HASKELL MINI-PATTERNS HANDBOOK

Date: August 17, 2020
Updated: September 7, 2020
Author: Dmitrii Kovanikov <> Veronika Romashkina

haskell tutorial patterns

Navigating in the ocean of Haskell possibilities is challenging even though Haskell is a powerful language that helps to implement robust and maintainable programs. The language supplies you with tons of awesome approaches, but it is not always trivial to see how and where to use them properly.

Fortunately, like any other mainstream programming language, Haskell also has its best-practices and recommended ways for producing high-quality code. Knowing Haskell programming patterns helps you create better libraries and applications and make their users more pleased. And, yes, Haskell actually has FP-oriented programming patterns in addition to the best-practices shared with other languages.

If you are aware of the best ways to solve common problems, you can become a better Haskell developer by using more efficient programming techniques specific to the language. Likewise, you can apply Haskell-specific patterns successfully outside of Haskell.

This blog post contains a structured collection of some programming mini-patterns in Haskell with the detailed description and examples, some small “quality of life” improvements that would help everyone on their developer journey.

> 📚 In this article, each pattern contains at least one task with the hidden solution, so you can test your understanding of a pattern by solving the proposed task.

Newtype

Pattern	Newtype
Description	Lightweight data wrapper.
When to use	When using the same primitive type (`Int`, `Text`, etc.) to represent semantically different entities (name, title, description, etc.).
Benefits	1. Improves maintainability. 2. Increases code readability. 3. Enables writing custom instances. 4. Allows reusing instance definitions with DerivingVia.
Costs	1. Additional wrapping and unwrapping. 2. Deriving boilerplate is required to duplicate existing behaviour of the underlying type.

One of the most common and useful Haskell features is `newtype`. `newtype` is an ordinary data type with the name and a constructor. However, you can define a data type as `newtype` instead of `data` only if it has **exactly** one constructor with **exactly** one field.

It is extremely easy to define a `newtype` in Haskell as no extra effort is required from the user compared to the data type declaration. For example, the following are the valid `newtype` definitions:

```haskell
-- + Valid definition of a wrapper around Double
newtype Volume = Volume Double

-- + Also valid newtype definition with the record field
newtype Size = Size
    { unSize :: Int
    }
```

But you can’t declare the following data types as a `newtype`:

```haskell
-- - Invalid newtype definition: more than 1 field
newtype Point = Point
    { pointX :: Int
    , pointY :: Int
    }

-- - Ivalid newtype definition: more than 1 constructor
newtype Shape
    = Circle Double
    | Square Double
```

`newtype` s have a lot of great benefits but I'm going to focus only on the *code maintainability* in this section.

You can think of a `newtype` as a lightweight data type. Newtype has the same representation in memory as the wrapped type. Wrapping and unwrapping are also free operations in runtime. So, `newtype` is a **zero-cost** abstraction.

Notwithstanding from the compiler point of view, different `newtype` s are different data types. And since they don't have any runtime overhead compared to the ordinary data types, you can provide a safer interface to your API without sacrificing performance.

Shall we look at the example now. Consider, that you have a function which takes a password, a hash and verifies whether the given hash is the hash of the given password. Since both password and hash are merely textual data, we could write the following function:

```haskell
validateHash :: ByteString -> ByteString -> Bool
```

The problem with this function is that you can easily pass arguments in a different order and get faulty results. And you need to think about proper order of arguments each time you are calling this function. Should it be `validateHash password hash` or `validateHash hash password`? This approach is error-prone.

But if you define the following `newtype` s instead:

```haskell
newtype Password = Password
    { unPassword :: ByteString
    }

newtype PasswordHash = PasswordHash
    { unPasswordHash :: ByteString
    }
```

You can implement a more type-safe version of the `validateHash` function, and additionally this improves the code readability:

```haskell
validateHash :: Password -> PasswordHash -> Bool
```

Now, `validateHash hash password` is a compile-time error, and it makes it impossible to confuse the order of arguments.

Another popular way of increasing code readability without sacrificing performance is introducing a new alias (which is simply another name for a data type) by using the `type` keyword. Unfortunately, this approach is only a partial solution since it helps only a developer, not the compiler as the compiler doesn't see the difference between type alias and type itself. And if you don't help your compiler, the compiler won't be there for you when you need it.

For example, the following code is flawed:

```haskell
type Attack  = Int
type Defense = Int

calculateDamage :: Attack -> Defense -> Attack
```

Of course, the above type signature is better than `Int -> Int -> Int`, but the compiler sees it exactly like this. So, you still can write `calculateDamage monsterDefense playerAttack` and get a runtime bug.

The approach of using `type`s instead of `newtype` can bring even more damage when you have a lot of similar data types. The more types you have the harder to maintain them without external help. Below you can see a code sample from one of the Haskell libraries:

```haskell
type WorkerId = UUID
type SupervisorId = UUID
type ProcessId = UUID
type ProcessName = Text
type SupervisorName = Text
type WorkerName = Text
```

The library safety can be improved by replacing all these type aliases with `newtype`s, and it can even help to discover some bugs that happen due to passing arguments in the wrong order.

Moreover, the `newtype` approach is more flexible since you can provide your custom instances or restrict some instances allowing you to create values in an unsafe way.

The cost of using `newtype` is small — you only need to wrap and unwrap it when necessary. But the benefits hugely outweigh this small price.

Newtype: Task

Improve the following code by applying the **Newtype** pattern.

```haskell
data Player = Player
    { playerHealth    :: Int
    , playerArmor     :: Int
    , playerAttack    :: Int
    , playerDexterity :: Int
    , playerStrength  :: Int
    }

calculatePlayerDamage :: Int -> Int -> Int
calculatePlayerDamage attack strength = attack + strength

calculatePlayerDefense :: Int -> Int -> Int
calculatePlayerDefense armor dexterity = armor * dexterity

calculateHit :: Int -> Int -> Int -> Int
calculateHit damage defense health = health + defense - damage

-- The second player hits first player and the new first player is returned
hitPlayer :: Player -> Player -> Player
hitPlayer player1 player2 =
    let damage = calculatePlayerDamage
            (playerAttack player2)
            (playerStrength player2)
        defense = calculatePlayerDefense
            (playerArmor player1)
            (playerDexterity player1)
        newHealth = calculateHit
            damage
            defense
            (playerHealth player1)
    in player1 { playerHealth = newHealth }
```

Show solution

Smart constructor

Pattern	Smart constructor
Description	Providing idiomatic ways for constructing values.
When to use	1. When a data type restricts some values (e.g. not every `ByteString` is a valid `Password`). 2. When you want to make construction of big data types easier. 3. To avoid runtime errors. 4. To make illegal states unrepresentable.
Benefits	1. More structured and maintainable code. 2. Separation of concepts. 3. Control of erroneous data inputs.
Costs	1. Some extra code. 2. Decide on the approach details. 3. Inability to define instances outside of the module.

Once you have a `newtype`, first you need to create its value to work with it. And sometimes you want to validate a value before proceeding with it. Usually you would create a value of a data type by using its constructor. However, when programming in a modular way, you want to have boundaries in your interface and avoid providing unsafe ways of constructing values without validation.

This programming pattern in Haskell is called **smart constructor**. It can be understood better by looking at the implementation of such approach based on the `Password` data type:

```haskell
module Password
    ( Password
    , unPassword
    , mkPassword
    ) where

import Data.ByteString (ByteString)
import qualified Data.ByteString as ByteString


newtype Password = Password ByteString

unPassword :: Password -> ByteString
unPassword (Password password) = password

-- | Smart constructor. Doesn't allow empty passwords.
mkPassword :: ByteString -> Maybe Password
mkPassword pwd
    | ByteString.null pwd = Nothing
    | otherwise = Just (Password pwd)
```

In this module, we want to reject empty passwords. This is what the `mkPassword` function does. We disincline to export the `Password` constructor.

But we need a way to deconstruct a value of type `Password`. Unfortunately, we can't define a record field of our `newtype` because it allows to change values using record update syntax. Hence the `unPassword` function.

> 👩‍🔬 It is important to hide the constructor to forbid value coercion.

Even if you don't allow creating unvalidated passwords, you may need to create passwords in your test-suite without extra hassle. So we could create a function `unsafePassword` with a hint in the name.

```haskell
-- | Used in testing.
unsafePassword :: ByteString -> Password
unsafePassword = Password
```

If you notice usages of this function in your application code during code review, something is clearly wrong. Moreover, it is even possible to set up some automatic tooling to perform such checks for you.

You can find multiple variations of this pattern in the wild that differ in some implementation details:

- Usage of PatternSynonyms
- Replacing `Maybe` with Either or Validation types for better error-reporting
- Rename the constructor to `UnsafePassword` instead of having a separate `unsafePassword` function (however, this is a less safer approach due to coercions)
- Validation of the statically-known values during compile-time, so you get errors in compile-time instead of runtime, and you don't need unsafe functions when you know that the values are valid
- Others

Unfortunately, the community has not reached the consensus on what is the only true way to implement smart constructors. All ways slightly differ in ergonomics and naming schemes, they all do their job and fit various use cases of different systems. But the general idea remains the same across all approaches.

Smart constructor: Task

Improve the following code by applying the **Smart constructor** pattern.

```haskell
module Tag where

-- | Tag is a non-empty string.
newtype Tag = Tag String

mkTag :: String -> Tag
mkTag tag
    | null tag = error "Empty tag!"
    | otherwise = Tag tag
```

```haskell
module TagsList where

import Data.List.NonEmpty (NonEmpty (..))
import Tag (Tag, mkTag)


-- | Non-empty list of non-empty tags.
newtype TagsList = TagsList (NonEmpty Tag)

mkTagsList :: [String] -> TagsList
mkTagsList [] = error "Empty list of tags"
mkTagsList (tag:tags) = TagsList $ mkTag tag :| map mkTag tags
```

Show solution

Evidence

Pattern	Evidence
Description	Replacing boolean blindness with the validation witness.
When to use	1. Always. But most importantly, when you want to make sure that data is validated or you want to reuse that knowledge in the future. 2. To make illegal states unrepresentable.

Benefits	1. More robust code. 2. Better maintainability (e.g. easier to refactor). 3. More context in data. 4. Better error-messages support.
Costs	1. Requires writing code in a slightly different way. 2. More code.

This topic naturally completes the previous pattern of “Smart constructors”. The approach was covered before in various excellent blog posts:

- Overcoming Boolean blindness with Evidence
- Code smell: Boolean blindness
- Parse, don’t validate

All above posts provide an amazing description and explanation of the **Evidence** pattern. Here we would like to add only a short overview with a small example.

For our example, let’s have a look at the function that sorts a list. Its type in Haskell looks like this:

```haskell
sort :: Ord a => [a] -> [a]
```

By sorting a list, we gained a knowledge that now all elements in the list are in the increasing (or decreasing) order. But the type of a sorted list is the same as the type of any other list. So there is no way to know in advance, whether a list is sorted or not.

Fortunately, this problem can be solved. We are going to follow Newtype and Smart constructor patterns:

```haskell
newtype SortedList a = SortedList [a]

sort :: Ord a => [a] -> SortedList a
```

> 👩‍🔬 It is possible to use more sophisticated techniques (such as dependent types) to ensure that the list is sorted by construction, but this approach has its own pros and cons.

By wrapping the list into the newtype we record (and can provide later) an *evidence* of the list elements order. It may be important to have this knowledge, because if you know that the list is sorted, you can implement some functions more **efficiently** than for the ordinary lists, e.g.:

1. Find the minimum of a list.
   ```haskell
   minimum :: SortedList a -> Maybe a
   ```
2. Keep only unique elements in the list.
   ```haskell
   nub :: Eq a => SortedList a -> SortedList a
   ```
3. Merge two sorted list into a new sorted list.
   ```haskell
   merge :: Ord a => SortedList a -> SortedList a -> SortedList a
   ```

Unfortunately, even if you follow the **Evidence** pattern in types, you still can misuse it in values. Consider the following code that throws away the knowledge about given evidence:

```haskell
add :: (a -> Maybe Int) -> (a -> Maybe Int) -> a -> Maybe Int
add f g x =
    if isNothing (f x) || isNothing (g x)
    then Nothing
    else Just (fromJust (f x) + fromJust (g x))
```

Writing such code is a sign that you are following the boolean blindness anti-pattern and it is time to refactor your code immediately.

The key issue here is that by calling a function that returns `Bool` you lose the information about earlier performed validation. Instead, you can keep this information by explicitly pattern-matching on the validation or result.

> 📚 **Exercise:** Try refactoring the above code without using `isNothing` and `fromJust` functions. Bonus points for using `Maybe` as a `Monad`.

Returning to our previous example with the `validateHash` function:

```haskell
validateHash :: Password -> PasswordHash -> Bool
```

You still can forget to call this validation function in the application code and allow users to work with invalid passwords.

```haskell
getUserPage, accessDenied :: User -> IO Page

loginUser :: User -> Password -> PasswordHash -> IO Page
loginUser user pwd pwdHash =
    if validateHash pwd pwdHash
    then getUserPage user
    else accessDenied user
```

The above code has two problems:

1. Even in the `else` branch you can call `getUserPage` and allow users to enter with the invalid passwords.
2. You can simply forget to call `validateHash` whenever it is needed.

One more type-safe approach (but also a bit more heavyweight solution) would be to return some sort of witness for the fact that a password was validated, and then require this witness in the future functions.

To implement this solution, first, we need to change the type of `validateHash`:

```haskell
-- opaque data type that can be created using only 'validateHash'
data UserAuth

-- Instead of returning 'Bool' we return 'UserAuth'
validateHash :: Password -> PasswordHash -> Maybe UserAuth
```

Now we change the type of `getUserPage` to take `UserAuth` as a required parameter:

```haskell
getUserPage :: UserAuth -> User -> IO Page
```

Since `UserAuth` can be created only with `validateHash`, the only way to return a user page is by performing password validation:

```haskell
loginUser :: User -> Password -> PasswordHash -> IO Page
loginUser user pwd pwdHash = case validateHash pwd pwdHash of
    Nothing   -> accessDenied user
    Just auth -> getUserPage auth user
```

You can see how we made the code safer and more robust by a small change. And, as always, there are multiple ways of solving this problem including usages of more advanced Haskell features for providing better guarantees, with better ergonomics or for solving more difficult problems. For example, the following blog post describes implementation of a similar problem using more advanced Haskell features:

- Type Witness in Haskell

Evidence: Task

Improve the following code by applying the **Evidence** pattern.

```haskell
import Data.IntMap (IntMap)
import Data.Maybe (fromJust)

import qualified Data.IntMap as IntMap


getNearestValues
    :: IntMap Double  -- ^ Map from positions to values
    -> Int            -- ^ Current position
    -> Double
getNearestValues vals pos
    -- both positions are in Map: returns sum of them
    | IntMap.member (pos - 1) vals && IntMap.member (pos + 1) vals =
      fromJust (IntMap.lookup (pos - 1) vals) + fromJust (IntMap.lookup (pos + 1)
vals)

    -- only left position is in Map
    | IntMap.member (pos - 1) vals =
      fromJust (IntMap.lookup (pos - 1) vals)

    -- only right position is in Map
    | IntMap.member (pos + 1) vals =
      fromJust (IntMap.lookup (pos + 1) vals)

    -- no neighbours in map
    | otherwise = 0.0
```

Show solution

Make illegal states unrepresentable

Pattern	Make illegal states unrepresentable
Description	Data types precisely describe the domain allowing to create all valid values and making it impossible to construct invalid values.
When to use	1. To restrict domain of possible values. 2. When it is feasible to describe the domain precisely.
Benefits	1. Correctness. 2. Need to write fewer tests. 3. Harder to introduce bugs for people not knowing the whole context of a big picture.
Costs	1. May require writing custom helper data types and functions. 2. Code can increase in size. 3. Harder to introduce logic changes.

This pattern is closely related to smart constructor and evidence patterns, and they can and should be used together.

The *make illegal states unrepresentable* motto is well-known in the FP community. Functional Programming features such as Algebraic Data Types (ADT), parametric polymorphism and others allow describing the shape of the valid data more precisely to the extent that it is impossible to construct any invalid values.

To give a simple example of this idea, consider a function that takes two optional values and does something with those values, but only when both values are present. The function assumes that you won't pass only a single value without the second element, so it doesn't bother to handle such cases.

```haskell
handleTwoOptionals :: Maybe a -> Maybe b -> IO ()
handleTwoOptionals (Just a) (Just b) = ... work with 'a' and 'b'
handleTwoOptionals Nothing Nothing = ... proceed without values
handleTwoOptionals _ _ = error "You must specify both values"
```

The type of `handleTwoOptionals` allows passing `Just a` and `Nothing :: Maybe b`, but in reality the function doesn't process such a combination. You can notice that it is straightforward to fix this particular problem by changing types barely: instead of passing two `Maybe` s separately, you need to pass `Maybe` of a pair.

```haskell
handleTwoOptionals :: Maybe (a, b) -> IO ()
handleTwoOptionals (Just (a, b)) = ... work with 'a' and 'b'
handleTwoOptionals Nothing = ... proceed without values
```

With this slight change we made it impossible to specify only a single value as `Nothing`. If you pass `Just` of something, you must always provide both values.

Let's move on to another example. Now we want to write a function that takes two lists, but those lists must have the same length. Our function doesn't work when lists have different sizes. The type signature can look like this:

```haskell
processTwoLists :: [a] -> [b] -> Int
```

It is up to the caller to verify that both lists have the same size. But if you don't verify list lengths, the `processTwoLists` function fails. Moreover, even if a caller checked this property, the type signature still doesn't capture the verification result. Note, that it could benefit from the usage of the evidence pattern to bear in mind this fact.

Again, it is pretty easy to fix the problem by changing the type of `processTwoLists`:

```haskell
processTwoLists :: [(a, b)] -> Int
```

Instead of passing two lists and expecting them to have the same size, the function simply takes a list of pairs, so it has the same number of `a` s and `b` s.

And one more example. Imagine, that you are writing a web application with the backend and frontend. And you have a function that deals with the settings for both backend and frontend to launch your app. Both setting configurations can be optional, but at least one of the settings parts should be specified.

You can start approaching this problem by writing the following code:

```haskell
data Settings = Settings
    { settingsBackend  :: Maybe BackendSettings
    , settingsFrontend :: Maybe FrontendSettings
    }

runApp :: Settings -> IO ()
runApp Settings{..} = case (settingsBackend, settingsFrontend) of
    (Just back, Just front) -> configureBack back >> configureFront front >> run
    (Just back, Nothing)    -> configureBack back >> run
    (Nothing, Just front)   -> configureFront front >> run
    (Nothing, Nothing)      -> throw "You must specify at least one settings"
```

But the above function has the same problem: the data type makes it possible to specify values that shouldn't happen in real life. To fix this problem, we need to change the shape our `Settings` data type in the following way by using sum types:

```haskell
data Settings
    = OnlyBackend BackendSettings
    | OnlyFrontend FrontendSettings
    | BothSettings BackendSettings FrontendSettings

runApp :: Settings -> IO ()
runApp = \case
    OnlyBackend back        -> configureBack back >> run
    OnlyFrontend front      -> configureFront front >> run
    BothSettings back front -> configureBack back >> configureFront front >> run
```

Now, even developers unfamiliar with the codebase won't be able to create invalid settings. The shape of our data precisely describes all valid states in our program.

Generally, by pushing requirements for our data types upstream, we can implement more correct code and make the life of our API users easier, because they can't shoot themselves in the foot by providing illegal values.

However, it is not always possible to easily make all invalid states unrepresentable. Consider the following example. You have an enumeration type representing answers to some questions in a form. And users of this form must enter up to 3 different values in that form. Even if requirements explicitly tell that we must have one, two or three distinct answers, you might go with the a simple data type like this:

```haskell
data Answers = Answers
    { answers1 :: Answer
    , answers2 :: Maybe Answer
    , answers3 :: Maybe Answer
    }
```

Sure, this data type allows many illegal states, and there is a room for improvement. But to be exactly precise in our data description, we probably need to create another enumeration type specifying only valid combinations of `Answer` s. However, such enumeration will be huge and unmaintainable. Even if your original `Answer` type has 8 values, the resulting combinations type will have approximately one hundred constructions that you will need to tackle all everytime the original data type changes. In this case, it is not feasible to describe only valid states with a data type, so we don't bother doing this.

Make illegal states unrepresentable: Task 1

Implement the following function by applying the *make illegal states unrepresentable* pattern.

```haskell
-- group sublists of equal elements
-- >>> group "Mississippi"
-- ["M","i","ss","i","ss","i","pp","i"]
group :: Eq a => [a] -> [[a]]
```

Hint: Use the `NonEmpty` list.

Show solution

Make illegal states unrepresentable: Task 2

Improve the following code by applying the *make illegal states unrepresentable* pattern.

```haskell
-- Find sum of up to the first 3 elements
sumUpToThree :: Num a => [a] -> a
sumUpToThree []        = error "Empty list"
sumUpToThree [x]       = x
sumUpToThree [x, y]    = x + y
sumUpToThree [x, y, z] = x + y + z
sumUpToThree _         = error "More than three values"
```

Show solution

Phantom type parameters

Pattern	Phantom type parameters
Description	Additional type-level information available during compile-time by introducing extra type variables.
When to use	1. To avoid duplication of many similar data types. 2. To increase code type-safety.
Benefits	1. More compile-time guarantees. 2. Better ergonomics compared to code duplication. 3. Flexibility. 4. Extensibility.
Costs	1. Need to know when not to use it. Can get out of control when applied to unsuitable situations. 2. Less beginner-friendly.

Sometimes you can improve benefits gained with the usage of `newtype` s even further by using "phantom type variables" — type variables that are not used after the "=" sign in the data type declaration. You can exploit them to distinguish usages of the same `newtype` for different purposes. Instead of writing:

```haskell
newtype Id = Id
    { unId :: Int
    }

isCommentByUser :: Id -> Id -> Bool
```

or more verbose

```haskell
newtype UserId    = UserId    Int
newtype CommentId = CommentId Int

isCommentByUser :: UserId -> CommentId -> Bool
```

you can have one newtype to represent that all:

```haskell
newtype Id a = Id
    { unId :: Int
    }

isCommentByUser :: Id User -> Id Comment -> Bool
```

This approach allows you to avoid creating multiple data types that have the same purpose and behaviour.

For example, you can have a lot of instances for your `newtype` s and don't want to repeat them for `UserId`, `AdminId`, `CommentId`, `MessageId`, etc. but still desire to have more type-safety. Adding a phantom type parameter to a single data type is a minor yet powerful change to your data types that imposes an extra layer of type-safety to your code.

Going back again to our favourite example with passwords: you can have multiple entities that can log into your application (users, administrators, third-parties, etc.). And those different entities may have diverse authentication methods or password validation algorithms. You could use phantom type variables here as well in order to track sign-in information on the type-level and bind the password with its hash using the type-level tag:

```haskell
newtype Password a = Password
    { unPassword :: ByteString
    }

newtype PasswordHash a = PasswordHash
    { unPasswordHash :: ByteString
    }

-- 'PasswordHash' is associated with the same entity as 'Password'
mkPasswordHash :: Password a -> Maybe (PasswordHash a)
```

Now it is no longer possible to validate the password of a user with the hash of an admin.

📚 **Exercise:** The `mkPasswordhash` function takes a password and maybe returns password hash. Can you notice which one of the previously discussed patterns is used here? 😉

Phantom type parameters: Task

Improve the following code by applying the *Phantom type parameters* pattern. Don't worry about function implementations, they are not important in this task.

```haskell
import Data.Binary (Binary)
import Data.ByteString (ByteString)


newtype PrivateKey = PrivateKey ByteString
newtype PublicKey  = PublicKey  ByteString
newtype Signature  = Signature  ByteString

-- | Derive public key from secret key.
createPublicKey :: PrivateKey -> PublicKey
createPublicKey = error "Not implemented"

-- | Sign the data using the given 'PrivateKey'.
sign :: Binary a => PrivateKey -> a -> Signature
sign = error "Not implemented"

-- | Check that the signature is produced by the 'PublicKey', derived for the
-- corresponding 'PrivateKey' that signed the same type of data
verifySignature :: Binary a => PublicKey -> Signature -> a -> Bool
verifySignature = error "Not implemented"
```

Show solution

MonadFail sugar

Pattern	MonadFail sugar
Description	Elegant syntax for pattern-matching on nested or multiple different parts of the data.
When to use	1. When doing pattern-matching a lot and when a particular failure reason is not important.

Benefits	1. Clean syntax, less noise. 2. No runtime errors by avoiding partial functions.
Costs	1. No detailed error messages about failures. 2. No immediate understanding of what code would do in case of the error to pattern-match.

When time comes to extract deeply-nested fields of a nested data structure, or to perform multiple validations in a single block, you start thinking on the neat way to do it at the best cost. You obviously fancy to avoid using partial functions because they can sometimes hide unhandled cases and fail at runtime unexpectedly.

There is a nice trick that combines the `Maybe` data type, `do` -notation and `MonadFail` typeclass that allows writing such code. Each piece of this spicy combination will be explained further, but first we need to understand the use-case. Let's explore the following example.

When you write a function like this:

```haskell
isThisTheAnswer :: Int -> Bool
isThisTheAnswer 42 = True
```

and if you pass a number like `37` to this function, it will surely fail.

However, when you pattern-match on any value on the left side of `<-` inside `do` -block, a slightly different set of rules is used for handling non-covered patterns. These rules involve usage of the `MonadFail` typeclass.

The `MonadFail` definition has nothing exclusive:

```haskell
class Monad m => MonadFail m where
    fail :: String -> m a
```

You can use this typeclass directly by calling the `fail` method. But most of the time you are using this typeclass implicitly, usually with the `IO`. It is important to mention here that the `fail` method of the `MonadFail` typeclass participates in the desugarisation of `do` blocks.

In other words, the following sweet code

```haskell
main :: IO ()
main = do
    [_, arg2] <- getArgs
    print arg2
```

will be desugared into something like this:

```haskell
main :: IO ()
main = getArgs >>= \args -> case args of
    [_, arg2] -> print arg2
    _ -> fail "Some compiler-generated message"
```

The `MonadFail` instance for `IO` throws an IO exception, but the instance for `Maybe` is rather curious:

```haskell
instance MonadFail Maybe where
    fail _ = Nothing  -- ignore the string input and always return Nothing
```

Now let's have a closer look at a more specific example to understand better how `Maybe` and `MonadFail` play together. Let's say, we want to perform the following check on our data:

- extract the last line of the text from some buffer (therefore it should contain at least one line)
- make sure that it contains exactly 3 words
- the first word must be `"CMD"`
- the second word is the number 42

In case all the above is true, we want to return the last word in this sequence of words. Such verification can be easily written using the `Maybe` type and its `MonadFail` instance:

```haskell
bufferLastLine :: Buffer -> Maybe String

cmdSequence :: Buffer -> Maybe String
cmdSequence buffer = do
    line <- bufferLastLine buffer
    ["CMD", number, cmd] <- Just $ words line
    42 <- readMaybe number
    pure cmd
```

You can see that the code is clean, it does exactly what we expect with a minimal syntactic overhead, and it is also safe as a bonus.

📚 **Exercise:** Desugar the above code manually.

In some cases, when performing multiple checks, it is crucial to know which one failed first in order to run the corresponding action next or provide a better error message for users. In that case, you should use `Either` instead of `Maybe`. Or if all your verification checks are independent of each other, you can use the `Validation` data type to output **all** errors that were fired along the way. This is useful to show the informative error messages to users. The approach with `Either` will stop on the first error, when the `Validation` approach will perform all checks anyways.

However, when using more detailed error-reporting, you won't be able to make use of this `MonadFail` trick, because both `Either` and `Validation` don't implement the `MonadFail` instance, so the manual pattern-matching or other type of handling is required.

Interestingly, unlike `Either`-like data types, the `MonadFail` instance for lists in Haskell has a similar power to `Maybe`'s. If you have a container of a complicated data structure and your goal is to filter it only by some structure-specific criteria, keeping only interesting elements, you can use list comprehension and `MonadFail` for list. List comprehension is a cute syntax sugar for constructing lists. `do`-notation is a syntax sugar for `>>=` from `Monad`, so you can think about list comprehension as a syntax sugar for `do`-notation specifically for lists.

👩‍🔬 Haskell also has the MonadComprehensions extension that allows using the list comprehension syntax for other monads.

Let's say, we get a list of pairs with some `Either` values as both elements of those pairs, and we want to extract only `Int`s that are inside both `Right` and are the same. The description is wordy, but the code for this task with the usage of list comprehension is much more elegant:

```haskell
keepOnlySameRights :: [(Either e1 Int, Either e2 Int)] -> [Int]
keepOnlySameRights xs = [n | (Right n, Right m) <- xs, n == m]
```

If you attempt to implement such a function by using manual pattern-matching, it would look a bit less cleaner.

📚 **Exercise:** Try implementing the `keepOnlySameRights` functions without list comprehensions and the `MonadFail` instance for list.

MonadFail sugar: Task 1

Implement the following functions applying the *MonadFail sugar* pattern.

```haskell
-- returns sum of exactly 3 numbers separated by spaces in a string
-- >>> sumThree "10 20 15"
-- Just 45
-- >>> sumThree "10 7"
-- Nothing
sumThree :: String -> Maybe Int
```

Show solution

MonadFail sugar: Task 2

Implement the following functions applying the *MonadFail sugar* pattern.

```haskell
-- return only values inside Just
catMaybes :: [Maybe a] -> [a]
```

Show solution

MonadFail sugar: Task 3

Implement the following functions applying the *MonadFail sugar* pattern.

```haskell
-- return 'b's only from 'Just'
mapMaybe :: (a -> Maybe b) -> [a] -> [b]
```

Show solution

MonadFail sugar: Task 4

Implement the following functions applying the *MonadFail sugar* pattern.

```haskell
-- return @Just ()@ only if all three arguments are Nothing
threeNothing :: Maybe a -> Maybe b -> Maybe c -> Maybe ()
```

Show solution

Polymorphisation

Pattern	Polymorphisation
Description	Assigning a more general type to a function reduces the chances of writing an incorrect implementation or providing incorrect inputs.
When to use	1. To reduce risks of using function incorrectly. 2. To use the same function on values of different types. 3. To make illegal states unrepresentable.
Benefits	1. Need to write fewer tests (or no tests at all!). 2. Increases code reusability.
Costs	1. Can make type signatures look more complicated. 2. Reusing polymorphic types in `where` requires enabling the `ScopedTypeVariables` extension, and this might be confusing for beginners.

Haskell has the Polymorphism feature and allows writing and using polymorphic data types and functions. A polymorphic function can be defined only once for some general types, and then it can be called with arguments of different types, satisfying the type signature.

For example, the following function

```haskell
firstArg :: a -> b -> a
firstArg x _ = x
```

can have types, depending on arguments, either `firstArgs :: Int -> String -> Int` or `firstArg :: Maybe s -> [a] -> Maybe s` but not `firstArg :: Int -> Double -> Double`.

If a monomorphic function has type `Int -> Int` then it can do many different things and you can't really guess its behaviour by only looking at its type. You can guess by the function name, but naming is hard and you still need to check the documentation and probably even source code.

However, if the function's type is general as `a -> a` then this function can do only one thing and in that case, its name doesn't matter, you already know what it does.

> 📚 **Exercise:** Do you see what the function with the type `a -> a` does?

The thought that a more general function is less powerful is counter-intuitive. If it can work with more types, how can it be less powerful?

But the more you think about this, the more sense it starts making. The more polymorphic the type is, the less information we know about arguments, the more common types satisfying constraints should be, and the fewer things you can do with them. You can't create values of some unknown type `a` out of nowhere because you don't know their constructor methods. On the other hand, you can do a lot of things with some specific type like `Int`: increment, decrement, divide, square, return always 0, return last digit, etc.

This fact can be used to implement safer interfaces. The following elaborate example demonstrates key features of the polymorphisation approach.

Let's say we want to implement a function that compares two pairs of `Int`s and something else by the first element of a pair:

```haskell
compareByFst
    :: (Int -> Int -> Ordering)
    -> (Int, Bool)
    -> (Int, Bool)
    -> Ordering
```

This type is fine, however, it's possible to implement not exactly what we wanted but still satisfy this type signature. For example:

```haskell
compareByFst cmp (a, _) (b, _) = cmp (a + 1) (b + 1)
```

> 📚 **Exercise:** Can you see when this function can produce wrong results?

We want to restrict what the function can do. So instead of using `Int` we can use some polymorphic variable:

```haskell
compareByFst :: (a -> a -> Ordering) -> (a, x) -> (a, x) -> Ordering
```

Though, since the function produces `Ordering`, you still can simply ignore all arguments and return `EQ` in all cases. But we can notice, that there's no need to stick to the `Ordering` as well:

```haskell
compareByFst :: (a -> a -> c) -> (a, x) -> (a, x) -> c
```

However, this function still has a problem: it is possible to apply arguments in the wrong order. The following two implementations are valid:

```haskell
compareByFst cmp (a, _) (b, _) = cmp a b
compareByFst cmp (a, _) (b, _) = cmp b a
```

When you are working only on this function, you may notice such logic errors during the development. However, programmers are humans. They can be sleepy or they can do some typos which can totally happen when you change a lot of code. And moreover, it is quite troublesome to debug them.

So let's do the final step and specify the most general form of this function, renaming it as well since it has nothing to do with `compare` anymore:

```haskell
applyToFst :: (a -> b -> c) -> (a, x) -> (b, y) -> c
applyToFst f (a, _) (b, _) = f a b
```

Now, there is only one way to implement `applyToFst`! If you write a wrong implementation, the compiler will tell you, and such bugs won't be even introduced in the first place before it goes to production. Furthermore, since the function is polymorphic, you don't need to change the code that uses this function. We can easily restore our first type signature from a polymorphic one by equating specific types with monomorphic variables:

```haskell
a ~ Int
b ~ Int
c ~ Ordering
x ~ Bool
y ~ Bool
```

Moreover, the described approach can be applied to already polymorphic functions, which are not polymorphic enough. Consider the following example of a general-purpose function that can be improved by the Polymorphisation technique:

```haskell
partition :: (a -> Bool) -> [a] -> ([a], [a])
```

The intention of this function is to separate elements of the list into two lists by predicate. Unfortunately, this type signature has several problems:

1. It's not clear, in which part of the tuple go elements that satisfy the predicate.
2. In the implementation of this function you still can add elements to a wrong list.

A better type signature would be:

```haskell
partitionWith :: (a -> Either b c) -> [a] -> ([b], [c])
```

Now it is much harder to implement this function in the wrong way and its type signature also tells us much more about its behaviour!

📚 **Exercise:** Implement `partitionWith`.

👩‍🔬 It is still possible to implement `partition` in a wrong way by not adding elements to the result list. However, LinearTypes can help with ensuring this invariant.

This trick is especially good with the MonadFail sugar pattern. Instead of filtering elements by predicate and using unsafe conversion functions, you can use `mapMaybe` and make illegal states unrepresentable.

Look at the following function:

```haskell
-- >>> sumEven "100 15 20 7"
-- 60
sumEven :: String -> Int
sumEven =
    sum
    . map (`div` 2)
    . filter even
    . map read
    . filter isNumber
    . words
```

It can be written in a slightly different way, that also allows extending filtering rules easily:

```haskell
sumEven :: String -> Int
sumEven = sum . mapMaybe parseNumber . words
  where
    parseNumber :: String -> Maybe Int
    parseNumber str = do  -- do-notation for Maybe monad
        num <- readMaybe str
        guard $ even num
        pure $ num `div` 2
```

Polymorphisation: Task 1

Improve the following functions by applying the **Polymorphisation** pattern.

```haskell
-- append all elements inside all Just values
-- >>> maybeConcat [Just [1,2,3], Nothing, Just [4,5]]
-- [1,2,3,4,5]
maybeConcat :: [Maybe [Int]] -> [Int]
```

Show solution

Polymorphisation: Task 2

```haskell
-- return lists containing a given integer
-- >>> containsInt 3 [[1..5], [2,0], [3,4]]
-- [[1,2,3,4,5], [3,4]]
containsInt :: Int -> [[Int]] -> [[Int]]
```

Show solution

Polymorphisation: Task 3

```haskell
-- split list in two parts stopping when predicate returns false
-- >>> span (< 3) [1, 2, 4, 2]
-- ([1, 2], [4, 2])
span :: (a -> Bool) -> [a] -> ([a], [a])
```

Show solution

Bidirectional parsing

Pattern	Bidirectional parsing
Description	Matching only a limited set with exhaustiveness checking and inversing matching function automatically.
When to use	1. When you need to show enumerations and parse them back. 2. When the `show` function is injective (maps distinct constructors to distinct values). 3. Basically, for any bidirectional conversion by implementing only one direction and getting the inverse conversion for free.
Benefits	1. Automatic code correctness by implementation. 2. Easier maintainability.
Costs	1. Extra dependencies or extra code. 2. Less manual control over code.

2. Less manual control over code.

It is often in programs, that you use enumeration types (data types with several constructors without any fields). Enums are classy at least because when we pattern-match on them, GHC yells at us about cases we forgot. But things become a bit more complicated when you also want to be able to parse your enums from strings, which is a completely sane and frequent requirement. This is problematic because when handling strings via pattern-matching, you are physically unable to handle all cases because there is an infinite number of them.

Consider the following example:

```haskell
data Colour
    = Green
    | Yellow
    | Blue

showColour :: Colour -> Text
showColour = \case
    Green  -> "green"
    Yellow -> "yellow"
    Blue   -> "blue"

parseColour :: Text -> Maybe Colour
parseColour = \case
    "green"  -> Just Green
    "yellow" -> Just Yellow
    "blue"   -> Just Blue
    _        -> Nothing
```

It’s important that you pattern match on all cases of enumeration when possible to avoid having partial functions.

If you add a new colour constructor, for example `Orange`, the compiler will warn you to update the `showColour` function. But, unfortunately, it won’t remind you to update `parseColour` as the `"orange"` string is already covered by the wildcard (`_`) case. Therefore it is all good from the compiler point of view.

If you have only a few data types with a few constructors and you don’t update them, then you don’t have such a problem. Otherwise it is not straightforward to maintain such code.

Fortunately, there is a way to avoid this issue. You can implement helper functions that will parse your enumeration types from text back to safe Haskell data types. One of such functions is `inverseMap` from relude:

```haskell
inverseMap
    :: forall e s . (Bounded e, Enum e, Ord s)
    => (e -> s)
    -> s
    -> Maybe e
```

📚 **Exercise:** Implement the `inverseMap` function.

Using this function, you can rewrite `parseColour` in a simpler and more safe way:

```haskell
parseColour :: Text -> Maybe Colour
parseColour = inverseMap showColour
```

Even if you are not using the `showColour` function, this approach still can be a better solution, since you can handle cases exhaustively and get a parsing function for free.

You can find more details about this approach in the Haddock documentation for the `inverseMap` function.

Bidirectional parsing: Task

Implement a function that will take a string of two space-separate words — colour and fruit name — and return them as data types.

Possible colours: red, green, yellow, blue. Possible fruits: apple, orange, lemon, blueberry.

Example:

```
ghci> parseFruit "red apple"
Just
    ( Fruit
        { fruitColour = Red
        , fruitName = Apple
        }
    )
```

Show solution

Recursive go

Pattern	Recursive `go`
Description	Moving recursion over data types into the separate function.
When to use	1. When recursion reuses some arguments and you want to avoid passing them again. 2. When recursion uses some internal state. 3. To avoid revalidating the same data.
Benefits	1. Cleaner code. 2. Possible performance improvements.
Costs	1. More code. 2. Requires to know about and enable the `ScopedTypeVariables` extension.

🙅 **Disclaimer** This pattern is not about recursive functions in the Go programming language.

Quite often Haskell developers end-up writing functions that recursively do some actions on different data types: lists, trees, numeric accumulators, etc. A function that returns the element of the list at the given position (if found) can be considered as the example of such function. And it could be written using pattern matching.

It is important that such a function has a stopping condition, and a step to recursively iterate through the list. In our case the stop signals are:

- Either you found the element on the given position
- Or you traversed the whole list but didn't find what you need

The step for this function is to examine the tail of the list (throwing away the first element) and decreasing the position number by one (for each thrown element).

Haskell implementation may look like this:

```
at :: [a] -> Int -> Maybe a
[] `at` _ = Nothing
(x:_) `at` 0 = Just x
(_:xs) `at` i = xs `at` (i - 1)
```

However, there is another way to write the above function. Here we are going to use the recursive `go` pattern – the pattern when instead of using the function itself for the recursion, you create the internal so-called `go` function that will do that.

The idea here is to run the `go` function with the empty "accumulator" and the starting list, and the `go` function should recursively update the accumulator and walk the whole list. This accumulator would be compared to the input position (in contrast to the comparison to 0 when you decrease the original value in the previous case).

You can code this idea this way:

```haskell
atGo :: [a] -> Int -> Maybe a
l `atGo` n = go 0 l
  where
    go :: Int -> [a] -> Maybe a
    go _ [] = Nothing
    go i (x:xs) = if i == n then Just x else go (i + 1) xs
```

The way the `go` pattern is implemented also is more flexible in the refactoring and "requirements"-change. If you, for example, need to check the `at` function index input on the negative number, it would be much easier and much more efficient to add this validation before the `go` function of `atGo`. You don't need to check the number of negativity with each recursive call. You can do this only once, and the recursive `go` function will work only with valid data.

📚 **Exercise:** Can you make these `at` / `atGo` functions more efficient on the corner cases?

This pattern could be especially handy when you need to do additional actions on the input data after you get the result of the recursive calculations. Or if you want to avoid passing arguments that never change during the recursive traverse of your data structure (e.g. `map`, `filter`, `foldr`, etc.).

Recursive go: Task 1

Improve the following code by applying the *Recursive `go`* pattern.

```haskell
sum :: Num a => [a] -> a
sum [] = 0
sum (x:xs) = x + sum xs
```

Show solution

Recursive go: Task 2

Write the `ordNub` function with the *Recursive `go`* pattern. The function should return all unique elements from a list in the order of their appearance.

```haskell
ordNub :: Ord a => [a] -> [a]
```

```
>>> ordNub [3, 3, 3, 2, 2, -1, 1]
[3,2,-1,1]
```

Show solution

Conclusion

In this blog post we covered some small techniques that increase code maintainability, readability and reusability. You can see how different Haskell features come together. Each pattern is a small improvement, but when used properly, they increase code quality significantly. Try to use them in your next Haskell project and let us know if they help! We hope that the described patterns will help you write better code.