

BABEȘ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER
SCIENCE
SPECIALIZATION COMPUTER SCIENCE

DIPLOMA THESIS

Architecture of Functional Programming Languages

Supervisor:

Assoc. Prof. Ph.D Crăciun Florin

Author:

Criste Denis Lorin

2021

UNIVERSITATEA BABEȘ-BOLYAI CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ ENGLEZĂ

LUCRARE DE LICENȚĂ

Arhitectura Limbajelor de Programare Funcționale

Conducător științific:

Conf. Dr. Crăciun Florin

Absolvent:

Criste Denis Lorin

2021

Abstract

Nowadays, functional programming is becoming more and more popular in the world of computer science. Many programming languages adopted the functional programming paradigm, including Java with its streams or Python with its lambda expressions. Functional Programming aims to build software by composing functions and avoiding shared state, mutable data and side effects.

This thesis focuses on the architecture of functional programming languages. It aims to offer a glimpse into the functional paradigm by presenting the main bricks that are used in the process of building a functional software application. The support application for this thesis provides a solution to the management of seminar's attendances and activities.

The motivation behind the practical application was born in the last academic year when the global pandemic forced our college to start its activity online. One of the problems that came with online teaching was the management of the seminars, which was usually done on private excel files held on the teacher's laptop. This practical application aims to solve this problem by creating a web application that will allow students to see their seminar data and teachers to create new attendances or activities for a given seminar.

This thesis is composed of 4 chapters, two of them theoretical, one practical and one for conclusions. The first theoretical chapter is covering the basics of functional programming. This includes types, type-checking, functions, laziness, partial application, type classes and some simple examples used as support for the explanations.

The second theoretical chapter looks into the world of effects, presenting how a functional language like Haskell produces and interacts with effects. This chapter also introduces more advanced topics like monads, software transactional memory or solutions to the onion architecture.

The practical chapter is presenting the main libraries used in the application together with pictures of the source code that will be explained in the process. The practical chapter also presents the life-cycle of the application and its functionalities.

This work is the result of my own activity. I have neither given nor received unauthorized assistance on this work.



Criste Denis Lorin

Contents

Introduction	6
Chapter 1. Haskell	8
1.1. Introduction	8
1.2. Types	8
1.3. Functions	11
1.4. Type Classes	14
1.5. Lambda Cube	16
1.6. Purescript	19
Chapter 2. Effects	20
2.1. Introduction	20
2.2. What is an effect?	20
2.3. What's a Haskell monad?	21
2.4. STM	24
2.5. MTL	26
2.6. DSLs	27
Chapter 3. Practical application	30
3.1. Introduction	30
3.1.1. Requirements and Specifications	30
3.1.2. Design	31
3.1.3. Testing	33
3.2. Servant	33
3.2.1. Servant Introduction	33
3.2.2. API types	33
3.2.3. STM	39
3.3. Database	40
3.4. Halogen	41
3.4.1. Halogen Architecture	41
3.4.2. Material Components Design	44
3.5. Microsoft Teams	45
3.5.1. Introduction	45
3.5.2. Tabs	46
3.5.3. Bots	47

3.5. Testing	49
Chapter 4. Conclusions	51
Bibliography	52

List of figures

Figure 1.1 Haskell a strongly typed language	8
Figure 1.2 JavaScript a weakly typed language	9
Figure 1.3. Creating a simple data type in Haskell	10
Figure 1.4 Currying and Partial application	11
Figure 1.5 Partial application in Haskell	12
Figure 1.6 Builder Pattern in Java	13
Figure 1.7. Haskell, Eq type class	14
Figure 1.8. Eq Type Class implementation	15
Figure 1.9 Simply typed lambda calculus syntax	16
Figure 1.10 Lambda cube	17
Figure 1.11 System F syntax	18
Figure 1.12 System Fw syntax	19
Figure 2.1. IO function in Haskell	21
Figure 2.2 Maybe monad, and >>=	22
Figure 2.3 Maybe monad type class	23
Figure 2.4 Atomically in Haskell	24
Figure 2.5 Optimistic locking	25
Figure 2.6 MaybeT monad instance Haskell	27
Figure 2.7 Example of DSL together with its interpreters	28
Figure 3.1 Use case diagram of the application	31
Figure 3.2 Communication diagram of the application	32
Figure 3.3 API of the practical application	34
Figure 3.4 From Handlers to Application, function app from practical application	36
Figure 3.5 ModelAPI type-class	37
Figure 3.6 ModelAPI instance	38
Figure 3.7 Server handlers	38
Figure 3.8 STM in practice, Microsoft Teams connection handler	39
Figure 3.9 Servant application with file-based Database	40
Figure 3.10 File-based database ModelAPI instance	41
Figure 3.11 Root component HTML code insert in the body tag	42
Figure 3.12 Root component in the practical application	43
Figure 3.13 JavaScript function that initializes all the tags with a certain class name	44
Figure 3.14. Entry point of the application initialized as a Microsoft Teams application	45
Figure 3.15. Microsoft Teams tab	46
Figure 3.16. Microsoft Teams bot	47
Figure 3.17. Microsoft Teams bot login and messaging	48
Figure 3.18. Testing Servant endpoints first part	49
Figure 3.19. Testing Servant endpoints second part	50

Introduction

Code inflation is a well-known problem in the industry of software development. The software applications tend to grow in complexity until they become hardly maintainable. This implies larger teams of programmers and specialized teams of testers. For the first time in history, the complexity of our software products overtakes human nature's ability to reasoning and understanding.

The more a programmer is responsible for, the larger is the probability of an error. This is the reason why in the last years, testing has become an essential skill for every programmer. Many different types of testing were developed (unit, integration, system and acceptance testing) and designed to catch as early as possible the inevitable human error.

Design patterns came as a partial solution to the problem mentioned above, and in no time, more than two dozen patterns imposed rules and restrictions in the world of computer science. Code reusability, single responsibility and the support for extension are only a few rules that every programmer have in mind before starting to write code.

Functional programming presents itself as another solution to the code inflation problem. The idea of having functions that are very close to the mathematical ones brings the main advantage of having reliable code. Working with functions that have the same output for the same input lets us reason more easily about the program correctness and even allows us to build proofs for the correctness of our software.

The programming language chosen for this thesis is Haskell, and the motivation is the power of its type system. In Haskell, not only the functions but also the effects are encoded into the type system. This allows the compiler to carry part of the burden of reasoning about the program correctness.

The theoretical chapters from this thesis make progressive steps into the world of functional programming using a lot of code examples to explain basic concepts at first and more advanced topics later. The theoretical part aims to present the main bricks that compose the architecture of a functional software application and also to give a couple of models of how functional software architecture can be built.

Regarding the practical application, it aims to facilitate the management of the seminar attendances and activities. However, there exist similar applications that help teachers with seminar management, like Fedena[1] or Creatix Campus[2].

They allow a teacher to assign attendances for different students and even specify if the student was late or if he left earlier. Attendances can be added if the teacher just clicks a date from a list of dates displayed next to each student or selects a date from a calendar input type and fills a checkbox corresponding to the attendance state: Present, Absent, Late, Leave.

My personal contributions to the problem of seminar management do not consist in extra functionality but rather in the approach taken in order to solve the problem.

The first step that separates this application from Fedena[1] or Creatix Campus[2] is the integration with Microsoft Teams. Students already used many platforms, each of them offering the functionality the other lacks. We used Zoom, Microsoft Teams, Slack, Canvas, Moodle, Impulse and Socrative. Our application is built to bring in one place another functionality rather than dealing with the seminar management problem in isolation.

The next important step is the fact that our application is built using only functional programming languages: Haskell for the backend part and Purescript for the frontend part. This will facilitate the future extension of this application, controlling better the problem of code inflation.

The last step is the type-driven development style used in the source code of the application, which is a programming style where the programmer firstly defines the type of the function and only after that, he implements it, having the function type as both documentation and requirements.

Chapter 1. Haskell

1.1. Introduction

This chapter consists of 5 sections, and it will cover the basics of functional programming. It will briefly present the Haskell type system (section 2), the foundations of a functional programming language (section 3), what are type-classes (section 4), the lambda cube (section 5) and finally, a sister language of Haskell, called Purescript (section 6).

1.2. Types

Haskell has a strong type system. Liskov defined a strongly-typed language in 1974 as one in which “whenever an object is passed from a calling function to a called function, its type must be compatible with the type declared in the called function” [3].

In figure 1.1, the function `add10` expects an argument of type `int`, and it receives a string instead, which leads to an error.

```
1 add10 :: Int -> Int
2 add10  x  = x + 10
3
4 main = do
5   print $ add10 "4"
```

```
$ghc -O2 --make *.hs -o main -threaded -rtsopts
[1 of 1] Compiling Main                ( main.hs, main.o )

main.hs:5:19: error:
• Couldn't match expected type 'Int' with actual type '[Char]'
• In the first argument of 'add10', namely '"4"'
  In the second argument of '($)', namely 'add10 "4"'
  In a stmt of a 'do' block: print $ add10 "4"
```

Figure 1.1 Haskell a strongly typed language

In figure 1.2, the same function, written in JavaScript (a weakly typed language), receives a string argument and still returns successfully.

		JavaScript	
1	<script>		14
2	function add10(x){		410
3	return x + 10;		
4	}		
5	document.write(add10(4));		
6	document.write(" ");		
7	document.write(add10("4"));		
8	</script>		

Figure 1.2 JavaScript a weakly typed language

During the type checking, the integer `10` was converted to a string, and the function returned a string concatenation. This conversion is called `type coercion` (the ability for a value to change its type implicitly in specific contexts), and it's not allowed in a strongly typed language.

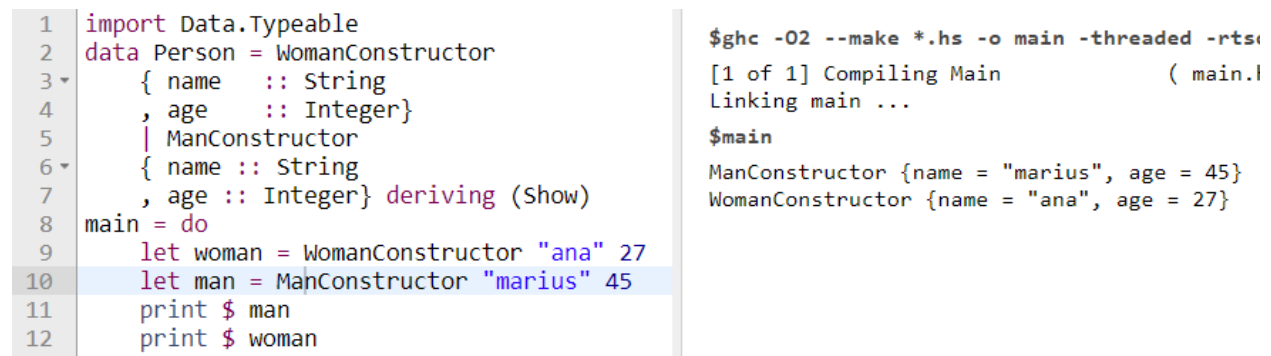
In a more general sense, a strongly typed language means that a language has no unsound types (it is always the case that the type of a variable can be trusted) in its type system, whereas weak typing means the type system can be unsound (for example, in the weak language C, you can easily cast a value of any type to another type of the same size).

The advantage of a strongly typed programming language is that it provides more information about functions (which are generally opaque) to the compiler (leading to more optimization opportunities) and the programmer, allowing the encoding of program properties at the level of types. These properties are verified for the whole program by the type checker.

However, this comes with a disadvantage: you have to be aware of the structure imposed by the chosen types on the application you are writing. This often amounts to the ceremony required to please the compiler, which may result in larger codebases and more verbosity. Type checking algorithms allow for the compiler to guess the types of variables that lack type annotations. This is called type inference, and in practice, most types can be inferred. The function `add10` from figure 1.1 can have its type deduced because of the expression `+ 10`, which says that the argument `x` and the result should have the same type as number 10.

Haskell's type system is static, which means the compiler needs to know exactly what types are being used at compile time. If the compiler can't infer a function type, the programmer has to declare it explicitly. The main advantage of a static type system is that one can catch errors at compile-time instead of catching them at runtime, which is much more expensive [4].

In Haskell, the types and values are similar to the classes and object-oriented programming objects. A data type can have zero or more data constructors, and each data constructor takes zero or more parameters to build a value.

The image shows a side-by-side comparison of Haskell source code and its compilation output. On the left, the source code is displayed with line numbers 1 through 12. It defines a data type 'Person' with two constructors, 'WomanConstructor' and 'ManConstructor', each taking a 'String' for 'name' and an 'Integer' for 'age'. It also includes a 'main' function that creates instances of these constructors and prints them. On the right, the output of the GHC compiler is shown, indicating successful compilation and linking, and displaying the resulting values for 'man' and 'woman' in a structured format.

```
1 import Data.Typeable
2 data Person = WomanConstructor
3   { name  :: String
4   , age   :: Integer}
5   | ManConstructor
6   { name  :: String
7   , age   :: Integer} deriving (Show)
8 main = do
9   let woman = WomanConstructor "ana" 27
10  let man = ManConstructor "maris" 45
11  print $ man
12  print $ woman
```

```
$ghc -O2 --make *.hs -o main -threaded -rtsopts
[1 of 1] Compiling Main             ( main.hs )
Linking main ...
$main
ManConstructor {name = "maris", age = 45}
WomanConstructor {name = "ana", age = 27}
```

Figure 1.3. Creating a simple data type in Haskell

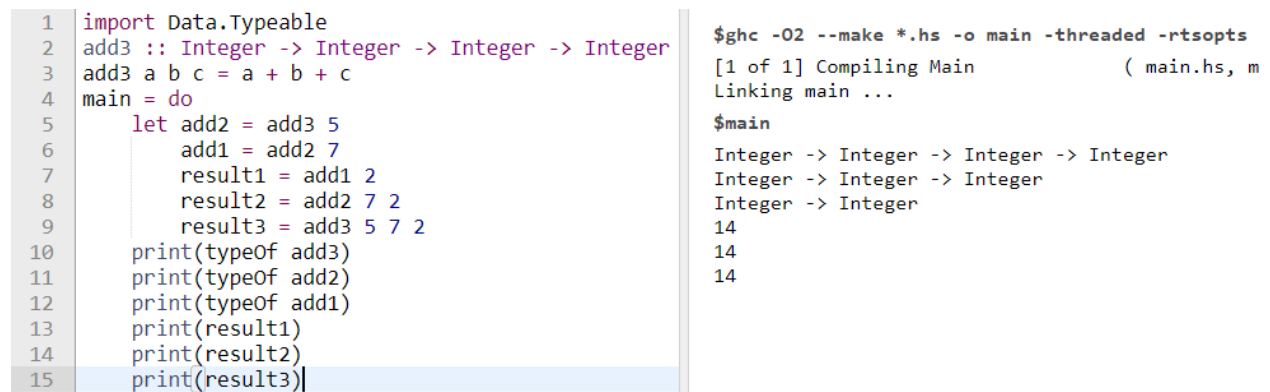
In figure 1.3, we define at line 2 a data type `Person` that has two data constructors. The ``data`` keyword signals that what follows is a type constructor, `Person`. The equal sign separates the type constructor from the data constructors, separated by the symbol ``|``. Each data constructor can take zero, one or more arguments (`WomanConstructor` and `ManConstructor` are taking both arguments: a name and an age).

A data type does not encapsulate methods, as opposed to an ordinary object, and the fields inside the data values can be accessed publicly. A data value can't be modified, but a new value can be created instead. This is similar to the way functions work in mathematics: they take an input and return an output without altering the input value. Creating a new data value whenever we need to modify a variable may seem problematic from an efficiency perspective. However, the immutability of Haskell allows the already existing attributes and values to be shared and reused freely [5]. Whenever a new value is created, only the modified fields will occupy a new space in memory.

1.3. Functions

In Haskell, all functions take only one parameter and return an output. Other programming languages (C++, Java) allow you to define functions that take multiple parameters, which is a feature that can be emulated by using tuples (or records) in Haskell.

However, in Haskell, functions may return another function. In practice, this allows the nesting of functions in a pattern called currying, which is functionally equivalent (isomorphic, witnessed by the `curry` and `uncurry` functions) to multiple parameter functions from other languages. [6].



```
1 import Data.Typeable
2 add3 :: Integer -> Integer -> Integer -> Integer
3 add3 a b c = a + b + c
4 main = do
5     let add2 = add3 5
6         add1 = add2 7
7         result1 = add1 2
8         result2 = add2 7 2
9         result3 = add3 5 7 2
10    print(typeOf add3)
11    print(typeOf add2)
12    print(typeOf add1)
13    print(result1)
14    print(result2)
15    print(result3)
```

```
$ghc -O2 --make *.hs -o main -threaded -rtsopts
[1 of 1] Compiling Main             ( main.hs, m
Linking main ...

$main
Integer -> Integer -> Integer -> Integer
Integer -> Integer -> Integer
Integer -> Integer
14
14
14
```

Figure 1.4 Currying and Partial application

Such an example is presented in figure 1.4, at line 2, where we define a function that receives three integers and returns their sum. In the function signature:

`Integer -> Integer -> Integer -> Integer`, the first three Integers are the types of the input parameters, and the last one is the type of the result.

The arrows which separate them are type constructors for functions. A type constructor acts similar to a data constructor, except, as the name implies, it is used to construct types rather than values. It receives two types as arguments, more exactly the input type and the output type of the function, and the result type is the type of the function.

The type constructor of functions is right-associative, meaning the signature of the function ``sum3`` can be rewritten in a way that emphasizes how currying works: `Integer->(Integer->(Integer-`

`>Integer`)). The function `add3` actually takes one `Integer` argument and returns a function, with the signature `Integer->(Integer -> Integer)`, as shown in figure 1.4, at lines 10-12.

Haskell is curried by default, but you can uncurry functions. Uncurrying means transforming a function into a derived function that takes a single argument: a tuple that contains all the arguments needed by the base function. The opposite function of `uncurry` is `curry`, which transforms a function that takes a tuple as an argument into nested functions that progressively takes all the arguments stored in that tuple.

The signature of `uncurry`: `uncurry :: (t1->t2->t) -> (t1,t2) -> t` together with the signature of `curry`: `curry :: ((t1,t2)->t) -> (t1 -> (t2 -> t))` easily tells us that the functions are isomorphic, resulting the following laws: `curry (uncurry f) = f` and `uncurry(curry f) = f`, where `f` is any function. [6]

<pre> 1 import Data.Typeable 2 data Person = PersonConstructor 3 { name :: String 4 , age :: Integer 5 , gender :: String } deriving Show 6 7 main = do 8 nameLine <- getLine 9 let name = init nameLine 10 namedPerson = PersonConstructor name 11 ageLine <- getLine 12 let ageString = init ageLine 13 age = read ageString :: Integer 14 namedAgedPerson = namedPerson age 15 genderLine <- getLine 16 let gender = init genderLine 17 completePerson = namedAgedPerson gender 18 print (typeof PersonConstructor) 19 print (typeof namedPerson) 20 print (typeof namedAgedPerson) 21 print (typeof completePerson) 22 print (completePerson) </pre>	<pre> \$ghc -O2 --make *.hs -o main -threaded -rtsopts [1 of 1] Compiling Main (main.hs, main.o) Linking main ... \$main [Char] -> Integer -> [Char] -> Person Integer -> [Char] -> Person [Char] -> Person Person PersonConstructor {name = "radu", age = 45, gender = "male"} </pre>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

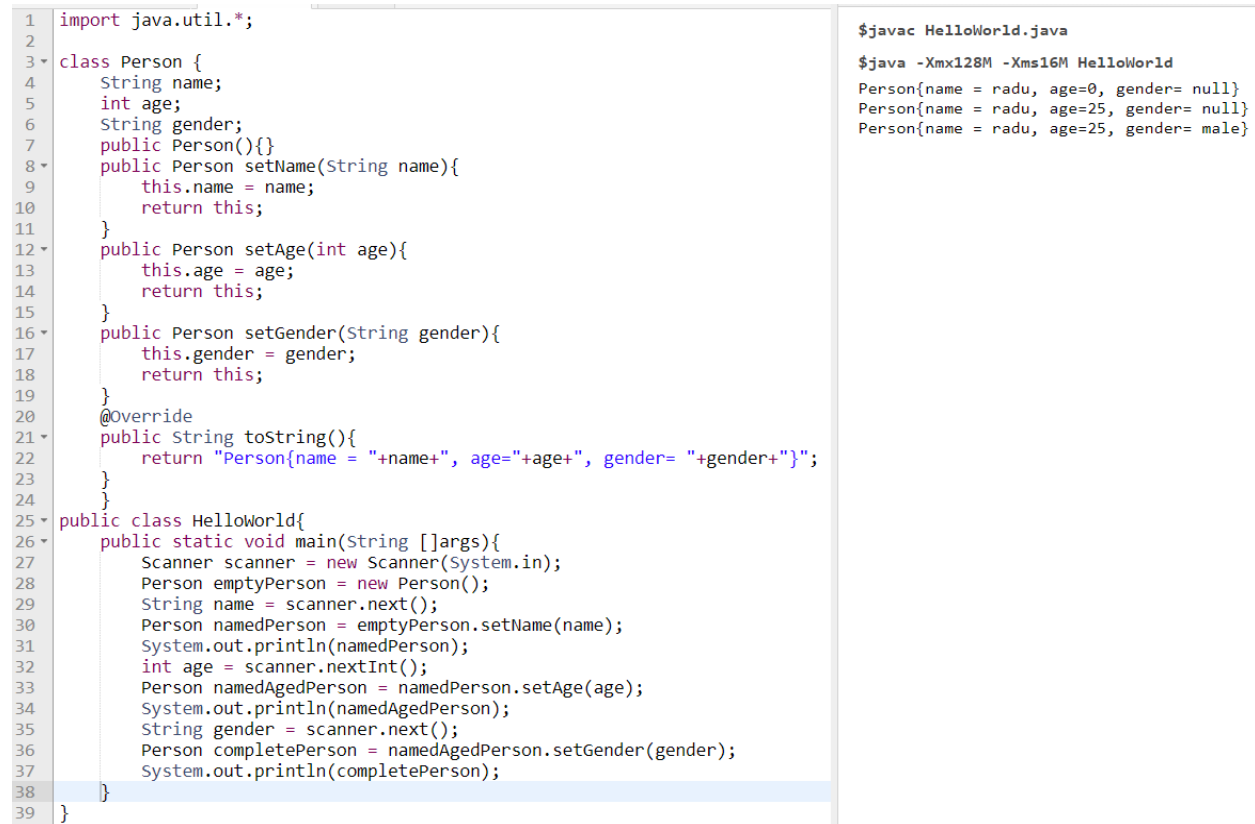
Figure 1.5 Partial application in Haskell

The ability to call a function that takes multiple parameters with only one parameter is called partial application. An advantage of the partial application is that it makes possible the evaluation/simplification of the functions [7]. Haskell also allows the partial application of the type constructors.

In Haskell, a data constructor acts like a function, meaning it can be partially applied. In figure 1.5, at line 2, we declare a `Person` data type that takes three parameters: `name`, `age` and `gender`. In

the main function (lines 10,14,17), the Person value is constructed in multiple steps, each applying, as soon as it is available, another parameter to the data constructor.

This step-by-step construction of a data type was necessary for imperative languages (Java, C++) that don't have support for partial application. The workaround is the builder design pattern, in which the construction of the object is done with setters, each setter returning an incomplete object that has default values for the unset attributes.



```
1 import java.util.*;
2
3 class Person {
4     String name;
5     int age;
6     String gender;
7     public Person(){}
8     public Person setName(String name){
9         this.name = name;
10        return this;
11    }
12    public Person setAge(int age){
13        this.age = age;
14        return this;
15    }
16    public Person setGender(String gender){
17        this.gender = gender;
18        return this;
19    }
20    @Override
21    public String toString(){
22        return "Person{name = "+name+", age="+age+", gender= "+gender+"}";
23    }
24 }
25 public class HelloWorld{
26     public static void main(String []args){
27         Scanner scanner = new Scanner(System.in);
28         Person emptyPerson = new Person();
29         String name = scanner.next();
30         Person namedPerson = emptyPerson.setName(name);
31         System.out.println(namedPerson);
32         int age = scanner.nextInt();
33         Person namedAgedPerson = namedPerson.setAge(age);
34         System.out.println(namedAgedPerson);
35         String gender = scanner.next();
36         Person completePerson = namedAgedPerson.setGender(gender);
37         System.out.println(completePerson);
38     }
39 }
```

```
$javac HelloWorld.java
$java -Xmx128M -Xms16M HelloWorld
Person{name = radu, age=0, gender= null}
Person{name = radu, age=25, gender= null}
Person{name = radu, age=25, gender= male}
```

Figure 1.6 Builder Pattern in Java

Figure 1.6 shows a very simple use of the builder design pattern, similar to the partial application of the data constructor from figure 1.5.

A pure function is similar to a mathematical function that always gives the same outputs for the same inputs: it receives input, makes some computations, and then returns. In imperative programming languages (Java, C++), a function can access global state and modify variables without reflecting these actions into the type of the function.

Pure functions, however, cannot do any of this, not even change a variable value. This is called referential transparency, and besides the fact that it allows the compiler to better reason about the program's behaviour, it also allows the programmer to know that a function does not have side effects. In Haskell, if your function uses side effects, you must declare them in the type-system. Otherwise, you will get a compile error.

The evaluation strategy is, by default, lazy. This means that a value or a function is evaluated only when it's needed. [8]

1.4. Type Classes

Let's take a closer look at the signature of the sort function in Haskell: `Ord a => [a] -> [a]`. What sets it apart from the signatures presented so far is the beginning of the signature, mainly `Ord a =>`. The rest of the signature shows that the function receives a list containing elements of type `a`, and it returns another list with elements of type `a`.

The symbol `=>` separates type constraints from the signature, so `Ord a` is seen here as a constraint, meaning that type `a` must implement all the functions from the `Ord` type-class (a class that defines an ordering). As interfaces are often explained in Java, a type class is a contract between some data types and the user. If a data type implements a type class, it can be used in any polymorphic function with a type variable constrained by the type-class.

```
1 class Eq a where
2   (==), (/=) :: a -> a -> Bool
3   x /= y = not (x == y)
4   x == y = not (x /= y)
```

Figure 1.7. Haskell, Eq type class

Figure 1.7 represents the definition of the equal type class in Haskell. In line 1, the keyword ``class`` defines a new type class named `Eq` which takes a type ``a`` as an argument.

In the body of the class definition, two methods are defined `==` and `/=`, and at least one of them must be explicitly implemented by any data type that is an instance of the `Eq` class type.

<pre> 1 import Data.Typeable 2 data Person = PersonConstructor 3 { name :: String 4 , age :: Integer 5 , gender :: String } 6 instance Eq Person where 7 (PersonConstructor name1 age1 gender1) == 8 (PersonConstructor name2 age2 gender2) = 9 ((name1 == name2) && 10 (age1 == age2) && 11 (gender1 == gender2)) 12 13 main = do 14 let alexPerson = PersonConstructor "Alex" 25 15 femaleAlex = alexPerson "female" 16 maleAlex = alexPerson "male" 17 sameMaleAlex = PersonConstructor "Alex" 25 "male" 18 print (femaleAlex == maleAlex) 19 print (maleAlex == sameMaleAlex) </pre>	<pre> \$ghc -O2 --make *.hs [1 of 1] Compiling Ma Linking main ... \$main False True </pre>
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------

Figure 1.8. Eq Type Class implementation

In figure 1.8, at line 6, the Person data type implements the function == of the Eq type class. Lines 7-10 specify that a Person is equal with another Person if they have the same name, age and gender.

Class types are very different from what classes mean in object-oriented programming. Their definition is closer to what interfaces are in Java. Class types can be used to restrict the types the polymorphic function arguments can take, letting the programmer use the functions defined in the type-class for any current or future instance of the type-class.

For example a simple function that sums up two numbers can have the following signature: `add2 :: (Num a) => a -> a -> a`. This means that the add2 function can be called with arguments of any data type that implements the Num type class.

Num type class defines function +, which is implemented by data types like Int, Integer, Float and Double. As long as the arguments provided have the same type, add2 can be successfully called with any values of those data types.

Type classes also let the compiler reason about the capabilities of a function. Let's look at a function with the following signature: `simple :: a -> a`. This function can only be the identity function that returns the input value. There is no other function that can manipulate the input without knowing anything about it.

If the function has to provide some mathematical operations, it must work only with types that implement the Num class type. So, the more things a value can possibly be, the less the compiler can reason about what it actually is. The freedoms we take at one level, lead to restrictions at another level [9].

Type classes allow a programmer to manage that balance, letting the functions be as abstract as possible under certain constraints. A function that only needs to perform an addition should not have the power to do something else, like printing on the screen.

This prevents unwanted side effects by constraining the things that a function can do without affecting the functionality of the function.

1.5. Lambda Cube

In the 1930s, Alonzo Church devised a new model of computation called lambda calculus. Calculus is a method of calculation or reasoning, and a lambda calculus is a process for formalizing a method. Like Turing machines, lambda calculus formalizes the concept of effective computability and determines which classes of problems can be solved. [10]

All functional programming languages are based on lambda calculus. All the Haskell functionalities are translatable into lambda calculus, and this is the reason why Haskell is considered a pure functional language.

<i>Type</i> ::= *	-- base type
<i>Type</i> → <i>Type</i>	-- function type
(<i>Type</i>)	-- grouping
<i>Term</i> ::= <i>TermVar</i>	-- term variable
<i>Term</i> <i>Term</i>	-- term application
λ <i>TermVar</i> : <i>Type</i> . <i>Term</i>	-- term abstraction
(<i>Term</i>)	-- grouping

Figure 1.9 Simply typed lambda calculus syntax¹

¹ Taken from <https://babel.ls.fi.upm.es/~pablo/Papers/Notes/f-fw.pdf>

function that takes a type and returns a term is defined; and universal application - which is the application of a type variable to the previous lambda function (hence the idea of terms depending on types).

```
Kind ::= *

Type ::= TypeVar
      |  $\forall$  TypeVar : Kind . Type  $\rightarrow$  Type

Term ::=  $\Lambda$  TypeVar : Kind . Term      -- universal abstraction
      | Term Type                       -- universal application
```

Figure 1.11 System F syntax³

In simply-typed lambda calculus, you lack the power of defining generic functions. For example, the identity function needs to be defined for each type even if all that differs from a definition to another is the type used. The identity functions for integers and for bools are defined in the following way in simply-typed lambda calculus: $\text{id: Int} \rightarrow \text{Int}$ and $\text{id: Bool} \rightarrow \text{Bool}$. System F allows the generalization of the identity function, so the function becomes $\text{id:: forall a . a} \rightarrow \text{a}$.

Haskell uses type classes in order to define polymorphic functions that will work on every data type that implements that type class. This is called parametric polymorphism, and it was introduced in many programming languages (Scala, Visual Prolog, Python, Ada). Java introduced “generics” for parametric polymorphism.

The extension of System F by adding types depending on types results in System F_w, which is the type system behind Haskell. The property of having types depending on types is also called having type operators or type constructors. This allows the definition of generic containers like lists whose types are abstractions that are waiting for their application in order to be concrete.

The type constructor of a list is the following: $\text{data List a} = \text{Nil} \mid \text{Cons a (List a)}$ where a is the type of the elements inside the list. The type becomes concrete (*) only when it receives another concrete type as an argument, for example, $\text{Nil:: List Integer}$. System F_w adds another couple of syntax rules (figure 1.12).

³ Taken from <https://babel.ls.fi.upm.es/~pablo/Papers/Notes/f-fw.pdf>

```

Kind ::= *
      | Kind → Kind

Type ::= Type Type           -- type application
      | λ TypeVar : Kind . Type -- type abstraction

```

Figure 1.12 System F_ω syntax⁴

The first syntax rule is type abstraction which is a lambda function that accepts a type and returns a type and the second rule is type application which is the application of the previous lambda function. A new Kind was also added, which defines a type that is not saturated or concrete, like the List type, and it waits for a type argument. The type abstraction can have unsaturated lambda type variables, which permit defining type constructors that have multiple arguments and their partial application. [11]

Java has polymorphism, but it doesn't allow partial application for type constructors. This is why the type system behind Java is System F.

1.6. Purescript

Purescript is a strongly, statically typed, purely-functional programming language that compiles to Javascript, written in and inspired by Haskell. It was initially designed by Phil Freeman in 2013, and it can be used to develop web applications, server-side applications, and also desktop applications with the use of Electron.

PureScript uses strict evaluation (as opposed to Haskell), type inference and persistent data structures. It also adds support for row polymorphism and extensible records. Purescript can also interact with JavaScript using the foreign function interface, making possible the use of some JavaScript libraries like `Material Design Web`. The main UI libraries used with PureScript are React-Basic and Halogen (presented in a further chapter).

⁴ Taken from <https://babel.ls.fi.upm.es/~pablo/Papers/Notes/f-fw.pdf>

Chapter 2. Effects

2.1. Introduction

This chapter introduces effects and monads. Further on, the chapter will present the “onion architecture” [12] and the mechanisms that are able to implement this type of architecture. The chapter consists of 6 sections: an introduction, two sections that introduce effects and monads, one section that presents Software Transactional Memory [13], and two chapters that provide solutions to the “onion architecture” [12].

2.2. What is an effect?

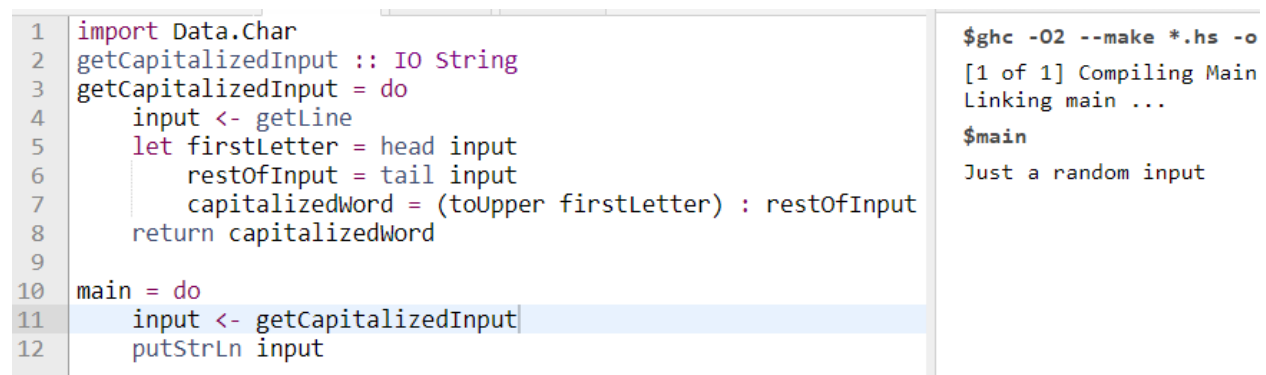
In Haskell, a pure function returns the same output for the same input. However, a function that reads a string from the console and returns that string is generally considered impure, having side effects. It could be made pure if the entire console could be passed as an argument. Haskell somehow simulates the idea of passing the entire console as an argument by encapsulating that value inside a type that is representative of the desired context.

Modifying the outside world willingly is called an effect. The type that deals with all the effects (the outside world) in Haskell is IO (shorthand for Input/Output), and all the following effects can happen in IO: reading from the console, writing to console, generating random values, getting the current date-time, creating threads, reading/writing transactional variables and handling exceptions.

A way to isolate the effects is to create your own types and encapsulate effects that can modify only a small part of the outside world. For example, a type called STM can only run effects related to the transactional memory.

In figure 2.1, the function defined at line 2 takes the first line from the console, capitalizes the string and returns the result. In imperative programming languages, this function would be considered impure. However, the function is not returning a string but the same output: an IO type

that will return a `String` when executed by the Haskell runtime. In the absence of runtime exceptions, the type of the returned value will be `String`.

The image shows a code editor with Haskell code on the left and its compilation output on the right. The code defines a function `getCapitalizedInput` that takes a line of input, capitalizes the first letter, and returns the result. The `main` function calls `getCapitalizedInput` and prints the result. The output shows the compilation process with flags `$ghc -O2 --make *.hs -o`, indicating it's compiling `Main` and linking it. The final output is `$main` followed by `Just a random input`.

```
1 import Data.Char
2 getCapitalizedInput :: IO String
3 getCapitalizedInput = do
4     input <- getLine
5     let firstLetter = head input
6         restOfInput = tail input
7         capitalizedWord = (toUpper firstLetter) : restOfInput
8     return capitalizedWord
9
10 main = do
11     input <- getCapitalizedInput
12     putStrLn input
```

`$ghc -O2 --make *.hs -o`
[1 of 1] Compiling Main
Linking main ...
\$main
Just a random input

Figure 2.1. IO function in Haskell

This type of function is called *effectful*, and it's basically a function whose result might encapsulate an effect that can alter the outside world, however, due to the fact that it returns the same result, even if at runtime that result produces different outputs when executed, the function is still considered *pure*. This is the reason why Haskell is a pure programming language: in Haskell, all functions are pure. [6]

2.3. What's a Haskell monad?

A monad can be explained informally as a programmable semicolon [14]. It provides functions that make possible the sequencing of some actions. In imperative programming (Java, C++), one can easily sequence multiple operations by separating them with a semicolon or a new line. In Haskell, where the evaluation is lazy, it's hard to reason about the order of evaluation because the expressions are evaluated when needed and not in sequential order.

Monads let us create blocks of instructions that are executed in sequential order. Each monad has a context in which it executes those instructions, and each instruction can modify the monadic context. The context of a monad represents the shape of the result obtained after executing the monad instructions. For example, the data type `list` implements the monad type-class, and its monadic context is the size of the list.

The programmable `semicolon` of a monad is the function `>>=`, also known as `bind`. This function executes (unwraps) a monad and passes the result to a function that will return a new monad instance with a new context. The only restriction is that the returned monad must have the same type as the input monad, as the bind signature suggests: `>>= :: m a → (a → m b) → m b` (`m` is the monad type, and `a` and `b` are the return types for those monads).

For example, the following function call: `[1,2,3] >>= (\number -> [number, number]) = [1,1,2,2,3,3]`, executes the list monad (`m` is `[]` and both `a` and `b` are `Number`), and for each element inside, it creates a new list with two elements equal with the input value. Finally, all the created lists are concatenated, and the result will be the output of the bind function.

<pre> 1 import Data.List 2 main = do 3 print \$ compareScores "Marius" "Dorel" 4 print \$ compareScores "Marius" "Alin" 5 print \$ compareScores "Alin" "Marius" 6 print \$ sameCompareScores "Marius" "Dorel" 7 print \$ sameCompareScores "Marius" "Alin" 8 print \$ sameCompareScores "Alin" "Marius" 9 10 compareScores :: String -> String -> Maybe Bool 11 compareScores firstName secondName = do 12 let dictionary = [("Marius",1), ("Alin",2), ("Gina",4)] 13 firstScore <- lookup firstName dictionary 14 secondScore <- lookup secondName dictionary 15 return \$ firstScore > secondScore 16 17 sameCompareScores :: String -> String -> Maybe Bool 18 sameCompareScores firstName secondName = 19 let dictionary = [("Marius",1), ("Alin",2), ("Gina",4)] in 20 (lookup firstName dictionary) 21 >>= \firstScore -> (lookup secondName dictionary) 22 >>= \secondScore -> Just \$ firstScore > secondScore </pre>	<pre> \$ghc -O2 --m [1 of 1] Com Linking main \$main Nothing Just False Just True Nothing Just False Just True </pre>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------

Figure 2.2 Maybe monad, and `>>=`

Every language needs to specify the lack of existence of some object or result (none, null), and Haskell elegantly makes this possible with its Maybe monad. The Maybe data type has only two data constructors, `Nothing`: which doesn't take any parameter, and `Just`: which takes one parameter of any type.

In figure 2.2 at line 20, the function lookup tries to find the argument `name1` in the dictionary defined in the previous line, and it returns `Nothing` if `firstName` was not among the keys or `Just` value otherwise. The result has the type `Maybe Number`, so the needed score is encapsulated in the monad `Maybe`. However, the function has to extract the data from the monad and pass it to another function that will bring the desired result.

In figure 2.3, at lines 5 and 6, the function `>>=` (which is infix) receives two parameters: a `Maybe` value and a function. If the value is `Nothing`, the result of the function `>>=` will automatically be `Nothing`. Otherwise, the output will be the result of the function received as a parameter, called with the data inside the `Maybe` value.

```
1  import Control.Monad
2
3  instance Monad Maybe where
4  return = Just
5  Nothing >>= f = Nothing
6  (Just x) >>= f = f x
7  fail _ = Nothing
```

Figure 2.3 Maybe monad type class

In figure 2.2, the evaluation of the function defined at line 17 has a short-circuit/minimal evaluation (like the `and/or` evaluation): If one of the lookups from lines 20-21 returns `Nothing`, the function returns `Nothing` without further evaluations. Only if both results are `Just` values, the comparison from line 22 can be made.

The function ``compareScores`` from figure 1.7, defined at line 10, is exactly the same function as the one defined at line 17. The notation of ``do`` is just syntactic sugar that translates to the form of the function, defined at line 17, when compiled. This notation helps programmers to code in a more readable way the sequencing of actions.

The binder `<-` runs the computation on the right side and binds its result to the name specified on the left side. Each row from the `do` block is interpreted as a `>>=` function application that extracts a monadic value and passes that value to the function associated with the next row. The last row of the `do`-block cannot extract another value because there will be no more functions to receive the extracted value.

Every time there is a do-block in a function, that function operates inside a monad. The entry point of a Haskell program is the main function with the signature: `IO ()`, which is indeed a monad. This means we have used monads right from the beginning for each of our very basic examples.

2.4. STM

Software Transactional Memory, or STM, is a library that allows using transactions rather than locks to synchronize processes that execute in parallel and share memory. A transaction makes two guarantees: atomicity (all the effects of the transaction block are visible at once to another thread) and isolation (the transaction is completely unaffected by other threads) [13].

```
1 transfer :: Account -> Account -> Int -> IO ()
2 transfer from to amount
3   = atomically (do deposit to amount
4                  withdraw from amount)
```

Figure 2.4 Atomically in Haskell

In figure 2.4, at line 1, we define a function that transfers an amount of money from a bank account to another. The critical region of the function is passed as an argument to the ``atomically`` function call, which assures that the block of code received will be executed as a transaction.

It looks like the function ``atomically`` is performing a ‘global lock’, assuring no other ``atomically`` function is run simultaneously. However, while we can assume a global lock, the actual implementation is based on optimistic locking.

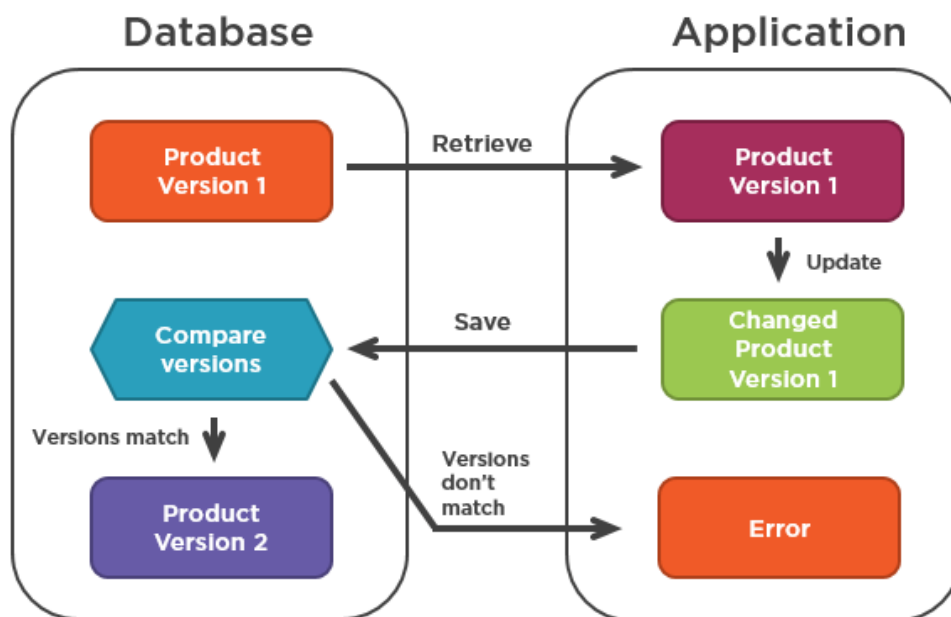


Figure 2.5 Optimistic locking⁵

As figure 2.5 suggests, in optimistic locking, each table has a version that is updated for any modification brought to the table. At the end of any modification, the version of the table received at the beginning of the operation is compared to the actual version of the table. If versions don't match, an error occurs, or the process is automatically retried depending on the locking implementation. Otherwise, the modifications are brought to the actual table, and its version is updated. [15]

The function ``atomically`` can do the following things at the exit of a transactional block: if no concurrent thread modified the same data, all the modifications would be available at once to other threads. Otherwise, the transactional block is automatically retried.

The function `atomically` has the following signature: `atomically:: STM a -> IO a`. This means it can run some computations defined inside the STM monad class, like modifying or reading TVars and make the results visible by running an IO computation. A TVar is a transactional variable that

⁵ Taken from <https://enterprisecraftsmanship.com/posts/optimistic-locking-automatic-retry/>

encapsulates a value that can be destructively modified inside the STM monad. No other side effects are allowed.

2.5. MTL

Most monads encode some form of side-effect: for example, Maybe monad helps to define if some data is defined or not (this replaces the `null` values in most other languages), IO monad creates side effects like printing on the screen or reading data, and State is holding a state variable which can be set or read. However, a `do` block restricts the programmer to use only one type of monad, restricting the functionality that a block of code can have.

Composing two monads won't result in another monad, so one way to use multiple functionalities is to stack the monads (having one monad inside of another). This is one solution to the onion architecture problem, which involves structuring the application as a series of layers. At the centre of the application, there is the business logic layer, wrapped by other layers, each with different functionality like logging or handling exceptions. [12]

Propagating the effects throughout all the monad stack can be done with the monad transformers defined in the MTL library. MTL, or monad transformer library, consists of a set of monads, together with their monad transformers. A monad transformer is a monad that can peel away a monad layer. For example the maybe monad transformer: `newtype MaybeT m a = MaybeT {runMaybeT :: m (Maybe a)}`, receives a monad `m` with its return type `a` [6].

The getter `runMaybeT` peels away the `MaybeT` monad layer by running it and brings the `Maybe` functionalities to the data inside the other monad `m`. If the `m` monad is another transformer, like `EitherT`: `newtype EitherT e m a = EitherT { runEitherT :: m (Either e a) }`, the process can continue: `runEitherT (EitherT String IO (Maybe a)) = IO (Either String (Maybe a))`. This `runMonadT` getter is used to extract the return value from a monad and pass it to the function argument of `>>=` for the monad transformer.

```

1 instance (Monad m) => Monad (MaybeT m) where
2     return = pure
3     (>>=) :: MaybeT m a -> (a -> MaybeT m b) -> MaybeT m b
4     maybeTvar >>= f =
5         MaybeT $ do
6             v <- runMaybeT maybeTvar
7             case v of
8                 Nothing -> return Nothing
9                 Just y -> runMaybeT (f y)

```

Figure 2.6 MaybeT monad instance Haskell

In figure 2.6, there is the implementation of the monad type class for the monad transformer MaybeT. In line 6, the value inside the monad m is extracted and wrapped in a Just if it exists or Nothing otherwise. If the monad m is IO, it might throw an error. In that case, there won't be any value inside IO, and the v variable from line 6 will be Nothing, making the `do` block to return (m Nothing). Otherwise, the `do` block will apply the argument y on the function f, and after that will extract the monad inside the MaybeT result with the getter runMaybeT. However, the result of the function >>= must be of the MaybeT type, so the result from the do block is wrapped in the MaybeT data constructor.

When we have a computation stack, different computations have to happen at different stack levels. Maybe effects have to happen at the level where the Maybe monad is placed. MTL takes care of searching for the proper level of our computation using the type-system, but we still need to specify a way to lift our computation through the monadic stack.

For that purpose we use the lift :: (Monad m) => m a -> t m a function. The lift function is used to lift a monad over another monad.

2.6. DSLs

DSLs or domain-specific languages are “mini-languages” that are designed to solve a certain problem in an application. In functional programming, a DSL is generally embedded, meaning data types and operators from the host language are used to construct programs in that DSL. The

programs written in the DSL are represented as abstract syntax trees that can be parsed and interpreted in multiple ways. An interpreter defines the meaning (semantics) of the grammar, and it returns an appropriate result.

The main benefit of using DSLs and interpreters is that it gives you the power of introspection. Having a program written inside the IO monad lacks introspection due to the monadic bind. As the signature of the bind function suggests, $\gg=:: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$, the information of previous monadic context is lost. Therefore the program will be a sequence of chunks of machine code that cannot be interpreted, introspected or transformed.

Haskell makes possible the composition of interpreters and algebras, which offers a nice solution to the onion layer architecture. In the onion architecture, each layer of the program has different functionality, and each layer can have a DSL and an interpreter that enhances the DSL and the interpreter from the previous layer.

```
1 data CloudFilesF a
2   = SaveFile Path Bytes a
3   | ListFiles Path (List Path -> a)
4
5 type CloudFilesAPI a = Free CloudFilesF a
6
7 saveFile :: Path -> Bytes -> CloudFilesAPI Unit
8 saveFile path bytes = liftF (SaveFile path bytes Unit)
9
10 listFiles :: Path -> CloudFilesAPI (List Path)
11 listFiles path = liftF (ListFiles path id)
12
13 data HttpF a
14   = GET    Path (Bytes -> a)
15   | PUT    Path Bytes (Bytes -> a)
16   | POST   Path Bytes (Bytes -> a)
17   | DELETE Path (Bytes -> a)
18
19 data LogF a = Log Level String a
20
21 cloudFilesI :: forall a. CloudFilesF a -> Free HttpF a
22 logCloudFilesI :: forall a. CloudFilesF a -> Free LogF Unit
23 loggingCloudFilesI :: forall a. CloudFilesF a -> Free (Coproduct LogF HttpF) a
24
25 executor :: forall a. Coproduct LogF HttpF a -> IO a
```

Figure 2.7 Example of DSL together with its interpreters

This 'pattern' can also be found in the MTL transformers stack. However, MTL doesn't allow for introspection, which means the programs cannot be optimized by applying dynamic

transformations. Free algebras allow more decoupling between interpreters, and they also allow introspection and minimal implementation of each semantic layer.

Figure 2.7 is a sketch of a program that saves a file to a cloud store, retrieves the list of the files added and uses logging functionality. The first three lines define the algebra (the set of operations the program can be reduced to) for the cloud files API, on which a lightweight DSL is constructed at lines 5-11.

A free algebra makes the DSL independent from their interpreters, in the same way interfaces are independent of their implementations. The created DSL defines the semantics of the cloud files API but does not specify how the functions should obtain the desired result. This is done by the implementations of the interpreters, which will specify exactly what operations will be made in order to obtain a result.

At lines 13-17, we define an algebra for a REST API that can be used to express the semantics of CloudFilesF, and at line 19, a new algebra is defined for the logging functionality. In lines 21-23, there are three interpreters defined. The first function interprets a CloudFilesF operation into a “program” of operations in HttpF, the second function interprets a CloudFileF operation into LogF, and the last function interprets a CloudFileF operation in a combination of the two algebras. Finally, the resulting algebra is mapped into IO by the executor function. [\[16\]](#)

Chapter 3. Practical application

3.1. Introduction

This chapter presents the life cycle of this application (Analysis, Design and Testing) together with the most important libraries used in its implementation. The next three subsections will present the requirements of the application together with its design and finally testing.

Section 2 presents the Servant API library, diving into the Servant's types and functions. Section 2 also presents how is software transactional memory used inside a stack monad transformer and how does a Servant API interact with a monad transformer stack. The next section (Section 3) presents how easily the database of our Servant application can be changed and what are the mechanisms that make this lightweight migration happen.

Section 4 presents a library called Halogen, which is used in our web application written in Purescript, and the last section (Section 5), introduces Microsoft Teams bots and tabs and presents the way our application uses these functionalities.

3.1.1. Requirements and Specifications

Our application helps with the seminar management, and it needs to be integrated with the Microsoft Teams platform. This implies two basic actions that a user can perform: adding and removing both an activity or an attendance. These actions should be performed in a Microsoft Teams tab or application. The application should also provide a way such that a student can register attendance for a given seminar.

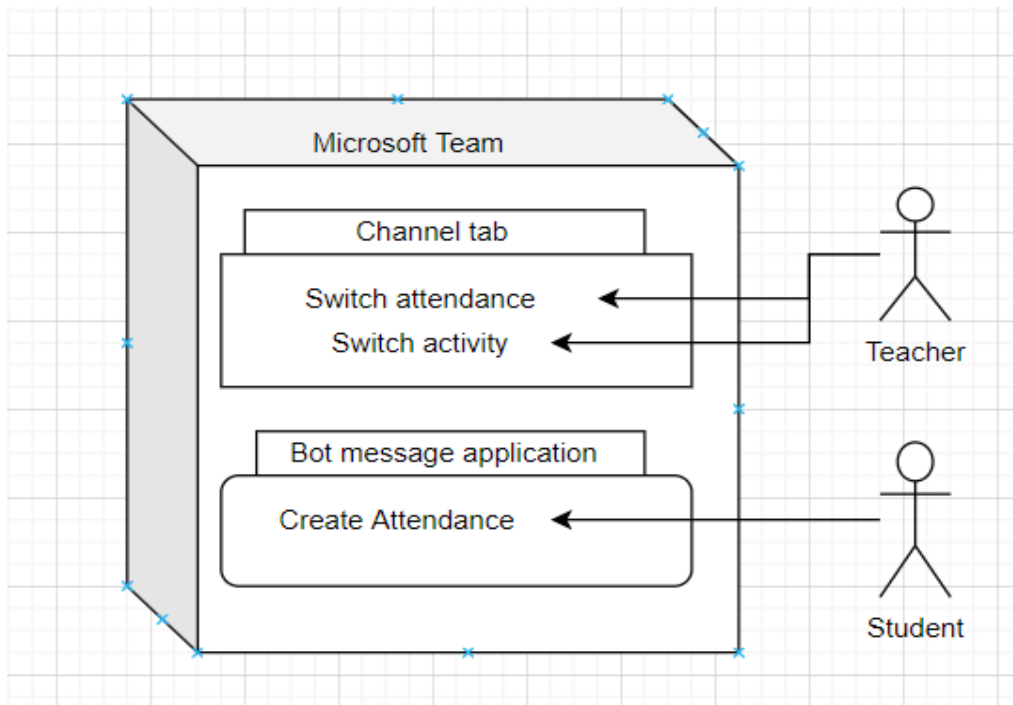


Figure 3.1 Use case diagram of the application

Instead of having two different buttons for adding and removing an entity, the teacher will have to press the same button that will switch the activity on or off, or it will add/remove an attendance. Figure 3.1 is a use case diagram that specifies the types of users that can use this application together with the corresponding actions they can perform. A student can only create an attendance, and a teacher can switch an attendance or an activity inside a Microsoft Team Tab.

3.1.2. Design

In order to use our application inside a Microsoft Teams tab, we need a web application that will be rendered inside an iframe tag in the Microsoft Teams tab. Purescript is a pure functional programming language, and it has great support for building type-safe user interfaces using libraries like Purescript-React-Basic or, in our case, Halogen. Halogen is a library written in Purescript that allows for the creation of encapsulated components that have their own state and can compose together to build complex user interfaces.

Our application uses Purescript for the frontend part and Haskell for the backend. The backend implements an API server that will communicate with the Purescript web application via API calls. The API server is implemented using the Servant library that facilitates the construction of a type-safe server and also provides good support for testing our application.

The feature that a student can create an attendance is implemented using a direct exchange of messages between the student and a Microsoft Teams bot that sends the messages to an API endpoint from the Servant server. The endpoint that communicates with the bot is constructing the attendance progressively using Software Transactional Memory that allows updating destructively and atomically a state that contains all the information needed for the creation of an attendance.

For the information storage, we use an SQL database which is an implementation of a type-class that specifies the crud operations for our entities.

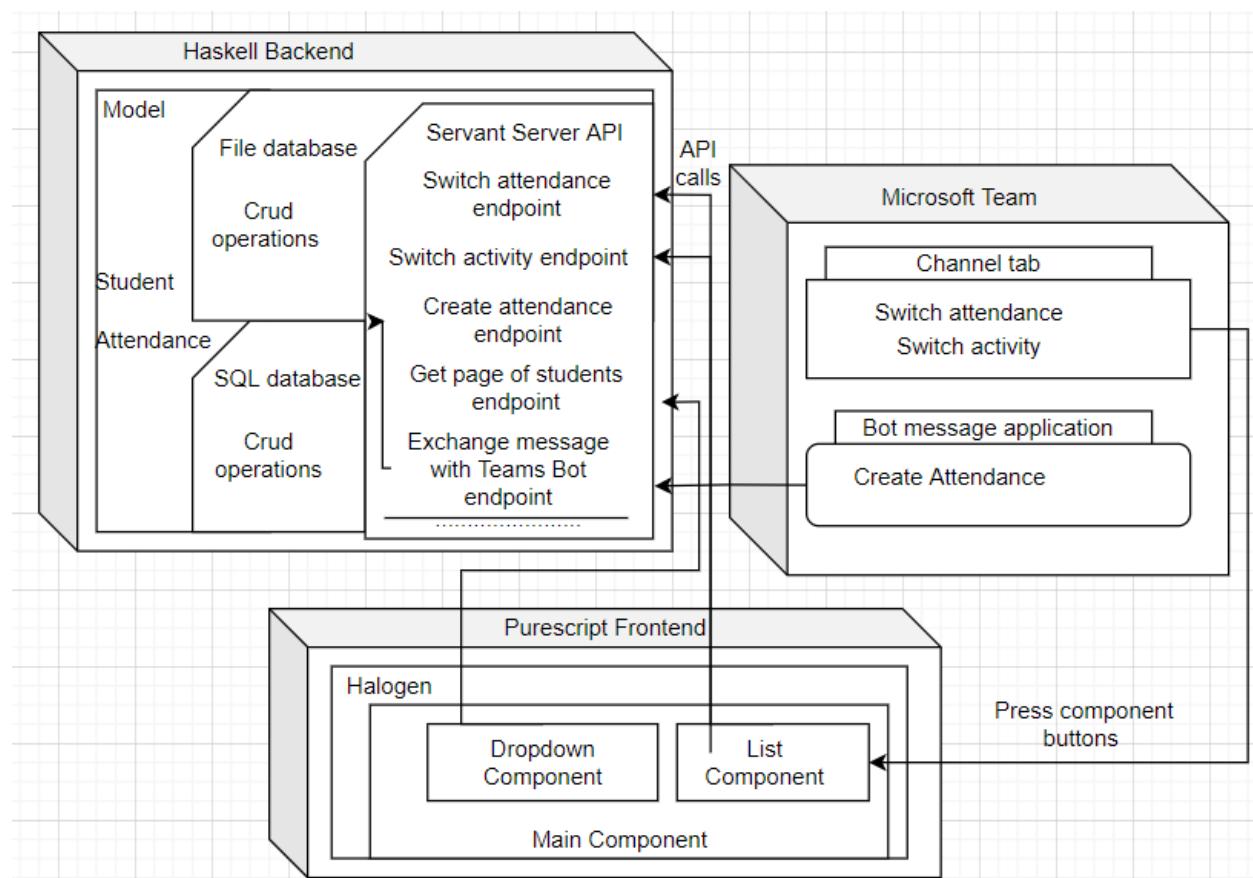


Figure 3.2 Communication diagram of the application

Figure 3.2 displays a communication diagram that shows how the main components from this application are communicating, together with their communication with Microsoft Teams.

3.1.3. testing

Our application uses the Servant library in order to perform integration testing for most of the server API endpoints. Servant has the ability to generate clients for a given API server and test if the actual result of an API call matches the expected result. Servant generates the source code from the type of the server API, following the approach of correctness by construction.

3.2. Servant

3.2.1. Servant Introduction

Servant is a set of libraries for writing type-safe web applications. This is achieved by representing the web API at the types level, which makes possible the verification that the server-side request handlers implement the API faithfully. Servant follows important principles like concision (the serialization and the deserialization are done at the level of types automatically), flexibility (Servant is open to extensions), separation of concerns (the handlers and the HTTP logic should be separate), and, maybe the most important, type safety (it makes sure the API meets the specification).

3.2.2. API types

Servant brings into the types level every detail of an API, from the URL construction to the HTTP request methods like GET, POST, UPDATE, and DELETE.

```

1 type API = "student" :> Capture "id" Integer      :> Get '[JSON] (Maybe Student)
2   :<|> "student"    :> ReqBody '[JSON] Student    :> Post '[JSON] ()
3   :<|> "student"    :> ReqBody '[JSON] Student    :> Put '[JSON] ()
4   :<|> "student"    :> Capture "id" Integer      :> Delete '[JSON] ()
5   :<|> "students"   :> "sort"      :> Capture "seminar" Integer
6   :> Capture "group" String      :> Get '[JSON] [Student]
7   :<|> "students"   :> "filter"    :> Capture "name" String
8   :> Get '[JSON] [Student]
9   :<|> "attendance" :> Capture "id" Integer      :> Get '[JSON] (Maybe Attendance)
10  :<|> "attendance" :> ReqBody '[JSON] Attendance :> Post '[JSON] ()
11  :<|> "attendance" :> ReqBody '[JSON] Attendance :> Put '[JSON] ()
12  :<|> "attendance" :> Capture "id" Integer      :> Delete '[JSON] ()
13  :<|> "attendances" :> Capture "seminar" Integer
14  :> Capture "group" String      :> Get '[JSON] [Attendance]
15  :<|> "activity"   :> ReqBody '[JSON] Attendance :> Post '[JSON] ()
16  :<|> "message"    :> ReqBody '[JSON] Message   :> Post '[JSON] ()

```

Figure 3.3 API of the practical application

Figure 3.3 represents the API type used by our application to implement the rest logic. The API type is constructed sequentially by the connectors `:<|>` and `:>`. The first connector, `:<|>`, links another endpoint to the API, and the last one separates the combinators that are building the endpoint.

These combinators actually formalize the specification of that API endpoint. For example, the first endpoint declared at line 1 says that it can be called at the URL `student/`, followed by an Integer identified by the name `id`. The combinator `"Get '[JSON] (Maybe Student)"` says that the endpoint is accessible through the HTTP GET request, and it will return a Maybe Student that is gonna be encoded as JSON.

The encoding can be done under different formats like `PlainText`, `FormUrlEncoded` or `OctetStream`, by adding more options in the square brackets: `"Get '[JSON, PlainText, FormUrlEncoded, OctetStream] (Maybe Student)"`. The choice of the encoding is determined by the `Accept` header specified in the client's request.

The next endpoint from line 2 has a `"ReqBody '[JSON] Student"` combinator, which gets the body of the post request, and deserializes the body into a data value of type `Student`. In the case of JSON, encoding and decoding are done automatically if the `Student` type has an instance of the type classes `ToJson` and `FromJson` from the `Data.Aeson` library.

The process of encoding/decoding JSON values into/from data types implies using a form of abstraction that allows defining functions that can operate on all the datatypes from the class ToJson/FromJson. This is called generic programming, and Servant uses this approach in multiple situations. For example, the record-based generic approach lets us define generic APIs inside a data type, from which a Proxy or a safe link can be easily generated by using generic functions like genericApi or fieldLink.

There are other combinators that could be used in an API endpoint, like QueryParam, QueryFlag, QueryParams and Header. “QueryParam “name” a” becomes an argument of type Maybe a, “QueryFlag “name”” a Bool, and “QueryParams “name” a” becomes an argument of type [a]. The request response, like a usual HTTP request, can also have headers: “Get ‘[JSON] (Headers ‘[Header “My-Header” Integer] (Maybe Student))”. A header has a name and a value, and all the headers are given as a list of parameters to the Headers combinator. [\[17\]](#)

An endpoint can also be declared protected by providing a Basic Authentication to an API. In order to have all our API protected, we could use the following type as API: type ProtectedAPI = BasicAuth “realm” User := API, where API is declared in figure 3.1. The BasicAuth combinator returns a type that holds the information about the user logged in, and the API's endpoints are now available only after successful authentication.

The request combinators: Get, Post, Delete, Patch, Put provide the return type of the handlers that are interpreting the endpoints. That type is wrapped in the Handler monad, so the signature of the handler for the get request defined at line 1 in figure 3.1 can be the following: createStudentHandler:: Integer -> Handler (Maybe Student). The Integer argument is from the Capture combinator, and the Maybe Student is the return type specified in the Get combinator.

The Handler monad is basically a simple monad that can throw a ServerError or perform IO actions: newtype Handler a = Handler {runHandler':: ExceptT ServerError IO a}. All the handlers are grouped together in a cluster called Server. The Server type takes an API as a parameter, and for each endpoint in the API, it has a handler, taken in the same order.

Next, a function called serve with the following signature: serve:: HasServer api '[] => Proxy api -> Server api -> Application, takes a proxy to an API, a server fed with that api and returns an Application. The application is turned into a running server by the function found in the

Network.Wai module called `run :: Port -> Application -> IO ()`, which takes a port, and an application and runs the application on localhost on that port.

However, the Handler monad may not be the perfect choice for your handlers. Some endpoints may need a state that contains a database configuration, which implies the Reader monad, others may need a different monad, so the solution provided by Servant is the `hoistServer` function :: `HasServer api '[] => Proxy api -> (forall x. m x -> n x) -> ServerT api m -> ServerT api n`. With `hoistServer`, you can provide a function to convert any number of endpoints from one type constructor to another. [17]

```

1  runServer :: AppConfig SQLConnection -> IO()
2  runServer config = run 8081 $ app config
3
4  app config = cors (const $ Just policy) $ serve api $
5      hoistServer api (nt config) server
6  where
7  policy = simpleCorsResourcePolicy {
8      corsRequestHeaders = ["Content-Type"],
9      corsMethods = "DELETE" : ("GET":("PUT":( "POST" : simpleMethods)))
10     }
11  nt config reader = liftIO $ runReaderT reader config
12
13  api :: Proxy API
14  api = Proxy
15
16  server = findStudHandler    :<|> addStudHandler    :<|> updateStudHandler
17          :<|> removeStudHandler :<|> sortStudsHandler :<|> filterStudsHandler
18          :<|> findAttHandler    :<|> switchAttHandler :<|> updateAttHandler
19          :<|> removeAttHandler  :<|> getAttsHandler   :<|> switchActHandler
20          :<|> teamsAttHandler
21
22  type App a m v = ReaderT (AppConfig a) m v
23  data AppConfig a = AppConfig
24  { connection :: a
25    , state      :: TVar (Map.Map StudentName (Maybe Attendance))
26  | }

```

Figure 3.4 From Handlers to Application, function `app` from practical application

Figure 3.4 has all the functions needed to transform the API from figure 3.3 to an actual REST API server. The function `server` gathers all the handlers that run their computations inside a `ReaderT` monad. The reader monad from the transformer stack holds all the information needed for the database connection and for the `teamsAttHandler`.

The ModelAPI type class has defined all the functions used inside the `server` handlers (findStudent, createStudent and so on), and each of that function returns a result encapsulated in the `m` monad from the ModelAPI. (Figure 3.5)

```

1 class ModelAPI a m where
2   createStudent    :: MonadIO m => Student    -> a -> m ()
3   removeStudent    :: MonadIO m => StudentId   -> a -> m ()
4   updateStudent     :: MonadIO m => Student     -> a -> m ()
5   findStudent       :: MonadIO m => StudentId   -> a -> m (Maybe Student)
6   getAllStudents    :: MonadIO m =>              a -> m ([Student])
7   createAttendance  :: MonadIO m => Attendance  -> a -> m ()
8   removeAttendance  :: MonadIO m => AttendanceId -> a -> m ()
9   updateAttendance  :: MonadIO m => Attendance  -> a -> m ()
10  findAttendance     :: MonadIO m => AttendanceId -> a -> m (Maybe Attendance)
11  getAllAttendances :: MonadIO m =>              a -> m ([Attendance])

```

Figure 3.5 ModelAPI type-class

Going further, the serve function from figure 3.4, line 4 needs a Server type which is basically ServerT API Handler, and the hoistServer function from the following line transforms the ReaderT m a monad into a Handler. The natural transformer function used by hoistServer is: “nt config reader = liftIO \$ runReaderT reader config”.

The function runReaderT will bring the Reader functionality to the values inside the generic monad m (Monad m can use the information held in the reader monad), transforming the ReaderT config m monad into m.

The liftIO :: (MonadIO m) => IO a -> m a function forces the generic monad m to be the IO monad which will be lifted to the Handler monad because Handler monad has an instance of the MonadIO. If we need to use a ModelAPI instance that has a different monad, the natural transformation function needs to be updated.

Figure 3.6 implements the ModelAPI class, specifying an SQLConnection data type is bound to the monad IO. The natural transformation (nt) defined in figure 3.4, line 11 restricts the monad from the ModelAPI type class to be IO, but it doesn't specify what type of instance will be used. This is automatically inferred from the argument given to the type constructor AppConfig received by the runServer function.

```

1 instance ModelAPI SQLConnection IO where
2   createStudent :: Student -> SQLConnection -> IO ()
3   createStudent student conn = do
4     db <- open (location conn)
5     execute db "INSERT into student(_sId,_sName,_sGroup,_sCode)
6               VALUES (?, ?, ?, ?)" student
7     close db
8   removeStudent :: StudentId -> SQLConnection -> IO ()
9   removeStudent studentId conn = do
10    db <- open (location conn)
11    execute db "DELETE from student WHERE _sId = (?)" (Only studentId)
12    close db
13   updateStudent :: Student -> SQLConnection -> IO()
14   updateStudent (Student sid sname sgroup scode) conn = do
15     db <- open (location conn)
16     execute db "UPDATE student set _sName = (?),_sGroup = (?),_sCode = (?))
17               WHERE _sId (?)" (sname, sgroup, scode, sid)
18     close db
19   findStudent :: StudentId -> SQLConnection -> IO(Maybe Student)
20   findStudent studentID conn= do
21     db <- open (location conn)
22     result <- query db "SELECT * from student where _sid = (?)" (Only studentID)
23     :: IO [Student]
24     close db
25     .
26     .
27     .

```

Figure 3.6 ModelAPI instance

The function `runServer` from figure 3.4 receives an `AppConfig SQLConnection` as an argument, which holds the information about the database and a transactional variable. The type signature of the handlers (figure 3.7) binds the generic data type of the `AppConfig` to be part of a `ModelAPI` instance.

```

1 findStudHandler :: (ModelAPI a m, MonadIO m) => Integer -> App a m (Maybe Student)
2 findStudHandler id = do
3   config <- ask
4   lift $ findStudent id (connection config)
5
6 removeStudHandler :: (ModelAPI a m, MonadIO m) => Integer -> App a m ()
7 removeStudHandler id = do
8   config <- ask
9   lift $ removeStudent id (connection config)
10
11 updateStudHandler :: (ModelAPI a m, MonadIO m) => Student -> App a m ()
12 updateStudHandler student = do
13   config <- ask
14   lift $ updateStudent student (connection config)

```

Figure 3.7 Server handlers

Finally, the app function from figure 3.4 returns an application that will be transformed to a real web server by the function `runServer` defined at line 1. The `cors` function called in the app function only adds an `Access-Control-Allow-Origin` header, with the basic HTTP requests, to the response.

3.2.3. STM

Servant works together with STM libraries, usually when a global state of the API is needed. The state of the API can have TVars that can be safely modified inside atomic blocks when needed or MVars which need regular locks. In the practical application, one functionality creates a connection with a Microsoft Teams bot, and it creates an Attendance incrementally after changing messages with the user.

```
1 teamsAttendance mess state conn = do
2   map <- liftIO $ readTVarIO state
3   let studName = (name <$> from) mess
4       attValue = fromMaybe Nothing (Map.lookup studName map)
5
6   listOfStudents <- getAllStudents conn
7   let tupleOptions = setOption attValue (text mess) listOfStudents
8   case fst tupleOptions of
9     Nothing -> return ()
10    Just att -> case snd tupleOptions of
11      "You're all set. See you later." -> do
12        createAttendance att conn
13        liftIO $ atomically $ modifyTVar state (\x -> Map.delete studName x)
14      _ -> liftIO $ atomically $ modifyTVar state
15        (\x -> Map.insert studName (fst tupleOptions) x)
16    liftIO $ sendRequest (snd tupleOptions) mess
17  return ()
```

Figure 3.8 STM in practice, Microsoft Teams connection handler

The incremental creation of an Attendance needs a state for that API endpoint that can store the incomplete state of an Attendance. That incomplete state is modified each time a new message is received from the user till the Attendance is completely created and the temporary data is deleted. This is done using the transactional variable held in the `ReaderT (AppConfig a)` monad, which has a dictionary with student names as keys and `Maybe Attendance` for the values. The configuration is extracted using the `ask` function defined in the `Monad Transformer` library, and it is passed to the function `teamsAttendance` from figure 3.8.

In figure 3.8, at line 2, the mutable dictionary is read, and the tupleOption (Maybe Attendance, String) from line 7 will tell what message will be sent back to the user and if the temporary date should be modified or deleted.

If the Maybe Attendance from the tupleOption has an attendance, it will update(override) the TVar dictionary value at the key studentName, using the modifyTVar function inside an atomically block, at line 14. If the Attendance is completely set, the attendance is added to the database at line 12, using the function createAttendance from the ModelAPI. After the insertion, at line 13, the partial attendance is deleted from the global dictionary, which ends the process of creation.

3.3. Database

Our application uses an SQL backend, but it can also use a file-based one. Because the ModelAPI is like an interface, its method definitions are separated from their implementations. This, combined with the fact that Servant lets the instantiation of the ModelAPI happen as late as possible, the database can be changed very easily.

In figure 3.9, the modifications made for using a new database consist in changing the runServer signature from `:: AppConfig SQLConnection -> IO()` to `:: AppConfig Connection -> IO()`. The main function that calls the function runServer has to call it with an AppConfig Connection argument, and the database of the application is successfully switched.

```
1 runServer :: AppConfig Connection -> IO()
2 runServer config = run 8081 $ app config
3
4 app config = cors (const $ Just policy) $ serve api $
5   hoistServer api (nt config) server
6   where
7     policy = simpleCorsResourcePolicy {
8       corsRequestHeaders = ["Content-Type"],
9       corsMethods = "DELETE" : ("GET":("PUT":( "POST" : simpleMethods)))
10    }
11   nt config reader = liftIO $ runReaderT reader config
```

Figure 3.9 Servant application with file-based Database

The file-based database needs an instance of the ModelAPI (see figure 3.5), where Connection stores the path of the file in which the database is written.

```
1 instance ModelAPI Connection IO where
2   createStudent :: Student -> Connection -> IO ()
3   createStudent student = update' (cStudents %~ (student :))
4
5   removeStudent :: StudentId -> Connection -> IO ()
6   removeStudent studentID = update' (cStudents %~
7     (filter ((Student studentID "" "" "" )/=)))
8
9   updateStudent :: Student -> Connection -> IO()
10  updateStudent student = update' (cStudents %~ (map
11    (\stud -> replaceStudent stud student)))
12
13  findStudent :: StudentId -> Connection -> IO(Maybe Student)
14  findStudent studentID = queryEntity (\content -> find
15    ((Student studentID "" "" "" )==)
16    (content ^. cStudents))
17  getAllStudents :: Connection -> IO([Student])
18  getAllStudents = queryList (\content -> content ^. cStudents)
19  .
20  .
21  .
```

Figure 3.10 File-based database ModelAPI instance

For a new database, like MySQL, you only need to create an instance of ModelAPI MySQL IO and change the app' signature.

3.4. Halogen

3.4.1. Halogen Architecture

Halogen is a type-safe library for building user interfaces. Halogen dynamically renders HTML code into the DOM, similar to the way React or Angular works. The atomic block of Halogen is a Component, and it represents the smallest part that can have a state and can respond to external events.

Halogen can produce HTML code outside of a component, but given the fact that Halogen is a Purescript library and Purescript is a pure functional language, that HTML code is just a value outside of a component. Components can be grouped together to build larger components, which

means the UI can be split into independent, reusable blocks that can be tested in isolation. This is called component architecture, and it implies that a halogen application should have at least one component.

A component is built from three main parts: a component state, accessible inside the component, a render function that will render the actual HTML code, and an eval function that specifies how the component should react to external events. A component can be a parent component if it renders other components, a child component if it is rendered by a parent component and a root component if all the other components are rendered by it [18].

```
1  main :: Effect Unit
2  main = HA.runHalogenAff do
3      body <- HA.awaitBody
4      runUI Button.parentComponent unit body
```

Figure 3.11 Root component HTML code insert in the body tag

In figure 3.11, the root component of our application, `parentComponent`, is filling the body tag of the final HTML page with its HTML rendered code. The root component keeps an open slot for each child inside the HTML so that each component will produce only its part of HTML and will share it with the parent component.

In figure 3.12, there is an example of a component. The component definition from line 1 says that a component receives five arguments. An HTML code generated by the render function, a query that specifies how a parent can ask for a child's internal information, an input (for example, the inputs defined at lines 24-29 fed to the Dropdown components from lines 18-20), an output which represents a value that a child returns to its parent, and the monad inside the computations are made, in our case, monad `Aff` (an asynchronous version of the `Eff` from Purescript or `IO` from Haskell).

The parent state keeps track of the application configuration - the group, the seminar and what we want to modify (an Attendance or an Activity), and an input text used to search a student by name. The render function uses HTML tags like `label`, `div` or `span`, each having as arguments a list of attributes (class, id, `onInputValue` etc.) and a list of HTML tags.

```

1 parentComponent :: forall q i o. H.Component HH.HTML q i o Aff
2 parentComponent = H.mkComponent{
3   initialState:
4     const {inputString : "", plugin: "Attendance", _aGroup : "931", _aSeminar:1},
5   render,
6   eval: H.mkEval $ H.defaultEval
7     {handleAction = handleParent, initialize = Just InitializeParent}
8 }where
9   render :: ParentState -> H.ComponentHTML ParentAction ParentSlots Aff
10  render state = HH.div_
11    [ HH.label [classes_ ["mdc-text-field", "mdc-text-field--filled", "space"]]
12      [ HH.span [class_ "mdc-text-field_ripple"] []
13        , HH.span [ class_ "mdc-floating-label", HP.id_ "my-label-id" ]
14          [ HH.text "Student Name" ]
15        , HH.input [ class_ "mdc-text-field_input", HA.labelledBy "my_label_id"
16                    , HP.type_ HP.InputText, HE.onValueInput (Just <<< FillInput)]
17          , HH.span [classes_ ["mdc-line-ripple"]][[]]
18        , HH.slot label 0 Dropdown.component input1 (Just <<< PluginHandle)
19        , HH.slot label 1 Dropdown.component input2 (Just <<< GroupHandle)
20        , HH.slot label 2 Dropdown.component input3 (Just <<< SeminarHandle)
21        , HH.slot _gridComponent 0 gridComponent state (\_ -> Nothing)
22      ] where
23        label = SProxy :: SProxy "dropdown"
24        input1 = { items: [ "Attendance", "Activity"], selection: Just "Attendance"
25                  , buttonLabel: "settings_applications|Plugin" }
26        input2 = {items: [ "931", "932", "933", "934", "935", "936", "937"], selection: Just "931"
27                  , buttonLabel: "group|Group" }
28        input3 = {items: [ "1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11", "12", "13", "14"]
29                  , selection: Just "1", buttonLabel: "format_list_numbered|Seminar" }
30  handleParent :: forall o . ParentAction -> H.HalogenM ParentState ParentAction ParentSlots o Aff Unit
31  handleParent = case _ of
32    InitializeParent -> pure upgradeElements
33    PluginHandle (Dropdown.SelectionChanged x y) -> H.modify_ _ {plugin = maybe "" (\a->a) y}
34    GroupHandle (Dropdown.SelectionChanged x y) -> H.modify_ _ {_aGroup = maybe "" (\a->a) y}
35    SeminarHandle (Dropdown.SelectionChanged x y) -> H.modify_ |
36      {_aSeminar = maybe 0 (\a->maybe 0 (\b->b) (parseInt a(toRadix 10))) y }
37    FillInput x -> H.modify_ _ {inputString = x}

```

Figure 3.12 Root component in the practical application

In the render function at lines 18-20, where the dropdown components are used, the last argument of the slot function specifies where the output of a child should be sent, more precisely to the Plugin/Group/SeminarHandle cases from the evaluation function handleParent.

The eval function lets the user define some basic behaviour for the component like initialization (defines if the component receives information from the parent when it's created or if it has to set up its state), receive (what information it will periodically receive from the parent), handleAction (defining methods that will alter its state during its life-cycle), handleQuery (define how it responds to queries made by a parent), and finalize (define what to do when it finishes its life-cycle).

3.4.2. Material Components Design

Material is a design system created by Google. The name Material is inspired by the physical world textures and how they reflect light and cast shadows. The building blocks that create the user interface are called Material Components, and they are available for Android, iOS, Flutter and web applications [19].

In our application, we used Material IO as a UI framework, integrating the Material Design scripts with Halogen components. In order to provide a special design for dynamic entities like a dropdown list, we need to instantiate the HTML tags from Halogen as Material Design Components.

The Material Component constructors take a CSS selector that references the Halogen tag and applies to it the specific behaviour of that material component. In order to instantiate the material components, we need to import in our Purescript application an external JavaScript function and call it after we initialize our Halogen components.

The code from figure 3.13 exports the function `upgradeElements`, which initializes the Halogen HTML tags as Material Design Components. The function is imported by a purescript module in the following way: `foreign import upgradeElements:: Unit`. The imported function `upgradeElements` can be further called by any Halogen component, for example, the root component from figure 3.12, at line 32.

```
1  const mdcTextField = require('@material/textfield');
2  const mdcMenu = require('@material/menu');
3  const mdcRipple = require('@material/ripple');
4  const MDCTextField = mdcTextField.MDCTextField;
5  const MDCMenu = mdcMenu.MDCMenu;
6  const MDCRipple = mdcRipple.MDCRipple;
7  exports.upgradeElements = function ()
8  {
9    for (const el of document.querySelectorAll('.mdc-text-field')) {
10     const textField = new MDCTextField(el);
11   }
12   for (const el of document.querySelectorAll('.mdc-menu')) {
13     const menu = new MDCMenu(el);
14     menu.open = true;
15   }
16   for (const el of document.querySelectorAll('.mdc-button')) {
17     const button = new MDCRipple(el);
18   }
19 }
```

Figure 3.13 JavaScript function that initializes all the tags with a certain class name

3.5. Microsoft Teams

3.5.1. Introduction

Microsoft Teams platform lets us integrate our web application with Teams and use the Teams platform capabilities. The app's existing API and data structures must be augmented with contextual information about Teams.

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <title>Document</title>
5   <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-3.4.1.min.js"></script>
6   <script src="https://statics.teams.cdn.office.net/sdk/v1.6.0/js/MicrosoftTeams.min.js"
7     ></script>
8   <script>
9     microsoftTeams.initialize();
10    microsoftTeams.settings.registerOnSaveHandler(function (saveEvent) {
11      microsoftTeams.settings.setSettings(
12        { entityId: "example"
13          , contentUrl: "https://de-nix.github.io/purescript-Halogen/"
14          , suggestedDisplayName: "example"
15          , websiteUrl: "https://de-nix.github.io/"
16          , removeUrl: "https://de-nix.github.io/purescript-Halogen/"
17        });
18      saveEvent.notifySuccess();
19    });
20    microsoftTeams.settings.setValidityState(true);
21  </script>
22  <script src="./node_modules/@webcomponents/webcomponentsjs/webcomponents-loader.js">
23    </script>
24  <link href="https://unpkg.com/material-components-web@latest/dist/material-components-
25    web.min.css" rel="stylesheet">
26  <!-- Fonts to support Material Design -->
27  <link rel="stylesheet" href="https://fonts.googleapis.com/css?family=Roboto:300,400
28    ,500,700&display=swap" />
29  <!-- Icons to support Material Design -->
30  <link rel="stylesheet" href="https://fonts.googleapis.com/icon?family=Material+Icons"
31    />
32  <link rel="stylesheet" href="index.css"/>
33 </head>
34 <body>
35   <script src="app.js"></script>
36 </body>
37 </html>
```

Figure 3.14. Entry point of the application initialized as a Microsoft Teams application

Lines 7-20 from figure 3.14 initialize our web application as a Microsoft Teams app providing context information (id of the application, URL of the page, name of the page). With the

application initialized, a new app can be created inside the Microsoft Teams platform using the App Studio tool. App Studio makes it easy to start integrating an app by streamlining the creation of the manifest and package of the application.

When integrating an app with App Studio, one can choose what capabilities it needs. Our application has a tab in which the web application is rendered and a bot that can communicate with students in order to create attendance.

3.5.2. Tabs

Tabs are Teams-aware web pages embedded in Microsoft Teams. Each tab is just a simple HTML `<iframe>` tag that points to a domain declared in the app manifest and can be added as a part of a channel inside a team, group chat, or personal app for an individual user.

Figure 3.15 represents the interface of our web application, rendered inside a Microsoft Teams tab named example. This tab is visible only in the Channel General from the gradixx team, which means that each channel of this team, representing a laboratory or a seminary group, could have its own tab that will manage the attendances and the activities for a particular set of students, members of that channel.

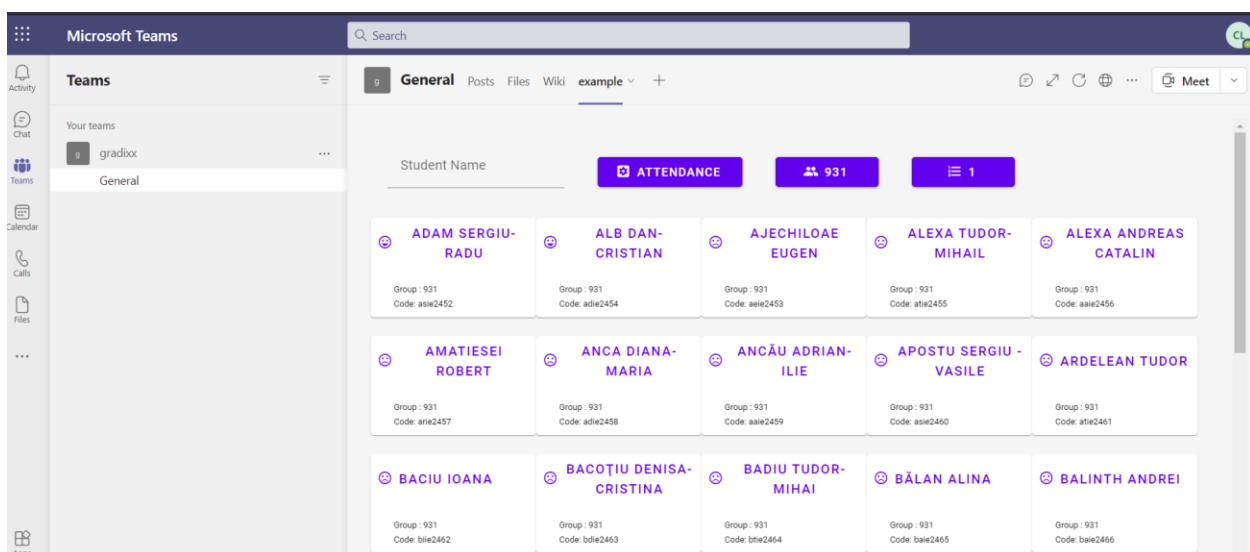


Figure 3.15. Microsoft Teams tab

3.5.3. Bots

“Bots are automated programs that respond to queries or give updates and notifications about details users find interesting or want to stay informed about. Bots allow users to interact with cloud services such as task management, scheduling, and polling in a Teams chat. Teams support bots in private chats and channels. Administrators can control whether the use of bots is allowed in a Microsoft 365 or Office 365 organization.” [20].

In our application, a Microsoft Teams bot is responsible for mediating the communication between the user and the Servant endpoint from figure 3.5. Each student can find the application called Gradix in the applications panel from Microsoft Teams and start a conversation with a bot. The bot will respond to predetermined messages suggested to the user when he tries to type anything in the chat.

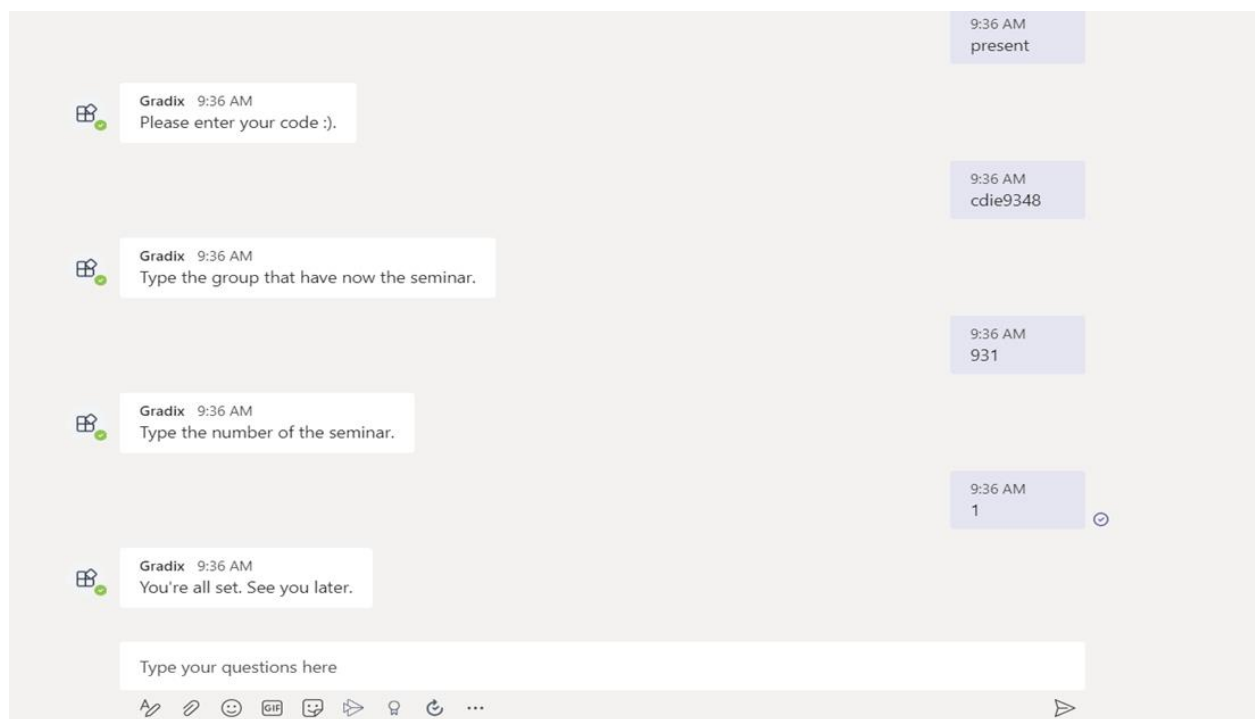


Figure 3.16. Microsoft Teams bot

In figure 3.16, a Microsoft Teams bot changes messages with the user and sends his responses to an endpoint server defined in the application manifest. In order to send a response back, the server endpoint requires to authenticate itself as a Microsoft Teams bot, and with the resulting authentication token, messages can be sent directly in the user chat.

```

1  sendRequest :: String -> Message -> IO ()
2  sendRequest text Message{mid=id, from=client, conversation=convObj, recipient=bot} = do
3    let response = Response text "message" bot client convObj id
4    client_secret = "4.Av925w5_tRKiW_j7jQ5_FzdSwn.lUHIf"
5    idClient = "a3ec8111-508d-494e-b2e0-2a7d61f7f102"
6    urlToken = "https://login.microsoftonline.com/botframework.com/oauth2/v2.0/token"
7    urlConversation = ("https://smba.trafficmanager.net/emea/v3/conversations/"
8                      ++ (conversationId convObj) ++ "/activities/" ++ id)
9    body = [ ("grant_type" , "client_credentials")
10            , ("client_id" , idClient)
11            , ("client_secret" , client_secret)
12            , ("scope" , "https://api.botframework.com/.default")]
13    initialReq <- parseRequest urlConversation
14    let request = initialReq {method="POST", requestBody=RequestBodyLBS $ encode response}
15    man <- newManager tlsManagerSettings
16    nakedRequest <- parseRequest urlToken
17    body <- responseBody <$> httpLbs (urlEncodedBody body nakedRequest) man
18    let token = fromMaybe "" (access_token <$> (decode body :: Maybe Token))
19    seqReq = addRequestHeader "Authorization" (packChars ("Bearer " ++ token)) request
20    httpLbs seqReq man
21    return ()

```

Figure 3.17. Microsoft Teams bot login and messaging

In figure 3.17, a get request is performed at "login.microsoftonline.com", requesting the token for the oauth2 authentication for our bot. The required fields for a successful authentication are the client_id which is an id that every bot receives when it is registered to the Microsoft Azure platform and a client_secret which is generated from a given password when setting up the bot security in Microsoft Azure.

The token received from the get request performed at line 17 is used to send back a message to the original conversation. The new request is performed at line 20 and given a conversation id, a message id, and an Authorization header with the previous token; it posts the message received as an argument by the function resulting from the login function (Figure 3.14).

3.5. Testing

For the practical application, we used integration testing in order to test both the business logic and the communication between the databases and the Servant server. The advantage of working with Servant lets us focus on testing the logic of the application because the parsing of the Json strings into the endpoint type values and the checking that the endpoint is called with the right number of parameters is done automatically.

```
1  main :: IO ()
2  main = hspec businessLogicSpec
3
4  withUserApp :: (Warp.Port -> IO ()) -> IO ()
5  withUserApp action = do
6    localAtt <- atomically $ newTVar Map.empty
7    Warp.testWithApplication
8      (pure (app' (State localAtt) (Connection "input.in")) action)
9
10 getStud :: Integer -> ClientM (Maybe Student)
11 postStud :: Student -> ClientM ()
12 putStud :: Student -> ClientM ()
13 deleteStud :: Integer -> ClientM ()
14 sortStud :: Integer -> String -> ClientM [Student]
15 filterStud :: String -> ClientM [Student]
16 getAtt :: Integer -> ClientM (Maybe Attendance)
17 postAtt :: Attendance -> ClientM ()
18 putAtt :: Attendance -> ClientM ()
19 deleteAtt :: Integer -> ClientM ()
20 getAtts :: Integer -> String -> ClientM [Attendance]
21 postAcc :: Attendance -> ClientM ()
22 mess :: Message -> ClientM ()
23 (  getStud    :<|> postStud :<|> putStud  :<|> deleteStud :<|> sortStud
24 :<|> filterStud :<|> getAtt   :<|> postAtt  :<|> putAtt    :<|> deleteAtt
25 :<|> getAtts   :<|> postAcc  :<|> mess) = client api
```

Figure 3.18. Testing Servant endpoints first part

In figure 3.18, the function `client api` from line 25 provides a calling function for each endpoint from the server api. Calling any of the functions from lines 10-22 will automatically create an api call for the corresponding endpoint from our api. The function defined at line 4 takes a function as an argument that will produce an IO computation if a Port number is given. The function

testWithApplication from line 7 takes as arguments the application of our server and the action function, which will be eventually fed with a port returning an IO computation.

```
1 businessLogicSpec :: Spec
2 businessLogicSpec = around withUserApp $ do
3   let createUser = client (Proxy :: Proxy API)
4   baseUrl <- runIO $ parseBaseUrl "http://localhost"
5   manager <- runIO $ newManager defaultManagerSettings
6   let clientEnv port = mkClientEnv manager (baseUrl { baseUrlPort = port })
7   describe "GET /student" $ do
8     it "should be able to get a student" $ \port -> do
9       result <- runClientM (getStud 190) (clientEnv port)
10      result `shouldBe` (Right $ Just $ Student
11        { _sId = 190, _sName = "VINCZI RICHARD"
12          , _sGroup = "937", _sCode = "vrie2641"})
13  describe "POST /student" $ do
14    it "should be able to add a student" $ \port -> do
15      result <- runClientM (getStud 1001) (clientEnv port)
16      result `shouldBe` (Right Nothing)
17      runClientM (postStud (Student 1001 "test" "test" "test")) (clientEnv port)
18      result <- runClientM (getStud 1001) (clientEnv port)
19      result `shouldBe` (Right $ Just $ (Student 1001 "test" "test" "test"))
```

Figure 3.19. Testing Servant endpoints second part

In figure 3.19, in lines 3-6, a client environment is built, which is needed for the function runClientM in order to run one of the functions defined in figure 3.16. The function `describe` starts a set of tests that are constructed with the function `it`. Each test case calls api endpoints that will modify the database and then a get call is made to extract the data and verify if the modification happened as expected.

Chapter 4. Conclusions

This dissertation covered the basics of functional programming and it introduced some more advanced topics like Effects, Monads, STM or MTL. The support application of this dissertation is purely functional, written in Haskell and Purescript, and integrated with Microsoft Teams.

Using the Microsoft Teams tab, a teacher can switch attendances and activities by clicking on the name of a student and selecting the configuration of the seminar – group, seminar number and the entity type we want to update. A student can chat with a Microsoft Teams bot and create an attendance after submitting data about the seminar and his code.

In the future, the development of this application could go in multiple directions. One idea is to extend our application by adding support for laboratories and courses. Another extension can be the support for seminar assignments together with a grading system or the support for multiple subjects. The application can also adopt the usage of DSLs and interpreters, which will make our application less dependent on the programming language used.

Bibliography

01. *Fedena*, <https://fedena.com/>. Accessed 19 05 2021.
02. *Creatix Campus*, <https://www.creatixcampus.com/attendance-management-system>. Accessed 19 05 2021.
03. *Strong Typing Without types*, 30 11 2020, <https://laptrinhx.com/strong-typing-without-types-3256031360/>. Accessed 3 3 2021.
04. *Static vs Dynamic Typing*, JBallin, 8 12 2017, <https://hackernoon.com/i-finally-understand-static-vs-dynamic-typing-and-you-will-too-ad0c2bd0acc7>. Accessed 20 2 2021.
05. *Persistent Data Structures*, augustl, 8 12 2019, https://augustl.com/blog/2019/you_have_to_know_about_persistent_data_structures/. Accessed 10 05 2021.
06. *Haskell Programming*, Christopher Allen, 2016, <https://haskellbook.com/>. Accessed 19 05 2021.
07. *Learn You A Haskell*, <http://learnyouahaskell.com/>. Accessed 12 06 2021.
08. *School of Haskell*, Brent Yorgey, 08 11 2013, <https://www.schoolofhaskell.com/school/starting-with-haskell/introduction-to-haskell/6-laziness>. Accessed 13 06 2021.
09. *Constraints Liberate, Liberties Constrain*, Runar Bjarnason, 4 1 2016, <https://www.youtube.com/watch?v=GqmsQeSzMdw>. Accessed 15 4 2021.
10. *Lambda Cube*, https://www.uio.no/studier/emner/matnat/ifi/nedlagte-emner/INF5160/h13/timeplan/bock_dependent_types.pdf. Accessed 12 06 2021.
11. *A Short Introduction to System F and Fw*, Pablo Nogueira, 22 02 2006, <https://babel.ls.fi.upm.es/~pablo/Papers/Notes/f-fw.pdf>. Accessed 12 06 2021.
12. *Modern Functional Programming: part 2*, John A De Goes, 27 09 2016, <https://degoes.net/articles/modern-fp-part-2>. Accessed 13 06 2021.
13. *Software Transactional Memory*, FPComplete, <https://www.fpcomplete.com/haskell/library/stm/>. Accessed 12 4 2021.
14. *Better Programming*, Marcel Moosbrugger, 04 02 2020, <https://betterprogramming.pub/monads-are-just-fancy-semicolons-ffe38401fd0e>. Accessed 13 06 2021.

15. *Optimistic locking and automatic retry*, Vladimir Khorikov, 18 09 2017,
<https://enterprisecraftsmanship.com/posts/optimistic-locking-automatic-retry/>. Accessed 13 06 2021.
16. *A Modern Architecture for FP*, JOHN A DE GOES, 28 12 2015, <https://degoes.net/articles/modern-fp>.
Accessed 7 4 2021.
17. *Servant Documentation*, 10 06 2021, https://docs.servant.dev/_/downloads/en/latest/pdf/. Accessed 13
06 2021.
18. *Halogen*, <https://purescript-halogen.github.io/purescript-halogen/>. Accessed 3 2 2021.
19. *Material Components*, <https://material.io/design/introduction#components>. Accessed 2 5 2021.
20. *Microsoft Teams*, <https://docs.microsoft.com/en-us/microsoftteams/deploy-apps-microsoft-teams-landing-page>. Accessed 19 05 2021.