Extensible Effects

(An Alternative to Monad Transformers)

Introducing Extensible Effects

Extensible Effects provides an alternative to monad transformer stacks in the form of a single monad `Eff`, for the coroutine-like communication of a client with its effect handler. The ambition is for Eff to be the only monad in Haskell; rather than defining new monads, programmers will be defining new effect interpreters.

- We denote handlers as authorities that control some resources and interpret some requests
- We develop an expressive type-and-effect system that keeps track of which effects are currently active in a computation, by maintaining an open union containing an unordered collection of current effects.
- The action of each effect handler is reflected by removing the effects that have been handled.

Implementing the Extensible Effects Framework

1. Reader Effect as a Coroutine Interaction

We implement the effect of reading from a dynamically bound environment.

Recall:

- We view effects as arising from communication between a client and an effect handler. Hence this may be easily modeled as a coroutine.
  1. A computation sends a request and suspends, waiting for a reply.
  2. A handler waits for a request, handles what it can, and resumes the client.

Hence:

- We use the continuation monad to implement such coroutines.

1. Reader Effect as a Coroutine Interaction

For now, we implement Eff specifically for just the reader effect.

Let the type `VE w` (where `VE` is short for Value-Effect) represent the answer/status of a coroutine.

```
data VE w = Val w | E (Int -> VE w)
```

The coroutine status shows that a computation can either:

1. `Val w`
   Produce the final value of type `w`
2. `E (Int -> VE w)`
   Send a resumable request which reads from the environment. This request, when resumed, continues the computation which then recursively produces another answer of type `VE w`.

We implement the `Eff` monad to represent the type of computations that perform control effects instantiated to *coroutine status types* `VE`.

```
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)
```

Note how `Eff` is exactly like the continuation monad, except the result type `r` is specialised to `VE w`

```
newtype Cont r = Cont { runCont :: ∀ r. (a -> r) -> r }
```

1. Reader Effect as a Coroutine Interaction

Given the following:

```
data VE w = Val w | E (Int -> VE w)
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)
```

We can now implement reader operations.

```
ask :: Eff Int
ask = Eff (\k -> E k)
```

The function `ask` sends a reader request. This request will obtain the current continuation `k` (to be performed after retrieving the environment) and incorporate it into the request constructor `E`. This constructs a request containing a function `k` which will be invoked by `runReader`.

```
admin :: Eff w -> VE w
admin (Eff m) = m Val
```

The function `admin` launches a coroutine with an initial continuation `Val` that expects the value, which must be the final result.

```
runReader :: Eff w -> Int -> w
runReader m env = loop (admin m) where
 loop :: VE w -> w
 loop (Val x) = x
 loop (E k)   = loop (k env)
```

The effect handler `runReader` launches the coroutine and checks its status.

1. If the coroutine sends an answer `(Val x)`, then the result is returned
2. If the coroutine sends a request `(E k)` asking for the current value of the environment, then the value `env` is given in reply.

2. Arbitrary Effects as a Coroutine Interaction

We now extend the framework to handle other effects. Let's first look at some concrete examples of other effects to get a gist of the underlying pattern.

Examples of other effects as coroutines:

We can model boolean exceptions. To do this, we reimplement the coroutine status type VE by redefining what the request E should be. We should send a request with the exception value Bool without specifying the continuation, since no resumption is expected. The status type for exception-throwing coroutines can be expressed as:

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

We can model non-deterministic effects, for example, one which non-deterministically chooses an element from a given list. To do this, we send the request that includes the list `[a]` and the continuation `a -> VEchoose w` expecting one element in the reply.

```
data VEchoose w = Val w | forall a. E [a] (a -> VEchoose w)
```

2. Arbitrary Effects as a Coroutine Interaction

Implementing the Groundwork for Generalized Coroutines

By looking at the answer/status types for the coroutines servicing reader, choice, and exception requests:

```
data VEreader w    = Val w | E (Int -> VEreader w)
data VEexception w = Val w | E Bool
data VEchoose w    = Val w | forall a. E [a] (a -> VEchoose w)
```

We make the observation that the coroutine status type `VE` for an effect `Eff` always includes:

1. The `Val w` alternative for normal termination with a final value.
2. An `E` alternative for carrying requests.

   The request typically includes the continuation of form `t -> VE effect w` where
   - `t` is the expected reply type which depends on the request
   - `VE effect w` is the status type of the coroutine

If we abstract this approach, the general type for the coroutine status is

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

The type parameter `r :: * -> *` describes a particular request

The effect `r` is a type constructor of kind `* -> *`, constructing the *type of the request* `E` from the status type of the coroutine `VE w r`. This follows directly from the recursive nature of the request type. This type will allow us to compose a single type of arbitrary effects.

For example, the Reader request would instantiate r with Reader e:

```
newtype Reader e v = Reader (e -> v)
```

Hence using the Reader request would look like `E (Reader (e -> VE w (Reader e)))`

2. Arbitrary Effects as a Coroutine Interaction

Implementing the Groundwork for Generalized Coroutines

Using this rich type for coroutine status types

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

We can generalize our coroutine monad `Eff` to arbitrary requests. It is now also indexed by the type of requests `r` that the coroutine may send (for now, without open unions, we can only index it by a single request).

```
newtype Eff r a = Eff { runEff : forall w. (a -> VE w r) -> VE w r }
```

The function `send` dispatches a request `r` and waits for a reply.

```
send :: (forall w. (a -> VE w r) -> r (VE w r)) -> Eff r a
send f = Eff $ \k -> E (f k)
```

- `f :: (forall w. (a -> VE w r) -> r (VE w r))` is a user-specified request builder.
  This is typically a type constructor representing a type of effect (e.g. `Reader`), which takes a suspended continuation.
- It obtains the suspension `k :: a -> VE w r`
- It applies `f` to `k`, to obtain the request body `f k :: r (VE w r)`
- It incorporates the request body `fk` into the request `E`

The function `admin` takes a coroutine and launches it with the initial continuation being `Val` which expects a final return value.

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

2. Arbitrary Effects as a Coroutine Interaction

The previously described coroutine library (along with open unions) provides the entire groundwork for our effect system. We demonstrate two such effects below:

Pure Computations

The type `Void` is the type of no requests, similar to using the `Identity` monad. It describes pure computations that contain no effects and send no requests.

```
data Void v -- no constructors
```

The function `run` serves as the handler for pure computations.

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

Reader Effect

The effect of reading from an environment is reimplemented with the generalized coroutine implementation of `Eff`

```
newtype Reader e v = Reader (e -> v)

ask :: Eff (Reader e) e
ask = send Reader

runReader :: forall 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 k)` requests, and completely handles them. The result is the pure computation with nothing left unhandled.

3. Open Unions

With our general, single-effect system, recall:

- To perform an effect `r`, the computation sends a request of that type to the effect handler. For example:

  ```
  send :: (forall w. (a -> VE w r) -> r (VE w r)) -> Eff r a
  send f = Eff $ \k -> E (f k)

  ask :: Eff (Reader e) e
  ask = send Reader
  ```

- The type of such a computation, `Eff r a`, is indexed by the type `r` of possible requests.

We now look at how to include more effects in a single computation. A computation that performs requests `r1` and `r2` may send requests of type `r1` or `r2`. Therefore, the request itself is a disjoint union, `r1 ∪ r2`. In order to add new request types at will, this must be an open union, which should be a type-indexed co-product. Projecting a value not reflected in the union type is guaranteed to fail, and thus should be statically rejected.

3. Open Unions

The open unions designed are abstract, where the users of the framework see the following interface.

```
type Union r :: * -> * --abstract
infixr 1 ▷
data (( a :: * -> *) ▷ b)
class Member (t :: * -> *) r
```

Open unions `Union` are 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
- `t ▷ r` inserts request `t` into the set `r`

We also provide a type-level assertion `Member t r` (a type class with no members) that assert that the set `r` contains the request `t`, without revealing the structure of `r`.

```
inj :: (Functor t, Member t r) ⇒ t v -> Union r v
```

- The injection `inj` takes a request of type `t` and adds it to the union, producing `r`. The constraint `Member t r` ensures that `t` participates in the union `r`.

```
prj :: (Functor t, Member t r) ⇒ Union r v -> Maybe (t v)
```

- The projection `prj` takes a union of type `Union r v` and projects out the request `t`, where the constraint `Member t r` ensures that `t` participates in the union `r`.

```
decomp :: Union (t ▷ r) v -> Either (Union r v) (t v)
```

- The decomposition `decomp`, given a value of type `Union (t ▷ r) v` that may have a member of type `t`, determines if the value has that request type `t`. If it does, then 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.

4. Full Library of Extensible Effects

Coroutine Status

Previously for a single effect, a coroutine status was defined as:

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

Now, using the open union of different possible effects `r`, the definition for a coroutine status is given by:

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

Which means that the type of request can be anything which is a member of the union `r`.

Sending Requests

Previously for a single effect, sending a request was defined as:

```
send :: (forall w. (a -> VE w r) -> r (VE w r)) -> Eff r a
send f = Eff $ \k -> E (f k)
```

Now using the open union, the definition for send is given by:

```
send :: (forall w. (a -> VE w r) -> Union r (VE w r)) -> Eff r a
send f = Eff $ \k -> E (f k)
```

Launching Coroutines

Previously for a single effect, launching a coroutine was defined as:

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

Now using the open union, the definition of admin stays the same.

4. Full Library of Extensible Effects

To run pure code, we have the function `run` which operates on the empty set of effects.

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

To handle arbitrary requests we have `handle_relay`. The pattern of this function is that given a request (open union), we either handle it with `h` or relay it with `loop`.

```
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
      -- = Eff (\k -> E (fmap k u)) >>= loop
```

- `u` is the union of requests `Union (t ▹ r) v` that may contain the request type `t`
- We try to decompose `u :: Union (t ▹ r) v`
  - If `u` contains `t`, then we are given the request `t v` and handle this request with the handler `h :: t v -> Eff r a`
  - If `u` does not contain `t`, then we relay the union of requests `Union r v` by running
    `send (\k -> fmap k u) >>= loop`
    This creates a coroutine `Eff` containing a function that when given a suspension/request k, it creates a request (`fmap k u`) consisting of mapping this suspension `k` over the union of requests `u`.
    It then processes the resulting requests with the `loop` handler.

4. Full Library of Extensible Effects

To interpose