Handling effects in monadic computations

(with no surprises)

Andrzej Rybczak

Monad transformers and mtl style effects

An `mtl` style effect is essentially:

1. A type class that defines a set of operations.
2. Several monad transformers, each providing an appropriate instance of said type class.

Why are they so prevalent in the Haskell ecosystem?

Monad transformers and mtl style effects

When learning Haskell, generally over time people:

- Get to know functions:
  - That produce no side effects, e.g. `showInt :: Int -> String`.
  - That can produce any side effect, e.g. `readInput :: Int -> IO String`.
  - That can produce some side effects, e.g. `flipSwitch :: State Bool ()`.

Monad transformers and mtl style effects

When learning Haskell, generally over time people:

- Get to know functions:
  - That produce no side effects, e.g. `showInt :: Int -> String`.
  - That can produce any side effect, e.g. `readInput :: Int -> IO String`.
  - That can produce some side effects, e.g. `flipSwitch :: State Bool ()`.
- Find out about the `transformers` library and play around with `ExceptT`, `ReaderT` and `StateT`.

Monad transformers and mtl style effects

When learning Haskell, generally over time people:

- Get to know functions:
  - That produce no side effects, e.g. `showInt :: Int -> String`.
  - That can produce any side effect, e.g. `readInput :: Int -> IO String`.
  - That can produce some side effects, e.g. `flipSwitch :: State Bool ()`.
- Find out about the `transformers` library and play around with `ExceptT`, `ReaderT` and `StateT`.
- Discover the `MonadTrans` type class and annoyance of manual lifting when stacking multiple monad transformers on top of each other.

Monad transformers and mtl style effects

When learning Haskell, generally over time people:

- Get to know functions:
  - That produce no side effects, e.g. `showInt :: Int -> String`.
  - That can produce any side effect, e.g. `readInput :: Int -> IO String`.
  - That can produce some side effects, e.g. `flipSwitch :: State Bool ()`.
- Find out about the `transformers` library and play around with `ExceptT`, `ReaderT` and `StateT`.
- Discover the `MonadTrans` type class and annoyance of manual lifting when stacking multiple monad transformers on top of each other.
- Check out the `mtl` library that generalizes operations of associated monad transformers to type classes and how using these alleviates the need of manual lifting.

Monad transformers and mtl style effects

When learning Haskell, generally over time people:

- Get to know functions:
  - That produce no side effects, e.g. `showInt :: Int -> String`.
  - That can produce any side effect, e.g. `readInput :: Int -> IO String`.
  - That can produce some side effects, e.g. `flipSwitch :: State Bool ()`.
- Find out about the `transformers` library and play around with `ExceptT`, `ReaderT` and `StateT`.
- Discover the `MonadTrans` type class and annoyance of manual lifting when stacking multiple monad transformers on top of each other.
- Check out the `mtl` library that generalizes operations of associated monad transformers to type classes and how using these alleviates the need of manual lifting.
- Realize that the signature of a function such as `flipSwitch` can be rewritten to `MonadState Bool m => m ()` and then it can be used in any monad that implements `MonadState Bool`, which allows for using it with various transformers that might behave differently.

Monad transformers and mtl style effects

When learning Haskell, generally over time people:

- Get to know functions:
  - That produce no side effects, e.g. `showInt :: Int -> String`.
  - That can produce any side effect, e.g. `readInput :: Int -> IO String`.
  - That can produce some side effects, e.g. `flipSwitch :: State Bool ()`.
- Find out about the `transformers` library and play around with `ExceptT`, `ReaderT` and `StateT`.
- Discover the `MonadTrans` type class and annoyance of manual lifting when stacking multiple monad transformers on top of each other.
- Check out the `mtl` library that generalizes operations of associated monad transformers to type classes and how using these alleviates the need of manual lifting.
- Realize that the signature of a function such as `flipSwitch` can be rewritten to `MonadState Bool m => m ()` and then it can be used in any monad that implements `MonadState Bool`, which allows for using it with various transformers that might behave differently.

Moreover, such functions can only perform monadic operations of type classes they are constrained to. This gives a lot of control over the code, which is very appealing.

Issues with mtl style effects

1. The need to define `O(n²)` type class instances for `n` effects. Solvable to an extent with default signatures or overlappable instances (requires the `monad-control` library for more complicated effects).

2. Boilerplate:

    - Each effect is a type class + a generic instance of said type class (or default implementations of its methods).

    - Each interpretation of an effect needs a separate monad transformer.

3. Poor performance (contrary to popular belief). In short: each `(>>=)` operation is `O(n)` function calls, where `n` is the size of the transformer stack (see excellent Effects for less talk by Alexis King for full analysis).

Issues with mtl style effects

1. The need to define `O(n²)` type class instances for `n` effects. Solvable to an extent with default signatures or overlappable instances (requires the `monad-control` library for more complicated effects).
2. Boilerplate:
    - Each effect is a type class + a generic instance of said type class (or default implementations of its methods).
    - Each interpretation of an effect needs a separate monad transformer.
3. Poor performance (contrary to popular belief). In short: each `(>>=)` operation is `O(n)` function calls, where `n` is the size of the transformer stack (see excellent Effects for less talk by Alexis King for full analysis).

These hurt, but there are several other subtle problems that can introduce bugs into your application.

Let's discuss them (in an increasing degree of seriousness).

State updates (1)

What is the output of the following program?

```
{-# LANGUAGE ScopedTypeVariables #-}
import Control.Exception (ErrorCall)
import Control.Monad.Catch
import Control.Monad.Trans.State

main :: IO ()
main = do
  s <- (`execStateT` (0::Int)) $ do
    (modify (+1) >> error "oops") `catch` \(e::ErrorCall) -> modify (+2)
  putStrLn $ show s
```

State updates (1)

What is the output of the following program?

```haskell
{-# LANGUAGE ScopedTypeVariables #-}
import Control.Exception (ErrorCall)
import Control.Monad.Catch
import Control.Monad.Trans.State

main :: IO ()
main = do
  s <- (`execStateT` (0::Int)) $ do
    (modify (+1) >> error "oops") `catch` \(e::ErrorCall) -> modify (+2)
  putStrLn $ show s
```

`3`, right?

State updates (1)

What is the output of the following program?

```
{-# LANGUAGE ScopedTypeVariables #-}
import Control.Exception (ErrorCall)
import Control.Monad.Catch
import Control.Monad.Trans.State

main :: IO ()
main = do
  s <- (`execStateT` (0::Int)) $ do
    (modify (+1) >> error "oops") `catch` \(e::ErrorCall) -> modify (+2)
  putStrLn $ show s
```

`3`, right?

No.

The answer is `2`. The problem is that state updates tracked by `StateT` within a computation wrapped in `catch` are discarded when an exception is raised.

There is no way to override this behavior.

This is confusing. Luckily it's easy to observe (unless the error is thrown very rarely and you won't test this particular code path), but at the very least it will cost you debugging time if you've never encountered this situation before.

State updates (2)

The initial state is `0`. What is the value of the state after `test` runs?

```haskell
{-# LANGUAGE FlexibleContexts #-}
import Control.Monad.Except
import Control.Monad.State

test :: (MonadError String m, MonadState Int m) => m ()
test = (modify (+1) >> throwError "oops") `catchError` \_ -> modify (+2)
```

After previous slide you will most likely be cautious about the answer.

State updates (2)

The initial state is `0`. What is the value of the state after `test` runs?

```
{-# LANGUAGE FlexibleContexts #-}
import Control.Monad.Except
import Control.Monad.State

test :: (MonadError String m, MonadState Int m) => m ()
test = (modify (+1) >> throwError "oops") `catchError` \_ -> modify (+2)
```

After previous slide you will most likely be cautious about the answer.

However, neither `2` nor `3` is correct! It depends on the transformer stack:

```
>>> putStrLn . show =<< (runExceptT . (`runStateT` (0::Int)) $ test)
Right ((),2)
>>> putStrLn . show =<< ((`runStateT` (0::Int)) . runExceptT $ test)
(Right (),3)
```

This is problematic, because:

1. You can't predict the behavior of code based on the definition of `test` alone.
2. Seemingly unrelated code change, i.e. rearranging the order of monad transformers in the stack will lead to subtle change of behavior in a completely different part of the application.

Resource handling

Concrete:

```
import Control.Exception

withResource
  :: (Resource -> IO a) -> IO a
withResource action = mask $ \unmask -> do
  r <- acquireResource
  a <- unmask (action r) `onException` do
    releaseResourceOnFailure r
  releaseResourceOnSuccess r
  pure a
```

Generalized:

```
import Control.Monad.Catch

withResource :: (MonadMask m, MonadIO m)
             => (Resource -> m a) -> m a
withResource action = mask $ \unmask -> do
  r <- acquireResource
  a <- unmask (action r) `onException` do
    releaseResourceOnFailure r
  releaseResourceOnSuccess r
  pure a
```

Do these functions work as expected?

Resource handling

Concrete:

```
import Control.Exception

withResource
  :: (Resource -> IO a) -> IO a
withResource action = mask $ \unmask -> do
  r <- acquireResource
  a <- unmask (action r) `onException` do
    releaseResourceOnFailure r
  releaseResourceOnSuccess r
  pure a
```

Generalized:

```
import Control.Monad.Catch

withResource :: (MonadMask m, MonadIO m)
             => (Resource -> m a) -> m a
withResource action = mask $ \unmask -> do
  r <- acquireResource
  a <- unmask (action r) `onException` do
    releaseResourceOnFailure r
  releaseResourceOnSuccess r
  pure a
```

Do these functions work as expected?

Concrete: **yes.** Generalized: **no.**

The generalized version won't release a resource when used in a monad containing the `ExceptT e` transformer if *action* throws an `ExceptT` specific error, because **it's not an exception**.

Resource handling

Concrete:

```haskell
import Control.Exception

withResource
  :: (Resource -> IO a) -> IO a
withResource action = mask $ \unmask -> do
  r <- acquireResource
  a <- unmask (action r) `onException` do
    releaseResourceOnFailure r
  releaseResourceOnSuccess r
  pure a
```

Generalized:

```haskell
import Control.Monad.Catch

withResource :: (MonadMask m, MonadIO m)
             => (Resource -> m a) -> m a
withResource action = mask $ \unmask -> do
  r <- acquireResource
  a <- unmask (action r) `onException` do
    releaseResourceOnFailure r
  releaseResourceOnSuccess r
  pure a
```

Do these functions work as expected?

Concrete: **yes.** Generalized: **no.**

The generalized version won't release a resource when used in a monad containing the `ExceptT e` transformer if *action* throws an `ExceptT` specific error, because **it's not an exception**.

The generalized version should use the `onError` function instead of `onException`. Notably, the documentation of the `exceptions` library mentions this issue. However, it's way too easy to introduce the version that will slowly leak resources by simply **changing the type signature of the concrete version** when refactoring.

A potential solution (1)

The ReaderT design pattern by Michael Snoyman.

A potential solution (1)

The ReaderT design pattern by Michael Snoyman.

Summary:

- Avoid `ExceptT`, `StateT` and `WriterT` because of their issues.
- Use the `ReaderT Env IO` as the main monad for your application with the `Env` data type that contains all configurable components.
- Store mutable state within `Env` in `IORef`S.
- Specialized monad transformers such as `ConduitT` are easily usable, but it's best to keep them to a small subset of your application.
- Use the `MonadUnliftIO` type class to work with higher-order `IO` functions like `withFile` (provided by the `unliftio` library).

A potential solution (1)

The ReaderT design pattern is simple and quite powerful.

Crucially, **it solves the performance problem of `mtl` style effects**. But:

- All your monadic functions can perform any side effect because they have `IO` access (unless you go back to `mtl` style constraints, at which point you reintroduce their issues mentioned earlier).
- When concurrency enters the picture, things suddenly become complicated.
    - Passing `Env` to multiple threads as a naive solution will work until you try to modify the state stored in `IORef`S, which are not thread safe.
    - You can use `MVar`S, but the state provided by `StateT` has a nice property of being thread local. You don't get any comparable solution with the same property here.

It's a good foundation, but a bit too bare bones by itself.

A potential solution (2)

What about extensible effect libraries?

There are quite a few of them, but they essentially fall into two groups:

1. Based on free monads, e.g. `freer-simple` and `polysemy`.
2. Based on monad transformers, e.g. `fused-effects`.

Admittedly, they solve some of the issues of `mtl` style effects like `O(n²)` instance problem and the need for boilerplate (to an extent), but introduce different ones:

- Poor interoperability with packages that make use of ubiquitous libraries such as `exceptions`, `monad-control` or `unliftio`.
- Limited expressiveness when compared to `mtl` style effects.

Moreover, performance still remains an issue (again, for full analysis please refer to the Effects for less talk by Alexis King).

A potential solution (3)

So, if:

- Effect libraries based on free monads and monad transformers are flawed.
- The ReaderT design pattern is fundamentally solid, but too bare bones.

Would it make sense to experiment with an extensible effect library based on the ReaderT design pattern?

A potential solution (3)

So, if:

- Effect libraries based on free monads and monad transformers are flawed.
- The ReaderT design pattern is fundamentally solid, but too bare bones.

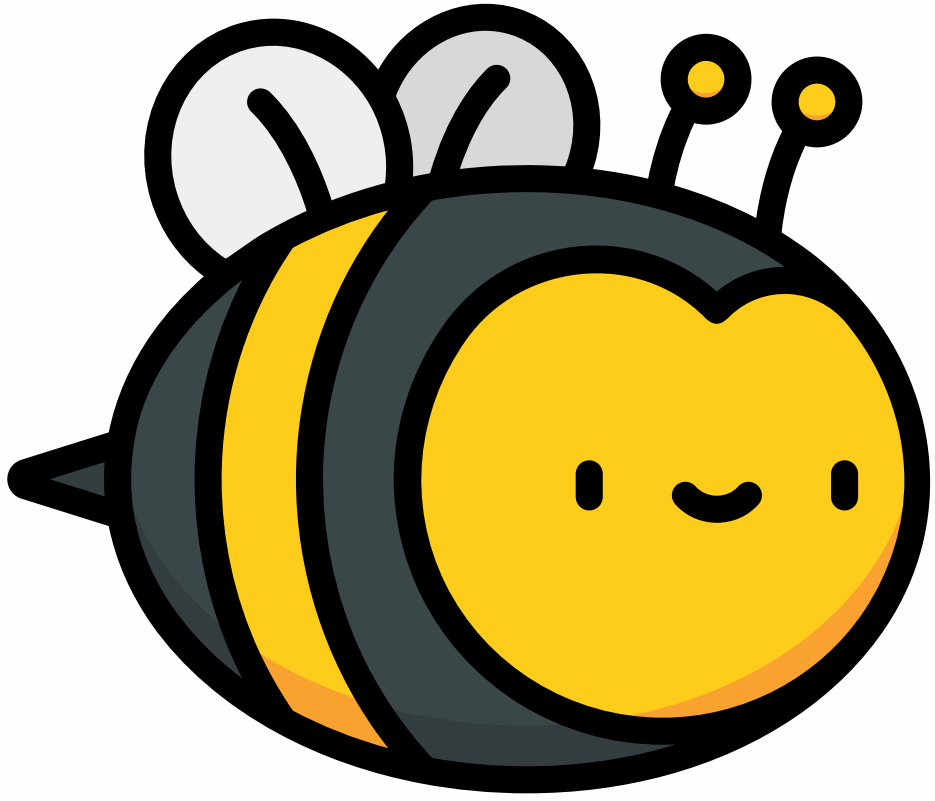
Would it make sense to experiment with an extensible effect library based on the ReaderT design pattern?

Yes!

Effectful

An extensible effect library based on the ReaderT design pattern

1. Very fast.
2. Expressive and provides a reasonably easy to understand API for working with effects (similar to the `MonadUnliftIO` type class).
3. Good integration with the existing Haskell ecosystem (`exceptions`, `monad-control`, `unliftio`).
4. No more lost state updates due to runtime exceptions (the transactional behaviour can be explicitly requested if needed).
5. Takes concurrency into account and provides thread local and shared versions of the `State` and `Writer` effects.
6. Introduces two types of effect dispatch:
    - Dynamic (interpretation switchable at run time).
    - Static (interpretation set at compile time).



Effectful

```
type Effect = (Type -> Type) -> Type -> Type

newtype Eff (es :: [Effect]) a = Eff (Env es -> IO a)
  deriving (Monoid, Semigroup)

instance Functor (Eff es)
instance Applicative (Eff es)
instance Monad (Eff es)
```

```
instance MonadThrow (Eff es)
instance MonadCatch (Eff es)
instance MonadMask (Eff es)
```

```
instance IOE :> es => MonadIO (Eff es)
instance IOE :> es => MonadUnliftIO (Eff es)
instance IOE :> es => MonadBase IO (Eff es)
instance IOE :> es => MonadBaseControl IO (Eff es)

instance Prim :> es => PrimMonad (Eff es)
```

```
instance NonDet :> es => Alternative (Eff es)
instance Fail   :> es => MonadFail (Eff es)
```

- The `Eff` monad:
  - Is the one and only monad of the library.
  - Keeps track of available effects at the type level.
  - Provides basic instances required by various libraries.
- The `Env` is an internal, mutable, thread local data type that holds everything needed for effect dispatch.
- The `Alternative` and `MonadFail` instances delegate to appropriate, dynamically dispatched effects.

Effectful (benchmarks)

The `effectful` compares performance of the most popular extensible effect libraries in two benchmarks:

- **countdown** - a microbenchmark counting down from `n` to `0` using a `State` effect that essentially measures performance of monadic binds and the effect dispatch.
- **filesize** - a more down to earth benchmark that repeatedly reads a file, computes its size and logs the result.

Effectful (benchmarks)

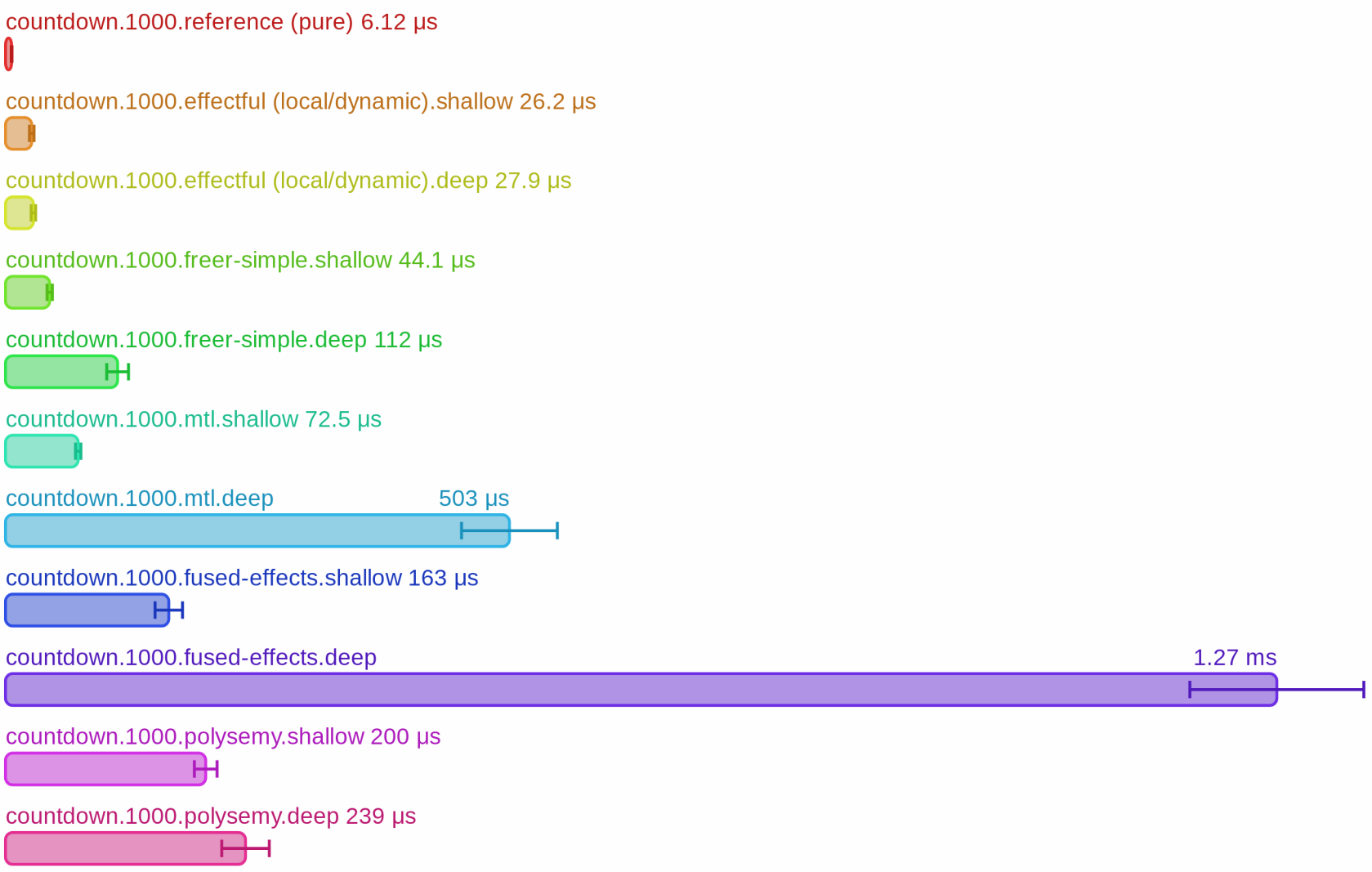
The `effectful` compares performance of the most popular extensible effect libraries in two benchmarks:

- **countdown** - a microbenchmark counting down from `n` to `0` using a `State` effect that essentially measures performance of monadic binds and the effect dispatch.
- **filesize** - a more down to earth benchmark that repeatedly reads a file, computes its size and logs the result.

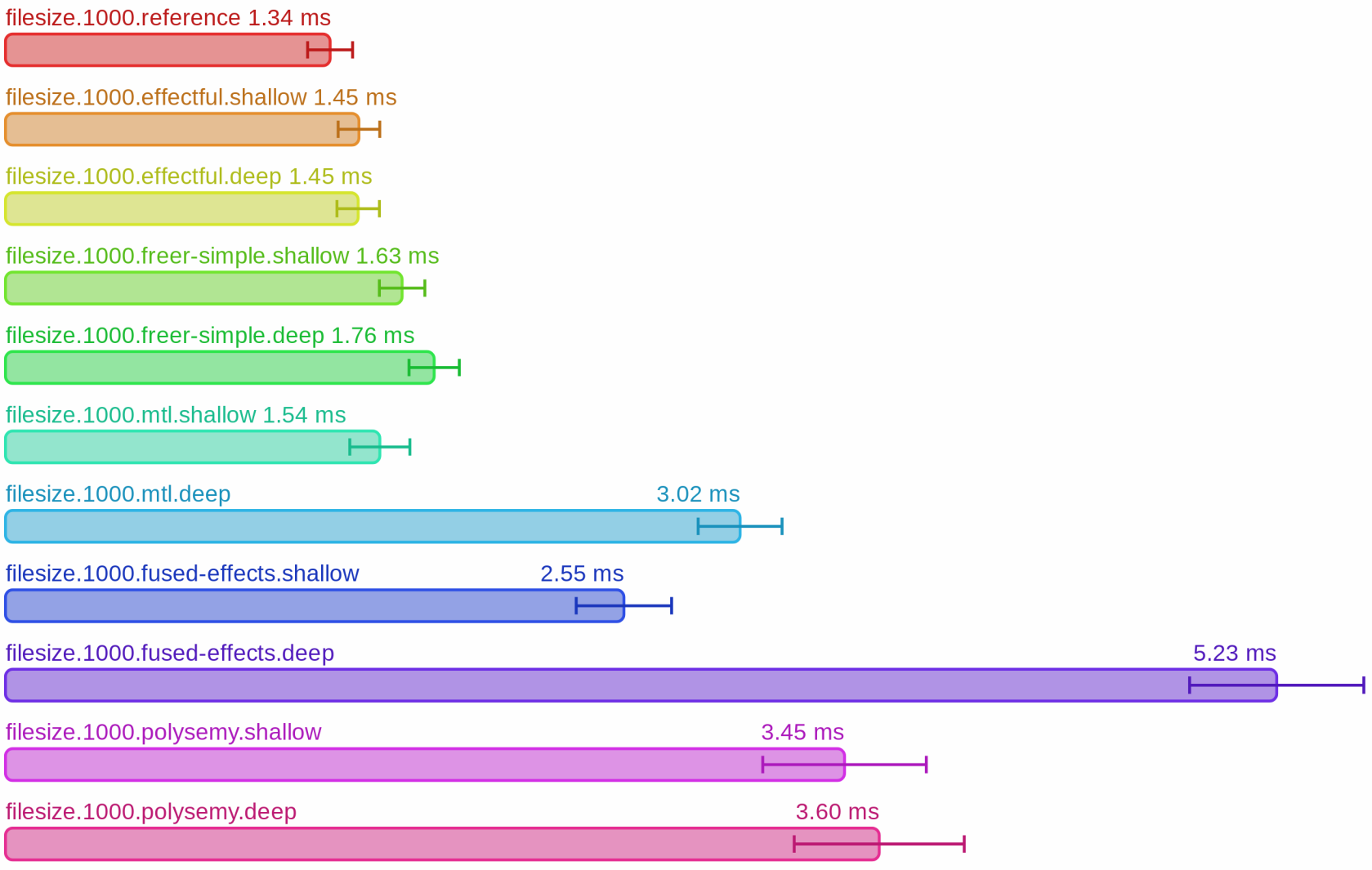
Moreover, each benchmark has two flavours that influence the amount of effects available in the context:

- **shallow** - only effects necessary for the benchmark:
  - **countdown** - the `State` effect.
  - **filesize** - the `Logging` and the `File` effect.
- **deep** - necessary effects + `5` redundant effects put into the context before and after the relevant ones (`10` in total). This simulates a typical scenario in which the code uses only a portion of the total amount of effects available to the application.

Effectful (the `countdown` benchmark)



Effectful (the `filesize` benchmark)



Effectful (expressiveness)

Consider the following `mtl` style effect:

```
class Monad m => MonadStore k v | m -> k v where
  update :: k -> (Maybe v -> m (Maybe v, r)) -> m r
```

- A generic instance is impossible to write because of limitations of the `MonadTransControl` type class.
- It cannot be expressed as a `polysemy` effect because of limitations of its machinery for higher order effects.

Effectful (expressiveness)

Consider the following `mtl` style effect:

```
class Monad m => MonadStore k v | m -> k v where
  update :: k -> (Maybe v -> m (Maybe v, r)) -> m r
```

- A generic instance is impossible to write because of limitations of the `MonadTransControl` type class.
- It cannot be expressed as a `polysemy` effect because of limitations of its machinery for higher order effects.

With `effectful`:

```
import Effectful
import Effectful.Dispatch.Dynamic
import Effectful.State.Static.Shared
import qualified Data.Map.Strict as Map

data Store k v :: Effect where
  Update :: k -> (Maybe v -> m (Maybe v, r)) -> Store k v m r
type instance DispatchOf (Store k v) = Dynamic

runStore :: forall k v es a. Ord k => Eff (Store k v : es) a -> Eff es a
runStore = reinterpret (evalState @(Map.Map k v) Map.empty) $ \localEs -> \case
  Update k action -> stateM $ \s -> localSeqUnlift localEs $ \unlift -> do
    (mv, r) <- unlift . action $ Map.lookup k s
    pure (r, maybe (Map.delete k s) (\v -> Map.insert k v s) mv)
```

Effectful

What's the catch?

Effectful

What's the catch?

The `Eff` monad doesn't support effect handlers that require the ability to suspend or capture the rest of the computation and resume it later (potentially multiple times).

Effectful

What's the catch?

The `Eff` monad doesn't support effect handlers that require the ability to suspend or capture the rest of the computation and resume it later (potentially multiple times).

This prevents `effectful` from providing (in particular):

- A `NonDet` effect handler that executes multiple `Alternative` branches and collects their results.
- A `Coroutine` effect.

Effectful

What's the catch?

The `Eff` monad doesn't support effect handlers that require the ability to suspend or capture the rest of the computation and resume it later (potentially multiple times).

This prevents `effectful` from providing (in particular):

- A `NonDet` effect handler that executes multiple `Alternative` branches and collects their results.
- A `Coroutine` effect.

The `eff` library by Alexis King can implement these by using delimited continuations underneath.

However, it faces other challenges that prevent it from being completed (see https://github.com/hasura/eff/issues for more information).

Effectful

What's the catch?

The `Eff` monad doesn't support effect handlers that require the ability to suspend or capture the rest of the computation and resume it later (potentially multiple times).

This prevents `effectful` from providing (in particular):

- A `NonDet` effect handler that executes multiple `Alternative` branches and collects their results.
- A `Coroutine` effect.

The `eff` library by Alexis King can implement these by using delimited continuations underneath.

However, it faces other challenges that prevent it from being completed (see https://github.com/hasura/eff/issues for more information).

My conjecture is that an extensible effect library in Haskell that uses delimited continuations underneath can't safely provide the `MonadUnliftIO` instance, which would significantly diminish its usability.

If that's the case, the approach `effectful` takes seems to be the sweet spot in terms of features vs restrictions.

Effectful (ecosystem)

The library is quite young, so the ecosystem is in the growth phase.

Already released packages that adapt existing libraries:

- `resourcet-effectful` - Deterministic allocation and freeing of resources.
- `crypto-rng-effectful` - Generation of cryptographically secure random bytes and numbers.
- `log-effectful` - Structured logging with multiple backends.
- `hpqtypes-effectful` - Access to a PostgreSQL database.
- `retry-effectful` - Combinators for monadic actions that may fail.

They're all lightweight wrappers and quite easy to write, so hopefully with time there will be many more ☺

Link to the GitHub organization: https://github.com/haskell-effectful

Acknowledgements

To all contributors of existing effect libraries: thank you for putting the time and effort to explore the space.

In particular, conversations in issue trackers of `cleff`, `eff`, `freer-simple`, `fused-effects` and `polysemy` repositories were invaluable in helping me discover, understand and solve some of the challenges in the space.

Without you `effectful` most likely wouldn't happen.

Thank you ☺

Questions?