an Interface for accessing Environment Variables

The way, that programs usually interact with *environment variables*, can be expressed beautifully in *Haskell*.

For me *dependent types* are the most natural way to semantically reason about programs. Unfortunately dependently typed languages are not really production-ready yet. *Haskell* and *GHC* often come close enough.

The source code of my homepage has become a playground for exploring new ideas in Haskell. And that's also where this idea originated from.

Example use case

Usually programs interact with a fixed set of environment variables. In this depiction environment variables have a *name*, a *content type* and a *default value*.

An example of environment variables used by an HTTP server.

`HOMEPAGE_CONFIG_FILE`

: A file path, where the configuration is read from. Defaults to `"./homepage.json"`.

`HOMEPAGE_LOG_FILE`

: A file path, where the log is appended. When unset the log is piped to *StdOut*.

`HOMEPAGE_LOG_LEVEL`

: The minimum log level. Messages with an even lower log level will be discarded. Defaults to `"LevelDebug"`.

Furthermore we will need a parser for each environment variable.

Dependently typed approach

In a dependently typed language, like Idris, we can just use a simple data type and functions.

We use the sum-type `EnvVarKind` to identify each environment variable.

A data type representing an environment variable.

```
data EnvVarKind = EnvVarConfigFile
                | EnvVarLogFile
                | EnvVarLogLevel
```

Now we can pattern-match on any `EnvVarKind`. The exhaustiveness-checker of Idris would warn us

about missing cases.

The name of an environment variable is a simple function.

```
EnvVarName : EnvVarKind -> String
EnvVarName = \case
  EnvVarConfigFile => "HOMEPAGE_CONFIG_FILE"
  EnvVarLogFile => "HOMEPAGE_LOG_FILE"
  EnvVarLogLevel => "HOMEPAGE_LOG_LEVEL"
```

The type of an environment variable is already a bit special.

```
EnvVarType : EnvVarKind -> Type
EnvVarType = \case
  EnvVarConfigFile => FilePath
  EnvVarLogFile => Maybe FilePath
  EnvVarLogLevel => LogLevel
```

The function `EnvVarType` returns a value of type `Type`. We can now use this in `defaultEnvVar` and `parseEnvVar`.

The default value of an environment variable is a dependently typed function.

```
defaultEnvVar : (x : EnvVarKind) -> EnvVarType x
defaultEnvVar = \case
  EnvVarConfigFile => "./homepage.json"
  EnvVarLogFile => Nothing
  EnvVarLogLevel => LevelDebug
```

The parser of an environment variable is also a dependently typed function.

```
parseEnvVar : (x : EnvVarKind) -> String -> Maybe (EnvVarType x)
parseEnvVar = \case
  EnvVarConfigFile => parseFilePath
  EnvVarLogFile => fmap Just . parseFilePath
  EnvVarLogLevel => readMaybe
```

With this setup we can access the final value through a function, which we initialize during startup of our application.

```
acquireEnvironment : IO ((x : EnvVarKind) -> EnvVarType x)
acquireEnvironment = undefined -- TODO: Left out for brevity.
```

The actual implementation is left out here, but the idea goes as follows.

For each environment variable, lookup the `EnvVarName`. When there is a match, parse the content using `parseEnvVar`. When parsing fails or when there is no match use `defaultEnvVar`. Add a case for

each environment variable in the resulting function and return the value. The exhaustiveness-checker will come in handy again.

Haskell approach

Unfortunately in Haskell we don't have dependent types. We can still get very close with todays language extensions (mainly *DataKinds*, *GADTs* and *FunctionalDependencies*).

`KnownEnvVar` *uses functional dependencies and sets the same laws as the previous section.*

```
class KnownSymbol name => KnownEnvVar (envVar :: EnvVarKind name value)
    | name -> envVar, envVar -> name, envVar -> value where
  parseEnvVar :: Proxy name -> String -> Maybe value
  defaultEnvVar :: Proxy name -> value
  caseEnvVar :: Proxy name -> EnvVarKind name value
```

In Haskell we can't use simple functions like `EnvVarName` and `EnvVarType`. Instead we use functional dependencies `envVar -> name` and `envVar -> value`.

> **NOTE** Additionaly we use `name -> envVar` to ensure, that we have exactly one `envVar` for each `name`. There are more details in the Appendix.

`name` and `value` are now required as type parameters to `EnvVarKind` to bring them into scope for the methods. Aside from the additional type level information, `EnvVarKind` has exactly the same constructors as before.

`EnvVarKind` *is now a GADT.*

```
data EnvVarKind :: Symbol -> Type -> Type where
  EnvVarConfigFile :: EnvVarKind "HOMEPAGE_CONFIG_FILE" FilePath
  EnvVarLogFile :: EnvVarKind "HOMEPAGE_LOG_FILE" (Maybe FilePath)
  EnvVarLogLevel :: EnvVarKind "HOMEPAGE_LOG_LEVEL" LogLevel
```

The information, that we previously encoded with simple functions, now becomes available on type-level through instances of `KnownEnvVar`.

Instances for `KnownEnvVar` *are required for each constructor of* `EnvVarKind`.

```
instance KnownEnvVar 'EnvVarConfigFile where
  parseEnvVar _ = parseFilePath
  defaultEnvVar _ = "./homepage.json"
  caseEnvVar _ = EnvVarConfigFile

instance KnownEnvVar 'EnvVarLogFile where
  parseEnvVar _ = fmap Just . parseFilePath
  defaultEnvVar _ = Nothing
  caseEnvVar _ = EnvVarLogFile
```

```
instance KnownEnvVar 'EnvVarLogLevel where
  parseEnvVar _ = readMaybe
  defaultEnvVar _ = LevelDebug
  caseEnvVar _ = EnvVarLogLevel
```

Finally the function we use to access an environment variable stays pretty much the same.

```
acquireEnvironment :: IO (forall name value. EnvVarKind name value -> value)
acquireEnvironment = undefined -- TODO: Left out for brevity.
```

TIP It's tempting to use *TypeFamilies* instead of *FunctionalDependencies* and remove the type parameters from `EnvVarKind`. Unfortunately this makes it impossible to implement the accessor function in `acquireEnvironment`.

To easily use this accessor function an mtl-style class can make sense. Here is an example of the usage of such a class.

The Haskell snippets from this article are used here. `acquireEnvironment` is actually implemented here.

Appendix A: Uniqueness of names

The functional dependency `name → envVar`, that we can use in Haskell, is very powerful and would require a bit more effort with dependent types. Together with `envVar → name` it enforces that names and environment variables are in a one-to-one relationship.

Without the `name → envVar` dependency there could be two references to the same environment variable name.

In a dependently typed language we would have to use a proof to ensure this property actually holds. We want `EnvVarName` to be injective.

Proving uniqueness of names in Idris.

```
ResolveEnvVarName : String -> Maybe EnvVarKind
ResolveEnvVarName = \case
  "CONFIG_FILE" => Just EnvVarConfigFile
  "LOG_FILE" => Just EnvVarLogFile
  "LOG_LEVEL" => Just EnvVarLogLevel
  _ -> Nothing

proofUniqueName : {x : EnvVarKind} -> Just x = ResolveEnvVarName (EnvVarName x)
proofUniqueName = case x of
  EnvVarConfigFile => Refl
  EnvVarLogFile => Refl
  EnvVarLogLevel => Refl
```