Parsax + Reparsec

Streaming descriptive parsers

Problems

- Security
- Performance
- Documentation

Security

- Aeson package is based on HashMap
- HashMap is vulnerable to collision attacks
- When a HashMap has many collisions, you encounter linear or even exponential time complexity
- This makes it fairly straight-forward to DoS a server

Performance

- Switch to a Data.Map instead; that doesn’t have the same problem. It’s a balanced tree, not buckets with hashes
- Fine, but:
  - Data.Map’s memory use is more heavy
  - We still have the problem that someone could send us 1000 keys and we’re only interested in 3.
  - We’d prefer to ignore those keys, and optionally tell the user so.

YAML and JSON parsers don’t say what they want

- Yaml:
  - Ever had to google what you can put in your stack.yaml?
  - That sucks. There should be a stack --help-yaml that tells us exactly what it accepts.
- Aeson and the yaml package don’t let us do this:
  - They’re monadic.
  - You can’t generate a description of a monadic value like with an Applicative.
- Consider
  - Optparse-applicative -- self-describing
  - Yesod-forms -- self-describing (they generate an HTML form and also consume the form)

Proposal

- A streaming, self-describing, key ignoring, kitchen sink

User experience constraints

- We want an optparse-style UX
- We want full backtracking
- We want to do monadic checks
- We want to report invalid keys, but ignore them otherwise

ValueParser

```
data ValueParser e m a where

  Scalar :: Text -> (Scalar -> Either e a) -> ValueParser e m a

  Object :: ObjectParser e m a -> ValueParser e m a

  ObjectMap :: ValueParser e m a -> ValueParser e m (Map Text a)

  Array :: ValueParser e m a -> ValueParser e m [a]

  FMapValue :: (x -> a) -> ValueParser e m x -> ValueParser e m a

  AltValue :: NonEmpty (ValueParser e m a) -> ValueParser e m a

  PureValue :: a -> ValueParser e m a

  CheckValue :: (i -> m (Either e a)) -> ValueParser e m i -> ValueParser e m a
```

ObjectParser

```
data ObjectParser e m a where

  Field :: Text -> ValueParser e m a -> ObjectParser e m a

  LiftA2 :: (b -> c -> a) -> ObjectParser e m b -> ObjectParser e m c -> ObjectParser e m a

  FMapObject :: (x -> a) -> ObjectParser e m x -> ObjectParser e m a

  AltObject :: (NonEmpty (ObjectParser e m a)) -> ObjectParser e m a

  PureObject :: a -> ObjectParser e m a
```

Looks like

Object ((,) <$> Field “k1” (Scalar ..) <*> Field “k2” …)

ObjectParser IS Applicative and Semigroup for alternative.

ValueParser is NOT Applicative, a value is just one thing, not many. IS an Semigroup.

Full backtracking?

We want the freedom to write:

Array (Object (((,) <$> Field “x” int <*> Field “y” int) <>

((,) <$> Field “x” int <*> Field “z” int))

Or

Array (Object ((,) <$> Field “x” int <*> (Field “y” int <> Field “z” int))

Depending on our mood.

Implementation for streaming

- Consume SAX-style from a conduit sink
  - YAML: get a stream of events sink from the yaml package.
  - JSON: write a simple streaming SAX-style sink using Data.Aeson.Parser’s combinators.
- ConduitT i Event m ()
- Generalized data type for events on YAML/JSON-like inputs

Reparsec

- We need a resumable, backtracking parser for the SAX tokens
- Resumable like attoparsec
- Backtracks like attoparsec
- Supports any token and error type, like megaparsec

```
data Result m i e r

  = Done !i !Int !More !r

  | Failed !i !Int !More !e

  | Partial (Maybe i -> m (Result m i e r))
```

Pipeline

Source (parse the events) (JSON or YAML)

Filtering/shema checking

Consuming in a resumable parser

Finally producing a result with warnings

Data types

```
-- | A SAX event, containing either a scalar, array or
object with keys.

data Event

  = EventScalar !Scalar

  | EventArrayStart

  | EventArrayEnd

  | EventObjectStart

  | EventObjectKey !Text

  | EventObjectEnd

  deriving (Show, Eq)
```

```
-- | A constant atomic value.

data Scalar

  = ScientificScalar !Scientific

  | TextScalar !Text

  | BoolScalar !Bool

  | NullScalar

  deriving (Show, Eq)
```

We make a schema from the parsers

This is used to ignore keys and report warnings for them.

```
data Schema =
  Schema
    { schemaScalar :: Bool
    , schemaObject :: Maybe (Map Text Schema)
    , schemaArray :: Maybe Schema
    }
```

Schema enforcing

```
enforceSchema ::
    Monad m
  => Maybe Schema
  -> ConduitT Event Event m (SchemaValidation, Seq ParseWarning)
```

Streaming into reparsec

```
loop parser = do
  mevent <- await
  result <- lift (parser (fmap pure mevent))
  case result of
    Partial resume -> do
      loop resume
    Failed remaining pos _more errors -> do
      leftovers <- makeLeftovers pos remaining
      pure (Left errors, makeWarnings leftovers)
    Done remaining pos _more a -> do
      leftovers <- makeLeftovers pos remaining
      pure (Right a, makeWarnings leftovers)
  where
    makeLeftovers pos remaining = do
      let leftovers = Seq.drop pos remaining
      mapM_ leftover leftovers
      pure leftovers
    makeWarnings leftovers =
      if Seq.null leftovers
        then mempty
        else pure (LeftoverEvents leftovers)
```

Brainmelt

- Parsing fields in alternatives with different types
- The fields come in random order, any order is fine
- What do?

Parsing fields in alternatives with different types

F <$> Field “x” int <|> (Y <$> Field “y” text <*> Field “x” text)

x: “1”
y: “2”

Or

y: “1”
x: “2”

Implementation

```
data EitherKey s e a where
  EitherKey :: !Text -> !(Vault.Key s (Either (ParseError e) a)) -> EitherKey s e a

data ParserPair e m s where
  ParserPair :: EitherKey s e a -> ValueParser e m a -> ParserPair e m s

data MappingSM s e m a = MappingSM
  { msmAlts :: !(Free.Alt (EitherKey s e) a)
  , msmParsers :: !(Map Text (NonEmpty (ParserPair e m s)))
  , msmVault :: !(Vault.Vault s)
  }
```

Making the free alt

```
toMappingSM :: ObjectParser e m a -> ST s (MappingSM s e m a)
toMappingSM mp = do
  (alts, parsers) <- runStateT (go mp) mempty
  pure MappingSM {msmAlts = alts, msmParsers = parsers, msmVault = Vault.empty}
  where
    go ::
         ObjectParser e m a
      -> StateT (Map Text (NonEmpty (ParserPair e m s))) (ST s) (Free.Alt (EitherKey s e) a)
    go (PureObject a) = pure $ pure a
    go (FMapObject f x) = do
      x' <- go x
      pure $ fmap f x'
    go (LiftA2 f a b) = do
      a' <- go a
      b' <- go b
      pure $ liftA2 f a' b'
    go (AltObject xs) = do
      xs' <- mapM go xs
      pure $ asum1 xs'
    go (Field t p) = do
      key <- lift $ EitherKey t <$> Vault.newKey
      let pp = ParserPair key p
      modify' $ M.insertWith (flip (<>)) t (pp :| [])
      pure $ Free.Alt (pure (toAltF key))
    toAltF x = Free.Ap x (pure id)
```

Running the alternatives

```
objectReparsec ::
     PrimMonad m
  => MappingSM s e m a
  -> Text
  -> ParserT (Seq Event) (ParseError e) m (MappingSM s e m a)
objectReparsec msm textKey = do
  case M.lookup textKey (msmParsers msm) of
    Nothing -> pure msm
    Just xs -> do
      updateVault <- foldMap1 makeAttempt xs
      pure
        msm
          { msmParsers = M.delete textKey (msmParsers msm)
          , msmVault = updateVault (msmVault msm)
          }
  where
    makeAttempt (ParserPair (EitherKey _ key) valueParser) = do
      result <- valueReparsec valueParser
      pure (Vault.insert key (Right result))
```

Finalizing the result

```
finishObjectSM :: forall s b e m. MappingSM s e m b -> Validation (ParseError e) b
finishObjectSM msm = Free.runAlt go (msmAlts msm)
  where
    go :: forall a. EitherKey s e a -> Validation (ParseError e) a
    go (EitherKey keyText key) =
      maybe
        (Failure (NoSuchKey keyText))
        (either Failure Success)
        (Vault.lookup key (msmVault msm))
```

Documentation generation

```
stackLikeGrammar :: ValueParser Text m (Int, [Either Int Int])

stackLikeGrammar = Object ((,) <$> yfield <*> (xfield <> zfield))

y: <scalar>

# one of

z: scalar

# or
```

```
putStrLn $ drawForest $ valueForest stackLikeGrammar
Object
|
+- Key "y"
|  |
|  `- Scalar
|
`- OneOf
   |
   +- Key "z"
   |  |
   |  `- Scalar
   |
   `- Key "x"
      |
      `- Array
         |
         `- OneOf
            |
            +- Scalar package-name
            |
            `- Object: Git reference
               |
               `- Key "location"
                  |
                  `- Scalar
```

Code

https://github.com/chrisdone/streaming-parsers

Test suite		Easy
High-level GADT for parser		Fairly straight-forward
Avoiding collision exploits		Difficult
Resumable parsing with backtracking		Difficult
Output warnings of ignored object keys		Easy
Custom errors		Straight-forward
Running m (IO/RIO) actions in a user parser		Fairly straight-forward
YAML backend		Straight-forward
JSON backend		Fairly straight-forward
Documentation generator		Straight-forward
Include line/col info in errors	-	Fairly easy
Supplying extra limits (array length, etc.)	-	Easy
Get code coverage to near 100%	-	Detailed
Memory/allocation tests	-	Fairly straight-forward