Right constructor and each bind has to pattern-match on it. The entire computation pays for something used only at the very end.

We have seen that the simple example of exception raising and recovery has a surprisingly complex and inefficient implementation because the order of monad transformer layers matters, and different parts of the same computation may require a different order of the layers. Luckily, we solved the conundrum by duplicating layers in this example, paying for that in efficiency. The next example shows that the desired effectful computation cannot be implemented at all with monad transformers.

5.2 Interleaving Effects

The next example illustrates the limitation of monad transformers forcefully: the example cannot be expressed at all using them. Two effects, dynamic bindings and coroutines, interleave and no statically fixed layering suffices. The example abstracts practical cases[8] as explored in a previous paper by a subset of the authors [16]. The gist of the example is the dynamic environment that starts off shared between the coroutine and its parent; when the coroutine changes it, the dynamic environment becomes thread-local.

Dynamic binding is a different name for the environment monad, or the Reader effect, for which MTL provides the monad transformer ReaderT. MTL does not provide the transformer for coroutines, but it is trivial to implement in terms of the continuation monad transformer ContT:

```
type CoT a m = ContT (Y m a) m
data Y m a = Done | Y a (() → m (Y m a))

yield  :: Monad m ⇒ a → CoT a m ()
runC   :: Monad m ⇒ CoT a m b → m (Y m a)
```

CoT a is the monad transformer for coroutines yield-ing the values of type a:[9] The operation runC executes a coroutine that either finishes (returns Done) or suspends on yield (returning a continuation that can be used to resume it).

Consider a scenario of a coroutine that does pretty-printing and may inquire the dynamic environment for the current paper width. The query should give the latest value set by the coroutine, or the latest value set by the parent if the coroutine has not bound that parameter. The dynamic environment therefore is only partly shared between the coroutine and its parent: it starts off shared but becomes coroutine-private when the coroutine alters it. The following characteristic example illustrates such an inherited and private access to the dynamic environment:

```
th3 :: MonadReader Int m ⇒ CoT Int m ()
th3 = ay ≫ ay ≫ local (+10) (ay ≫ ay)
 where ay = ask ≫= yield
```

The computation ay queries the dynamic environment and yields the result; th3 does this operation twice using the current dynamic environment that is inherited from the parent. It then changes environment, and queries and yields it twice more. As the type of th3 indicates, CoT layer is over the MonadReader layer.

The parent thread:

```
c31 = runReaderT (loop =≪ runC th3) 10 where
 loop (Y x k) = liftIO  (print  x) ≫ local  (+1) (k ()) ≫= loop
 loop Done    = liftIO  (print  "Done")
```

starts the coroutine in an environment with the current value 10, prints whatever the coroutine yields, and resumes the coroutine in the changed environment, with the current value 11. The printed result is disconcerting: 10 11 21 11 "Done". It starts well: the coroutine reads and yields the current environment, which is initially 10. The parent changed that value to 11 during the dynamic scope of resuming the coroutine, and the coroutine noticed that. The coroutine changed the environment to 21, and the result shows it. The last printed number is incorrect: the local changes have been lost after the suspension. We failed at maintaining the coroutine-local dynamic environment.

The failure is expected from the types: th3 executed in the monad CoT Int (ReaderT Int IO) a. Its type expands to:

```
(a → (Int → IO w)) → (Int → IO w)
```

where w is (Y Int (ReaderT Int IO), the answer-type. The suspension has the type () → (Int → IO w). Therefore, when the parent resumes it, it has the ability to pass its own dynamic environment. The type also shows that the suspension does not close over the dynamic environment, always accepting the dynamic environment from the suspension's caller. A private dynamic environment is not possible then.

With the opposite order of layers:

```
th4 :: Monad m ⇒ ReaderT Int (CoT Int m) ()
th4 = ay ≫ ay ≫ local (+10) (ay ≫ ay)
 where ay = ask ≫= lift ∘ yield
c4 = loop =≪ runC (runReaderT th4 10)
 where loop (Y x k) = liftIO  (print  x) ≫ (k ()) ≫= loop
       loop Done    = liftIO  (print  "Done")
```

the printed result 10 10 20 20 "Done" shows that the dynamic environment does become private upon the local rebinding to 20. However, the parent can no longer affect the coroutine by changing its dynamic environment: in fact, attempting to resume in a different environment, local (+1) (k ()) does not type check. Again, the types indicate what is going on. Now th4 has the monad type ReaderT Int (CoT Int IO), which is Int → (a → IO w) → IO w where w is (Y Int IO). The suspension has the type () → IO w, closing over the dynamic environment of the coroutine. The suspension no longer accepts any environment from the parent.

With monad transformers, the same effects cannot interact differently in different parts of the same computation. The static layering of monadic effects lets us implement either the shared dynamic

[8] http://lambda-the-ultimate.org/node/1396#comment-16128
http://keepworkingworkerbee.blogspot.com/2005/08/
i-learned-today-that-plt-scheme.html

[9] For simplicity, the resumption from yield, and hence yield's result type is (). It is easy to generalize to full coroutines that not only yield values but also accept values on resumption.

environment or the private one; we cannot express (practically significant) programs that need both types of interaction.

5.3 Overcoming Limitations of Monad Transformers

As a demonstration of extensible effects, we re-implement the problematic examples from above, and see their behavior is expected and no longer puzzling. Re-implementation is a strong word: switching from monad transformers to extensible effects results in little or no changes in the code. The changes are mostly confined to the type signatures.

Exceptions. The first problem in §5.1 was to guard a computation m Int: check the produced value and throw throwing an exception if the value is above some threshold. The code below is identical to the code ex2 from §5.1.

```
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
```

Only the signature differs, with the Member (Exc TooBig) r constraint instead of MonadError TooBig m. The first part of the example, testing that ex2 (choose [5,7,1]) raises the TooBig exception discarding non-deterministic choices, behaves as in §5.1;

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

ex2_1 = run ∘ runErrBig ∘ makeChoice $ ex2 (choose [5,7,1])
-- Left (TooBig 7)
```

Unlike §5.1, the same ex2 supports the second part of the example: recovering from the TooBig exception if the exceptional value was not really too big. There are no longer failures or surprises. The recovery code exRec is identical to the eponymous monad-transformer code in §5.1; only signatures differ.

```
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
```

The recovery behaves exactly as it was supposed to:

```
ex2r_1 = run ∘ runErrBig ∘ makeChoice $
           exRec (ex2 (choose [5,7,1]))
-- Right [5,7,1]
ex2r_2 = run ∘ runErrBig ∘ makeChoice $
           exRec (ex2 (choose [5,7,11,1]))
-- Left (TooBig 11)
```

If the exception is truly recovered from, see ex2r_1, the computation proceeds as if no exception happened, with no effect on non-deterministic choices—unlike what we have seen with monad transformers in §5.1.

Delimited Dynamic Scope. The second example in §5.2 was about dynamic binding in coroutines. First we need to define a new extensible effect, coroutine, in our framework. This can be done with little effort:

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

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

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

runC :: Typeable a ⇒ Eff (Yield a ▷ r) w → Eff r (Y r a)
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))
```

The Yield effect models computations that yield values of type a and that are resumed with the value of type (). Following the standard pattern, (of Reader requests) we define a new request Yield. The request carries the value to yield and the 'return address' (which is a function with argument type () since we expect only the () reply). The action yield sends the request. Its inferred signature tells that the monad Eff r has to include the Yield a effect. The interpreter for Yield requests straightforwardly produces the status of coroutine, defined to be Y r a.

It took a few lines of code to define a new effect. Since the coroutine library is essentially the same as the one we implemented with MTL earlier, the running example of querying the dynamic environment and yielding the result maintains the same form:

```
th3 :: (Member (Yield Int) r, Member (Reader Int) r) ⇒ Eff r ()
th3 = ay ≫ ay ≫ local (+(10:: Int)) (ay ≫ ay)
 where ay = ask ≫= yield'
       yield' x = yield (x:: Int)
```

The only difference from the MTL code is the type signature and a couple of annotations that fix the value of the dynamic environment to be Int. (These annotations can be dropped as described in §4.) The code that runs th3 coroutine, prints the yielded value and resumes in a modified dynamic environment is also the same as before modulo a few annotations:

```
c31 = runTrace $ runReader (loop =≪ runC th3) (10:: Int)
 where loop (Y x k) = trace (show (x:: Int)) ≫
                        local (+(1:: Int)) (k ()) ≫= loop
       loop Done    = trace "Done"
```

The result is 10 11 21 21 Done—which is in stark contrast with anything we could achieve with MTL. The result shows the coroutine shares the dynamic environment with its parent; however, when the environment is locally rebound, it becomes private to the coroutine and therefore no longer affected by the parent (that's why the last two numbers in the trace are the same). The library of extensible effects has managed to express the important programming pattern—a task at which the MTL failed.

We have thus seen that our framework of extensible effects subsumes MTL. We can express any MTL computations with our framework (in essentially the same syntax), and we can also write code that is unwieldy or impossible with monad transformers.

5.4 Non-Determinism with Control

We close this section with a final example that shows that it is much more straightforward to *reason* about effects in our library than the corresponding effects in the MTL. In a classic paper, Hinze [9] shows how to use simple reasoning principles to derive monad transformers. When Hinze considered backtracking with the Prolog-like 'cut', the derivations of the two monad transformer implementations fail to follow the suggested reasoning principles. In particular, the term-based monad transformer is not based on free algebra; the context-passing implementation has to pattern-match on the context and hence cannot be in a continuation-passing style. Mainly, the derivation is far from being simple or mechanical, relying on "mind-boggling" types and requiring properties that have not been proven (the paper hints on proofs by induction; however the induction does not apply to recursive Haskell terms, which are not well-founded). It is instructive to compare Hinze's derivation with the remarkably straightforward implementation of the same example below.

The goal is to extend the monad of backtracking (with operations mzero and mplus) with 'cut' and 'call' whose specification [9, Sec. 5] we repeat below for reference. Hinze recommends implementing cut in terms of a primitive cutfalse :: m a, which is like mzero in that it discards further choices. The primitive expresses a common Prolog idiom and has a clearer semantics with the following equational laws:

```
cutfalse  ≫= k          ≡ cutfalse
cutfalse  `mplus` m  ≡ cutfalse
```

That is, cutfalse is the *left* zero of both (≫=) and mplus. The operation call :: m a → m a delimits the effect of cutfalse: call m executes m; if cutfalse is encountered, only choices made since the execution of m are discarded. Hinze postulates the axioms of call:

```
call  mzero                    ≡ mzero
call  (return  a `mplus` m) ≡ return  a `mplus` call  m
call  (m `mplus` cutfalse )  ≡ call  m
call  (lift   m ≫= k)         ≡ lift   m ≫= (call ∘ k)
```

The fundamental problem that so complicates the derivation of monad transformers is the absence of the axiom specifying the interaction of call with (≫=) and the way to simplify nested invocations of call.

The framework of extensible effects does not encounter such difficulties. First we notice that the property cutfalse being the left zero of (≫=) tells that cutfalse is an exception:

```
data CutFalse = CutFalse deriving  Typeable
cutfalse  = throwError CutFalse
```

Since call delimits the effect of cutfalse, it is supposed to completely handle the CutFalse exception. As for the Choose effect, call should intercept these requests (to accumulate the choice points it may have to discard) and re-issue them later. The signature of call summarizes these properties. The complete code follows:

```
call  ::  Member Choose r ⇒ Eff (Exc CutFalse ▷ r) a → Eff  r  a
call  m = loop [] (admin m) where
 loop jq  (Val x)  = return x `mplus` next jq
 loop jq  (E u) = case decomp u of
    Right (Exc CutFalse) → mzero
    Left  u → check jq  u

 check jq  u |  Just (Choose []  _) ← prj  u   = next jq
 check jq  u |  Just (Choose [x]  k) ← prj  u = loop jq  (k x)
 check jq  u |  Just (Choose lst  k) ← prj  u =
    next $ map k lst  ++ jq
 check jq  u = send (\k → fmap k u) ≫= loop jq

 next []      = mzero
 next (h:t) = loop t  h
```

Each clause corresponds to an axiom of call or cutfalse, covering all axioms. The code clearly expresses the intuition that call watches the choice points of its argument computation. When it encounters a cutfalse request, it discards the remaining choice-points. The accompanying `Eff.hs` has several examples of using cutfalse and call, including nested occurrences of call, which present no problems.

We have demonstrated how to define, and reason about, the interaction of two existing effects, Choose and Exc. We simply defined a new, joint handler, which can be used with the previously written code with Choose effects and alongside other handlers of Choose (e.g., makeChoice).

6. Related Work

Effect systems have become an active research area, with several approaches being pursued concurrently and sometimes independently. The main approaches improve monad transformers [27], encode handlers for algebraic effects [1, 25], or start with a type-and-effect system [8, 17, 32]. Our system, independently developed from EDLS years ago, ended up with many similarities to the above approaches, and also with several notable differences. First, our framework is not a stand-alone language; rather, it is an *ordinary Haskell library*. Haskell with common extensions turns out capable of expressing the type-and-effect system similar to those mentioned above.

Monad transformers and abstractions. One particular problem with monads composed of many transformers is the difficulty of accessing a particular transformer layer: implicit lifting may be impossible (e.g., when there are several State layers) and explicit lifting is too painful. Schrijvers and Oliveira [27] cleverly address the lifting problem with a new, convenient layer access mechanism while Swamy et al., [31] provide a system for ML that infers where and how to lift operations. We sidestep the entire lifting problem through type-indexed effects. Our approach also solves the problem of indicating which of several State or Reader, etc., effects to perform. Since State effects are indexed by the type of the state, the state types naturally and automatically distinguish the effects, without any extra mechanisms. Multiple State layers of the same type can be handled via a standard newtype trick:

```
newtype SInt = D Int deriving  (Typeable,  Num, Eq)

incr  ::  (Typeable a,  Num a, Member (State a) r) ⇒ Eff  r  a
incr  = do x ← get;  put (x+1); return  (x+1)

doubleIncr  = do x ← incr ;  y ← incr ;  return  (x:: Int , y:: SInt)

run $ runState (runState doubleIncr  (0:: Int )) (5:: SInt)
−− (((1,6),1),6)
```

The two explicit type annotations Int and SInt are sufficient to distinguish the usage of incr in both calls of doubleIncr; we can use the newtype trick to navigate our "transformer stack" implicitly.

Handlers for algebraic effects. Our approach is closely related to the recent congruence of ideas about *effect handlers* [1, 2, 13, 21, 25, 33]. Each of these effect handler implementations incorporates an effect typing mechanism and decides on the subtle trade-offs of effect processing.

Effect handlers are the generalization of exception handlers to other effects – the insight first realized by Plotkin and Pretnar [25]. Our handle_relay in Figure 2 is indeed quite like catch from the current Control.Exception library [20]: handle_relay only deals with requests/exceptions of a certain type, automatically re-throwing all other requests for an 'upstream' handler to process. Using existentials and Typeable to implement open union is also the same. However, our requests ('exceptions') are resumable rather than abortive and our open unions are typed.

Like Frank [21], our library implements *shallow handlers*: each run*M* selects by case analysis only those effects it wishes to handle. Much other work [1, 2, 13, 33] focuses on *deep handlers*, in which all effects of a particular kind are handled uniformly. Compared to shallow ones, deep handlers are often more efficient but less flexible.

Our library manages sets of effects using both type-level constraints and type-level lists; Kammar et al. [13] 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. Kammar et al. rely on Template Haskell to avoid much of the inconvenience of type-class encoding and provide a pliable user interface. Our library is syntactically lighter-weight and requires no special syntax.

Another trade-off is allowing multiple instances of the same effect. We permit not only both State Int and State Float in the same computation but also several instances of State Int (although the latter can be easily disabled). A request for state value or change will be handled by the closest handler of the appropriate state type. Eff [1] uses a region-based approach where effect instances are tied to region indices while Brady's library for Idris [2] allows users to name and nominally refer to individual instances.

Type-level denotation of effects. Swierstra, Filinski, and others (e.g. [8]) provide type-level systems for layering effects. However, such systems lack the ability to *subtract* effects (which occurs when these effects have been handled in our system). Our continuation encoding and union type are similar to Filinski's, but our union type is shrunk as effects are removed.

7. Conclusions

We have presented a new framework for extensible effects that subsumes previous Haskell approaches in expressiveness. The main highlights of our system are:

- easy combination of computations performing arbitrary effects; the type system collects the effects from each computation into a larger union.
- effects arise from an interaction between a client and a handler; a handler can choose to service one effect or several possibly interleaving effects;
- the effects that have been completely handled are subtracted from the type leaving a computation with fewer effects that can be further composed with other effects or handled.

We have demonstrated that our framework of extensible effects subsumes the MTL: any effectful MTL computation can be rewritten in our framework, in essentially the same syntax; we can also write code that is unwieldy or impossible using monad transformers. Furthermore, we have gained *flexible efficiency:* our dynamic order of layers is contained in one monad, parts of the computation that don't use a particular effect don't pay for it (meaning each effect is "pay-per-use").

There are several natural avenues for future work:

- Partitioning "large" monads like the IO monad into compartmentalized sections that may each be added and removed from computations. These partitions include IORefs, IO exceptions, time functions, pure reading, and reading *and* writing.
- Implementing ST-like effects, explicitly marking state, and providing an allocation system using monadic regions to the user.
- Providing a delimited control interface with shift, prompt and control, or a similar collection of operators.
- Cleaning up the implementation by including the Eff (continuation) monad natively in Haskell (like IO and ST). We would then come full-circle to the Scheme and ML families of languages, providing *efficient* native state, IO, and continuations and mechanisms for users derive other effects from them. The advantage here is that we have a type system to track these user-defined effects.

Acknowledgments

We are very grateful to Simon Peyton-Jones for many helpful comments and discussions. Insightful comments and suggestions by Sam Lindley and anonymous reviewers are greatly appreciated. This material is based upon work supported by the National Science Foundation under Grant No. 1117635.

References

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

[2] E. C. Brady. Programming and reasoning with algebraic effects and dependent types. In ICFP [11], page To Appear.

[3] 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.

[4] D. Espinosa. Modular denotational semantics. Unpublished manuscript, 1993.

[5] D. Espinosa. Building interpreters by transforming stratified monads. Unpublished manuscript, 1994.

[6] D. A. Espinosa. *Semantic Lego*. PhD thesis, Columbia University, New York, NY, USA, 1995. UMI Order No. GAX95-33546.

[7] A. Filinski. Representing monads. In POPL [26], pages 446–457.

[8] A. Filinski. Representing layered monads. In *POPL '99*, pages 175–188, New York, NY, USA, 1999. ACM.

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

[10] 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.

[11] ICFP. *ICFP '13*, 2013. ACM Press.

[12] M. Jaskelioff and E. Moggi. Monad transformers as monoid transformers. *Theor. Comp. Science*, 411(51/52):4441 – 4466, 2010.

[13] O. Kammar, S. Lindley, and N. Oury. Handlers in action. In ICFP [11], page To Appear.

[14] D. King and P. Wadler. Combining monads. In J. Launchbury and P. Sansom, editors, *Functional Programming, Glasgow 1992*, Workshops in Computing, pages 134–143. Springer London, 1993.

[15] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Proc. 2004 workshop on Haskell*, pages 96–107, New York, NY, USA, 2004. ACM.

[16] O. Kiselyov, C.-c. Shan, and A. Sabry. Delimited dynamic binding. In *ICFP '06*, pages 26–37. ACM Press, 2006.

[17] D. Leijen. Koka: A language with row-polymorphic effect inference. In *1st Workshop on Higher-Order Programming with Effects (HOPE 2012)*. ACM, September 2012.

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

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

[20] S. Marlow. An extensible dynamically-typed hierarchy of exceptions. In *Proc. 2006 Haskell Workshop*, pages 96–106. ACM Press, 2006.

[21] C. McBride. The Frank manual. https://personal.cis.strath.ac.uk/conor.mcbride/pub/Frank/, 2012.

[22] E. Moggi. Computational lambda-calculus and monads. In *LICS*, pages 14–23, Piscataway, NJ, USA, 1989. IEEE Press.

[23] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Edinburgh Univ., 1989.

[24] B. O'Sullivan, J. Goerzen, and D. Stewart. *Real world Haskell – code you can believe in*. O'Reilly, 2008.

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

[26] POPL. *POPL '94*, 1994. ACM Press.

[27] T. Schrijvers and B. C. Oliveira. Monads, zippers and views: virtualizing the monad stack. In *ICFP '11*, pages 32–44, New York, NY, USA, 2011. ACM.

[28] M. Shields and E. Meijer. Type-indexed rows. In *POPL '01*, pages 261–275, London, United Kingdom, Jan. 17–19, 2001. ACM Press.

[29] G. L. Steele, Jr. How to compose monads. Technical report, Thinking Machines Corporation, Cambridge, Massachusetts, July 1993.

[30] G. L. Steele, Jr. Building interpreters by composing monads. In POPL [26], pages 472–492.

[31] N. Swamy, N. Guts, D. Leijen, and M. Hicks. Lightweight monadic programming in ML. In *ICFP'11*, pages 15–27, Sept. 2011.

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

[33] S. Visscher. Control.effects, 2012. URL http://github.com/sjoerdvisscher/effects.

[34] P. Wadler. The essence of functional programming. In *POPL '92*, pages 1–14, New York, NY, USA, 1992. ACM.