Structuring effectful programs

BOB 2023

Andres Löh

2023-03-17

Well-Typed
The Haskell Consultants

This is a talk about abstraction

What are effects?

Aspects of your code . . .

- . . . that are awkward to express in a purely functional style.
- . . . you might want to abstract from.
- . . . you might want to interpret differently.
- . . . that are inherently side effects.

Well-Typed

More concretely / examples

- state (i.e., mutable variables)
- error handling, exceptions
- non-determinism
- logging
- concurrency
- randomness
- disk access
- database access
- networking
- ...

Effectful applications

Most applications need various effects.

We would like to control them:

- safety,
- flexibility,
- design.

Monads and applicative functors

In Haskell, we generally use **monads** (and related interfaces such as (applicative) functors to model effects).

A computation of type `M a` represents a computation yielding values of type `a`, but potentially encapsulating various effects.

Monads and applicative functors

In Haskell, we generally use **monads** (and related interfaces such as (applicative) functors to model effects).

A computation of type `M a` represents a computation yielding values of type `a`, but potentially encapsulating various effects.

```
pure :: a -> M a
```

embeds a “pure” value of type `a` into `M a` without making use of any effects.

Monads and applicative functors

In Haskell, we generally use **monads** (and related interfaces such as (applicative) functors to model effects).

A computation of type `M a` represents a computation yielding values of type `a`, but potentially encapsulating various effects.

```
pure :: a -> M a
```

embeds a "pure" value of type `a` into `M a` without making use of any effects.

```
(>>=) :: M a -> (a -> M b) -> M b
```

sequences two effectful operations, where the second can depend on the results of the first.

Monad laws

To be intuitive, we want to capture some of the informal notions we have about `pure` and `(>>=)` in **laws**:

Monad laws

To be intuitive, we want to capture some of the informal notions we have about `pure` and `(>>=)` in **laws**:

```
pure x >>= k = k x
m >>= pure   = m
```

(`pure` should really not use any effects.)

Well-Typed

Monad laws

To be intuitive, we want to capture some of the informal notions we have about `pure` and `(>>=)` in **laws**:

```
pure x >>= k = k x
m >>= pure   = m
```

(`pure` should really not use any effects.)

```
(m >>= k) >>= l = m >>= (\ x -> k x >>= l)
```

(`(>>=)` should really sequence.)

How to find the right monadic type?

Monolithic approach

We can design a specific type `M` that does everything we need.

Monolithic approach

We can design a specific type `M` that does everything we need.

But then:

- We have to define the instances by hand.
- Easy to make mistakes / extra work to be convinced that the laws are satisfied.
- Encourages us not to make changes, or to be very imprecise.
- Inflexible if we want different implementations for different settings (in particular testing / staging).

Combining effects

We can construct a type `M` in some way out of a library of effects.

Two popular approaches for this are:

- monad transformers,
- (algebraic / extensible) effects.

Examples

Effectful code using mtl / transformers

```
validate payload = do
  let pid = payloadId payload
  liftIO (putStrLn ("Validating " <> show pid))
  case checkPayload payload of
    Ok -> do
      ctr <- get
      put (ctr + payloadSize payload)
    NotOk -> do
      liftIO (putStrLn ("Ignoring payload " <> show pid))
    FatalError ->
      throwError (FatalValidationError ...)
```

---

mtl and transformers packages are originally by Andy Gill and Ross Paterson

Effectful code using mtl / transformers

```
validate ::
  Payload -> StateT Int (ExceptT ValidationError IO) ()
validate payload = do
  let pid = payloadId payload
  liftIO (putStrLn ("Validating " <> show pid))
  case checkPayload payload of
    Ok -> do
      ctr <- get
      put (ctr + payloadSize payload)
    NotOk -> do
      liftIO (putStrLn ("Ignoring payload " <> show pid))
    FatalError ->
      throwError (FatalValidationError ...)
```

`mtl` and `transformers` packages are originally by Andy Gill and Ross Paterson

Effect interfaces

Transformer-based effects come with interfaces . . .

Effect interfaces

Transformer-based effects come with interfaces ...

```haskell
class MonadState s m where
  get :: m s
  put :: s -> m ()
```

Effect interfaces

Transformer-based effects come with interfaces . . .

```
class MonadState s m where
  get :: m s
  put :: s -> m ()
```

```
class MonadError e m where
  throwError :: e -> m a
  catchError :: m a -> (e -> m a) -> m a
```

Effect interfaces

Transformer-based effects come with interfaces . . .

```
class MonadState s m where
  get :: m s
  put :: s -> m ()
```

```
class MonadError e m where
  throwError :: e -> m a
  catchError :: m a -> (e -> m a) -> m a
```

```
class MonadIO m where
  liftIO :: IO a -> m a
```

Effect interfaces

Transformer-based effects come with interfaces …

```
class MonadState s m where
  get :: m s
  put :: s -> m ()
```

```
class MonadError e m where
  throwError :: e -> m a
  catchError :: m a -> (e -> m a) -> m a
```

```
class MonadIO m where
  liftIO :: IO a -> m a
```

… and suitable instances such that in

```
StateT Int (ExceptT ValidationError IO) ...
```

we can use all these methods.

Effectful code using `mtl` / `transformers`

```
validate ::
  ( MonadState Int m
  , MonadError ValidationError m
  , MonadIO m
  ) => Payload -> m ()
```

```
validate payload = do
  ...  -- exactly as before
```

Well-Typed

Effectful code using `effectful`

```
validate ::
  ( State Int :> es
  , Error ValidationError :> es
  , IOE :> es
  ) => Payload -> Eff es ()
validate payload = do
  ...  -- exactly as before
```

`effectful` package is by Andrzej Rybczak

All of these versions are bad

. . . because of course we should **abstract**!

Let us revisit the code

```
validate payload = do
  let pid = payloadId payload
  liftIO (putStrLn ("Validating " <> show pid))
  case checkPayload payload of
    Ok -> do
      ctr <- get
      put (ctr + payloadSize payload)
    NotOk -> do
      liftIO (putStrLn ("Ignoring payload " <> show pid))
    FatalError ->
      throwError (FatalValidationError ...)
```

Our use of state

```haskell
validate payload = do
  ...
  case checkPayload payload of
    Ok -> do
      ctr <- get
      put (ctr + payloadSize payload)
  ...
```

Our use of state

```haskell
validate payload = do
  ...
  case checkPayload payload of
    Ok -> stepCounterBy (payloadSize payload)
  ...
```

```haskell
stepCounterBy :: State Int :> es => Int -> Eff es ()
stepCounterBy i = do
  ctr <- get
  put (ctr + i)
```

Our use of state

```haskell
validate payload = do
  ...
  case checkPayload payload of
    Ok -> countPayload payload
  ...
```

```haskell
stepCounterBy :: State Int :> es => Int -> Eff es ()
stepCounterBy i = do
  ctr <- get
  put (ctr + i)
```

```haskell
countPayload :: State Int :> es => Payload -> Eff es ()
countPayload payload =
  stepCounterBy (payloadSize payload)
```

Our use of IO and exceptions

Similarly:

```
logMsg :: IOE :> es => String -> Eff es ()
logMsg msg = liftIO (putStrLn msg)

stop ::
  Error ValidationError :> es
   => ValidationError -> Eff es a
stop err = throwError err
```

An improved version

```
validate payload = do
  let pid = payloadId payload
  logMsg ("Validating " <> show pid)
  case checkPayload payload of
    Ok -> countPayload payload
    NotOk -> logMsg ("Ignoring payload " <> show pid)
    FatalError -> stop (FatalValidationError ...)
```

This version is still bad

We have abstracted the terms, but not the types

The types are as before …

Transformers:

```
validate ::
  ( MonadState Int m
  , MonadError ValidationError m
  , MonadIO m
  ) => Payload -> m ()
```

Effects:

```
validate ::
  ( State Int :> es
  , Error ValidationError :> es
  , IOE :> es
  ) => Payload -> Eff es ()
```

Well-Typed

Low-level versus high-level effects

Libraries (by necessity) offer mostly **low-level** effects:

- They are the fundamental building blocks.
- They are most widely applicable.

Low-level versus high-level effects

Libraries (by necessity) offer mostly **low-level** effects:

- They are the fundamental building blocks.
- They are most widely applicable.

. . . **but they are also least informative!**

High-level effects

- If we need a counter, we should reflect that in the types (and not use `State`).
- If we need a logger, we should reflect that in the types (and not use `IO`).
- …

High-level effects

- If we need a counter, we should reflect that in the types (and not use `State`).
- If we need a logger, we should reflect that in the types (and not use `IO`).
- …

```
class MonadCounter m where
  stepCounterBy :: Int -> m ()
class MonadLogger m where
  logMsg :: String -> m ()
```

High-level effects

- If we need a counter, we should reflect that in the types (and not use `State`).
- If we need a logger, we should reflect that in the types (and not use `IO`).
- ...

```
class MonadCounter m where
  stepCounterBy :: Int -> m ()
class MonadLogger m where
  logMsg :: String -> m ()
```

(These classes are also usable for an effects library, and/or you can define new effects ...)

High-level effects

- If we need a counter, we should reflect that in the types (and not use `State`).
- If we need a logger, we should reflect that in the types (and not use `IO`).
- ...

```
data Counter :: Effect where
  StepCounterBy :: Int -> Counter m ()
type instance DispatchOf Counter = Dynamic
stepCounterBy :: Counter :> es => Int -> Eff es ()
stepCounterBy = send . StepCounterBy
```

More meaningful effects

```
validate ::
  ( MonadCounter m
  , MonadError ValidationError m
  , MonadLogger m
  ) => Payload -> m ()
```

The use of low-level effects (such as `State` or `IO`) should generally be an implementation detail.

More meaningful effects

```
validate ::
  MonadValidate m => ... -> m ...
```

The use of low-level effects (such as `State` or `IO`) should generally be an implementation detail.

We can go yet further to an **application-specific effect** …

Another design question

What is the better type for `countPayload` ?

```
countPayload :: MonadCounter  m => Payload -> m ()
countPayload :: MonadValidate m => Payload -> m ()
```

Another design question

What is the better type for `countPayload` ?

```
countPayload :: MonadCounter  m => Payload -> m ()
countPayload :: MonadValidate m => Payload -> m ()
```

The former is more precise.

Another design question

What is the better type for `countPayload` ?

```
countPayload :: MonadCounter  m => Payload -> m ()
countPayload :: MonadValidate m => Payload -> m ()
```

The former is **too** precise.

The pitfalls of excessive bottom-up design

```
f1 :: MonadX1 m => m ...
f2 :: MonadX2 m => m ...
f3 :: MonadX3 m => m ...
...
```

The pitfalls of excessive bottom-up design

```haskell
f1 :: MonadX1 m => m ...
f2 :: MonadX2 m => m ...
f3 :: MonadX3 m => m ...
...
```

```haskell
composition ::
  ( MonadX1 m
  , MonadX2 m
  , ...
  , MonadX1000 m
  ) => m ...
```

The pitfalls of excessive bottom-up design

Being precise about effects, and propagating them bottom-up leads to:

- a temptation to grant functions all effects they (seem to) need,
- sometimes, difficulty in adding effects that would be useful,
- many different combinations of effects when combining code.

The pitfalls of excessive bottom-up design

Being precise about effects, and propagating them bottom-up leads to:

- a temptation to grant functions all effects they (seem to) need,
- sometimes, difficulty in adding effects that would be useful,
- many different combinations of effects when combining code.

Often, effects interact, and their interpretations interact. It is easier to reason about few sets of specific combinations.

Top-down design

- Think about what effects certain parts of your applications really need, and also what effects they should **not** need (such as general `IO`).
- Be (over-permissive) in allowing a function belonging to one component all these effects, but be cautious in adding new ones.
- Keep testing (different implementations of effects in mind at all stages).

Conclusions

- Abstract! Do not let your choice of effects library leak too much. Abstract both the implementation and the types.
- Use meaningful, application-oriented effects rather than overly generic ones.
- Push effects down rather than letting them bubble up.
- Do not unnecessarily commit to one implementation. Keep testing in mind.