# BACHELOR PAPER

Term paper submitted in partial fulfillment of the requirements for the degree of Bachelor of Science in Engineering at the University of Applied Sciences Technikum Wien - Degree Program Computer Science

# Composable Functional User Interfaces using the Elm Architecture

By: Silvia Bäs-Fischlmair

Student Number: 1510257002

Supervisor: DI Harald Steinlechner BSc.

Vienna, May 15, 2018

FH University of Applied Sciences
TECHNIKUM WIEN

# Declaration

“As author and creator of this work to hand, I confirm with my signature knowledge of the relevant copyright regulations governed by higher education acts (see Urheberrechtsgesetz /Austrian copyright law as amended as well as the Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I hereby declare that I completed the present work independently and that any ideas, whether written by others or by myself, have been fully sourced and referenced. I am aware of any consequences I may face on the part of the degree program director if there should be evidence of missing autonomy and independence or evidence of any intent to fraudulently achieve a pass mark for this work (see Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I further declare that up to this date I have not published the work to hand nor have I presented it to another examination board in the same or similar form. I affirm that the version submitted matches the version in the upload tool.“

Vienna, May 15, 2018 Signature

# Kurzfassung

Die Wiederverwendbarkeit von Code war schon immer ein wichtiges Thema für die Softwareentwicklung. Diese Arbeit konzentriert sich auf funktionale Konzepte, wie pure Funktionen und Composition, um Software mit einem hohen Grad an Wiederverwendbarkeit zu erstellen. Im ersten Teil gibt es einen Überblick über funktionale Programmierung, die ELM-Architektur und Plattformen, die diese Architektur verwenden. Der nächste Abschnitt beschreibt Composition detaillierter. Danach wird die Aardvark.Media Platform beschrieben. Im nächsten Teil ist die Umsetzung meiner Arbeit mit einer kurzen Diskussion über Vor- und Nachteile jedes Ansatzes. Der letzte Teil ist eine allgemeine Diskussion über die Arbeit.

**Schlagworte:** Aardvark.Media, Composition, Elm Architektur, F#, Funktionale Programmierung, Funktionales User Interface

# Abstract

The reusability of code has always been an important topic for software development. This work focus on functional concepts, such as pure functions and composition, to build Software with a high degree of reusability. The first part gives an overview of functional programming, the ELM-Architecture and platforms that use these Architecture. The next section describes Composition in more detail. Then the Aardvark.Media Platform is described. The next part is the implementation of my work with a short discussion about advantages and disadvantages of each approach. The last part is a general discussion about the work.

**Keywords:** Aardvark.Media, composition, Elm Architecture, F#, functional programming, functional user interface

# Contents


**1 Introduction** **1**

**2 Background information** **1**

2.1 Functional Programming . . . . . 2

2.1.1 Mathematical background - Lambda Calculus . . . . . 3

2.1.2 F# . . . . . 4

2.1.3 Mutable and immutable data structures in F# . . . . . 4

2.2 ELM-Architecture . . . . . 5

2.3 Functional GUIs based on the ELM Architecture . . . . . 5

2.3.1 ELM . . . . . 6

2.3.2 Fable-Elmish . . . . . 6

2.3.3 React / Flux . . . . . 6

2.3.4 Redux . . . . . 6

**3 Composition** **6**

3.1 Function Composition . . . . . 7

3.2 Type Composition . . . . . 8

3.3 Composition of Applications . . . . . 8

**4 Aardvark.Media** **8**

4.1 Incremental updates . . . . . 10

**5 Implementations** **12**

5.1 To Do List . . . . . 13

5.1.1 Model . . . . . 13

5.1.2 Update . . . . . 13

5.1.3 View . . . . . 14

5.1.4 Output . . . . . 16

5.1.5 Advantages and disadvantages . . . . . 16

5.2 3 dimensional Vector Composition . . . . . 17

5.2.1 Counter . . . . . 17

5.2.2 Composed application for the vector . . . . . 17

5.2.3 Output . . . . . 19

5.2.4 Advantages and disadvantages . . . . . 19

5.3 n dimensional Vector Composition with a List . . . . . 19
5.3.1 Composed application for the n dimensional vector . . . . . 19
5.3.2 Output . . . . . 21
5.3.3 Advantages and disadvantages . . . . . 22
5.4 Tree view . . . . . 22
5.4.1 Model . . . . . 23
5.4.2 Update . . . . . 24
5.4.3 View . . . . . 26
5.4.4 Output . . . . . 27
5.4.5 Advantages and disadvantages . . . . . 27
5.5 Generic Tree View . . . . . 28
5.5.1 Generic Tree View Application . . . . . 28
5.5.2 Example Application, that uses the Tree View . . . . . 30
5.5.3 Output . . . . . 32
5.5.4 Advantages and disadvantages . . . . . 32

**6 General discussion** **33**

**Bibliography** **34**

**List of Figures** **36**

**List of Tables** **37**

**List of Code** **38**

# 1 Introduction

One of the first things I heard while studding informatics was the DRY Principle - "Don't repeat yourself". While following these principle, a programmer must learn to abstract functionality into functions or methods that can be reused. Reusing code can save time, resources and reduce redundancy, as this code has already been created and tested.

A study about Open Source Software (OSS) from 2009, which examined the code reuse from the 1311 leading OSS projects in other OSS projects showed, that the reuse represented 316,000 staff years and tens of billions of dollars in development costs. This shows that reusability is not only important for programmers, it has a large economic aspect.

One way to build reusable software is using design patterns. Design pattern are well known design patterns for common design problems [1].

There are several design patterns like MVC (Model View Controller), MVVM (Model View ViewModel) or MVP (Model View Presenter) which try to separate the logic of programs, the data and the Graphical User Interface (GUI). One of the benefits of these patterns is that the code is reusable due to the abstraction of the layers [13].

Functional programming has several mechanism that help to build reusable code. These mechanism are pure functions, stateless programming and composition [2, Ch. 1].

This work focus on functional concepts to develop software with a high degree of reusability. The core mechanic to achieve this, is composition. Individual Software parts, that contains GUI and logic, can be build and tested and composed to a larger and more complex Software.

The rest of this work is structured as follows, Chapter 2 gives an short overview what functional programming is, about the ELM-Architecture and platforms that use these Architecture, Chapter 3 describes Composition in more detail, Chapter 4 describes the *Aardvark.Media* Platform that I used for my work, Chapter 5 describes the implementation of my work with a short discussion about advantages and disadvantages of every approach, Chapter 6 presents a General discussion about the work.

# 2 Background information

## 2.1 Functional Programming

As indicated in the previous section, the approach used in this work heavily uses functional programming principles in order to achieve reusable user interface implementations. In this section I give a short overview to functional programming starting from its theoretical foundations, required data structures and concepts in order to finally set the stage for the F# implementation based on the ELM architecture.

There is no clear definition of functional programming. [16, Ch. 1.1, 2]

The following definition comes from the FAQ of an academic mailing list, that compares functional programming with imperative programming:

"Functional programming is a style of programming that emphasizes the evaluation of expressions, rather than execution of commands. The expression in these languages are formed by using functions to combine basic values." [8]

For an imperative programming language this means, that a program is a sequence of commands. Typically these languages work with objects and the commands specify how the end result is obtained by creating and manipulating these objects.

For a functional language it means, that the program code is an expression, that specifies properties of the object that should be evaluated. It's not necessary to construct these objects and they cannot be used before they are created [16, Ch. 1.1].

Various functional languages exist and every language emphasizes a different set of aspects while giving less importance to others. There are some concepts that are shared from every language.

Some of these concepts are:

**First class functions** This means that it is possible to pass a function as an argument to another function, return it as the value from other functions or assign them to a variable [2, Ch. 1].

**Pure functions** A pure function is a function that does not have a side effect. A side effect is an action that modifies a state outside of the function, like changing a global variable or a printfn. By using a pure function the output only depends on the input, so evaluating a pure function with the same input will always result the same output [2, Ch. 1, 3].

**Recursion** Recursion is a function to call itself. With recursion it is possible to write small algorithm that only look at the input of the function [2, Ch. 1].

**Immutable variables** In an object orientated environment data are mutable. Objects can be modified directly or by calling a method for the object. At functional programming most data structures are immutable, so it is not possible to change them. When you use a

function with the data structure, it can only return a new data structure, but it can not modify the state of it [17, Ch. 1.3.3].

**Nonstrict evaluation (Lazy evaluation)** Strict evaluation means that a statement is immediately evaluated, when a variable is defined. At nonstrict or lazy evaluation the value of a function is not evaluated, when the variable is assigned, it is evaluated when the value is used [2, Ch. 1].

**Pattern matching** At functional programming variables are often encapsulated by types. With pattern matching it is possible to type check these and execute something different for each match [2, Ch. 1].

### 2.1.1 Mathematical background - Lambda Calculus

Functional programming has its roots in Alonzo Church's lambda calculus in 1932. He tried to formalize mathematical constructs with functions.

A mathematical function can be written like:

$$f(x) = x + 10 \tag{1}$$

Church assumed that functions could be used everywhere and that assigning a name to every function would be impractical. For this reason he introduced a new notation for functions without giving it a name:

$$(\lambda x.x + 10) \tag{2}$$

The function 2 is the same as function 1 but written in lambda calculus. The $\lambda$ is there that it stays for a function. The first x is for the variable name followed by a dot and the body of the function. When you want to calculate this function for a value x = 32 the result would be:

$$(\lambda x.x + 10)\ 32 = 32 + 10 = 42 \tag{3}$$

As in function 3, when there is an argument for a function, it's written directly after the function and for the calculation the value of the argument replaces the variable.

In lambda calculus any function can take a function as argument. With this, it's possible to write functions that takes a function and another value as parameter and call the operator with the value as arguments:

$$(\lambda op.\lambda x.(op\ x\ x)) \tag{4}$$

For writing functions with more arguments, it is necessary to use the lambda symbol multiple times like in function 4. In this example op should represent a function and x should represent the first and second argument of the op function. An example for this function would be:

$$(\lambda op.\lambda x.(op\ x\ x))(+)\ 21 = (\lambda x.((+)\ x\ x))\ 21 = (+)\ 21\ 21 = 42 \tag{5}$$

A function with multiple parameters first takes the first argument and returns a lambda expression, which is another function. In function 5 the operator was exchanged with the $(+)$ argument. In the next step the x parameter was exchanged with the value 21. This results in a function where the + operator can be used on the value 21. The calculation continues until there is no other function, that could be evaluated [16, Ch. 2.1].

### 2.1.2 F#

F# was released 2002 from Microsoft. It started as a Research Project by Don Syme and his colleagues with the goal of bringing functional programming to .NET. It was designed as a functional language with support for objects, but it resulted in a hybrid between functional language and object orientated language. It combines the runtime, libraries, interoperability and object model of the .NET Framework with succinct, expressive and compositional style of functional programming [16, Ch. 1.2.2]. It runs on most platforms including Linux and Smart Phones (via Mono and .NET Core) [23, p. 15].

### 2.1.3 Mutable and immutable data structures in F#

As mentioned before, mutable data structures are data structures that can be modified and immutable data structure can not be changed.

Code 1 shows an example for generating a map and adding new values to it. Since it is not possible to modify the data structure, a copy of the maps is generated (*map2* and *map3*) [24].

```
1 let map1 = Map.ofList [ (1, "one"); (2, "two") ]
2 let map2 = map1.Add(0, "zero")
3 let map3 = map2.Add(2, "twice")
4 Map.iter (fun key value -> printfn "key: %d value: %s" key value) map3
5 //Output
6 //key: 0 value: zero
7 //key: 1 value: one
8 //key: 2 value: twice
```

Code 1: Example for immutable data structures in F# (source: [10])

In F# it is possible to mixture object orientated notation with functional programming notation. *map1.add* is the object orientated way to add values to a map, which gets redirected to the

functional implementation. The functional implementation would be to use *Map.add key value map1* [10].

*Map.iter* is a functional notation, which is part of the module of *Map*. It iterates over every key value pair of the map and use the function on every key value pair. The signature for it is: *Map.iter : ('Key -> 'T -> unit) -> Map<'Key,'T> -> unit*. In this example it prints the keys and values pairs of every element of the map [11].

## 2.2 ELM-Architecture

The Elm Architecture was introduced to build web applications with the ELM Software. The concept of this Architecture is, that there is a basic pattern, which consists of Model, Update and View. The elm architecture goes back to the ELM language by Evan Czaplicki and his work on functional reactive programming [4].

The Model contains the current state of the application and represents your data. As it works with immutable data, the Model can't be changed.

The View describes, how the Model is displayed as HTML. It also contains the calls for the Update, like buttons, text fields or JavaScript.

The Update describes, how a Model gets transformed. It receives a message and a model, then it updates the model according to the message and returns a new model.

Figure 1 shows a Diagram of the Elm Architecture. It adds an additional part, the Runtime. At the Runtime, the program gets initialized with the model, view and update. It also handles all effects of an application or reacts to it. With Effects interactions with the outside world are meant, like loading and sending data via AJAX, writing to a local storage of the browser, interoperating with JavaScript or Web socket communication.

Once started, the program executes in a continuous loop. It taking actions from users, changing the state according to these actions and representing these changes in the view [3, 19].

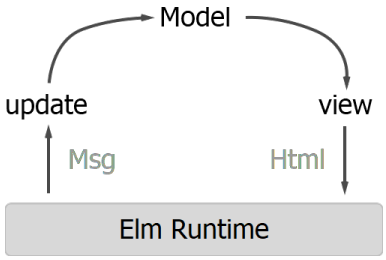


Figure 1: Diagram of the Elm Architecture (Source: [19])

## 2.3 Functional GUIs based on the ELM Architecture

In this section is a small overview about some languages or libraries that use the ELM Architecture, or an approach that is influenced by the ELM Architecture. The languages listed here

are only a small part of available languages and libraries.

### 2.3.1 ELM

Elm is a functional programming language for creating websites and web applications. The code compiles to JavaScript which is interpreted by a browser. It is easy to use the ELM Architecture with Elm, as it emerged from it [3].

### 2.3.2 Fable-Elmish

Elmish is a F# library, that use the Fable compiler to generate JavaScript out of F# code. Elmish implements the necessary abstractions to build applications that follow the Elm Architecture. The librarby itself does not model any view, so it needs other libraries with a DOM/renderer, like React/ReactNative or VirtualDOM [5].

### 2.3.3 React / Flux

Another application architecture, that use a similar approach is Flux. It was originally designed for Facebook applications and use the React library. React itself is a JavaScript library.

A Flux application consists of three parts: the dispatcher, the stores and the views (see Figure 2). The store contains the data and get updated from the dispatcher. After the store is updated, a new view is generated [6].

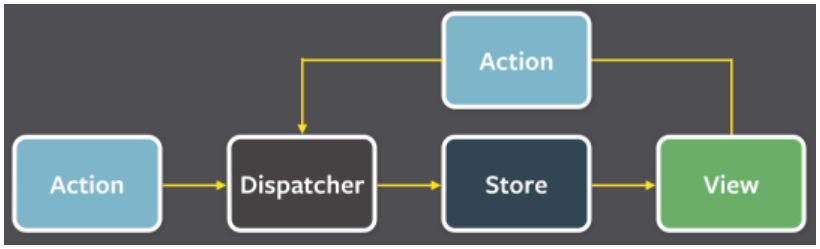


Figure 2: Structure and Data Flow model of Flux (Source: [6])

### 2.3.4 Redux

Redux is a predictable state container for JavaScript applications, which means that it is a data-flow architecture. It was inspired by Flux and Elm.

The dispatcher of the Flux-Model was replaced with a reducer. The reducer is a function, that takes a state and an action and returns a new state of the application. This is similar to the update function of the ELM-Application [18].

# 3 Composition

In the previous chapter the concepts of functional programming were described. In this chapter I describe how these concepts help to build reusable software.

Composition is a way to build something larger and complex from small pieces. It is possible to compare this with Lego, where you have bricks that can be stuck together to form a large piece.

There is an simple philosophy behind this for Lego:

- "All pieces are designed to be connected"
- "Connect two pieces together to get another 'piece' that can still be connected"
- "The pieces are reusable in many contexts" [22]

This works for programming too, when you want to build a larger and more complex program you can build it out of smaller pieces and compose them together to get a complex application.

## 3.1 Function Composition

Code 2 shows an example, where 3 functions are composed together and form a more complex function which executes the simpler functions one after another and pass the output from one function to the input of the other.

```
1 let add1 x = x + 1
2 let double x = x + x
3 let square x = x * x
4
5 let add1_double_square = add1 >> double >> square
6
7 let x = add1_double_square_5    //144
```

Code 2: Example for a composition in F# (source: [22])

Another way to compose functions is piping. It is similar to the piping in Unix / Linux, where the output is passed to the the next function (see Code 3) .

```
1 5 add1 |> double |> square  //144
```

Code 3: Example for a composition with piping in F# (source: [22])

While relying on functional concepts, a function is pure and has no side effect. While evaluating the function with the same input the output is always the same, even when you chain several functions together [22].

## 3.2 Type Composition

It is possible to compose types, so that new types can be build from smaller types. This is possible, as types are only data and have no methods attached.

There are two different types of type composition:

**Composing with "AND"** By composing different types with an "AND" every type is represented in the resulting type. Code 4 the FruitSalad is an example for this type of composition. It means, that a FruidSalad contains an Apple, a Banana and a Cherry. These types are called *record types*.

**Composing with "OR"** By this type of composition, the resulting type can only be one of the types. At Code 4 the Snack is an example for this type. A Snack can only be one an apple or a Banana or a Cherry. These types are called *discriminated unions* [22].

```
1 type FruidSalad = {
2     Apple : AppleVariety
3     Banana : BananaVariety
4     Cherry : CherryVariety
5     }
6
7 type Snack =
8     | Apple of AppleVariety
9     | Banana of BananaVariety
10    | Cherry of CherryVariety
```

Code 4: Example for a composition of Types (source: [22])

## 3.3 Composition of Applications

It is possible to compose complete applications to get more complex applications with more functionality. The applications are like black boxes where the same input result in the same output. For this the applications have to be pure with no side effects. Examples for this type of composition are showed in Chapter 5 [22].

# 4 Aardvark.Media

As indicated in Chapter 2, the approach used in this work heavily uses functional programming principles in order to achieve reusable user interface implementations. In this section we give

a short overview to functional programming starting from its theoretical foundations, required data structures and concepts in order to finally set the stage for the F# implementation based on the ELM architecture.

The previous chapter gave an overview of the concepts needed. This chapter focuses on the *Aardvark.Media* platform where the work was performed. *Aardvark.Media* is a collection of libraries, which perform an ELM implementation for user interfaces as well as 3D graphics [21].

Code 5 shows the listing of a the Model of a small Counter Application. The Model is an integer which represents the current number of the counter. The Message contains the actions, that can be performed by an update.

```
1 namespace counterModel
2
3 type Message =
4     | IncCounter
5     | DecCounter
6
7 [<DomainType>]
8 type Model =
9     {
10         counter : int
11     }
```

Code 5: Model of Counter

Code 6 shows the listing of the update function and the view. The update function has to handle every possible action that was described in the *Message* and generates an updated version of the *Model*. The view does not use the normal *Model*, it uses the *MModel* which is structurally equivalent to the original model. The reason for this is described in Section 4.1.

The view in this example contains the body of the html page that is formed with this application. The first square bracket of the containers like *body*, *div* or *h2* takes a list of Attributes for styling the container with inline css or JavaScript Event handlers, like the onClick event for the buttons. The second square bracket takes a list of *DomNodes*, which are the content of the containers.

In the *app* part there are the settings for the application, like how the update and view functions are called, the initial values for the models are set, and if necessary a thread pool is started.

In Figure 3 you see the resulting html application.

```
1 module counterApp
2 open counterModel
3
4 let update (model : Model) (msg : Message) =
5     match msg with
6     | IncCounter -> { model with counter = model.counter + 1 }
7     | DecCounter -> { model with counter = model.counter - 1 }
8
```

```
let view (model : MModel) =
    body [] [
        div [] [
            h2 [] [text "Counter:"]
            Incremental.text ( model.counter |> Mod.map string )
            br []
            button [onClick (fun _ -> IncCounter)] [text "Count up"]
            br []
            button [onClick (fun _ -> DecCounter)] [text "Count down"]
        ]
    ]

let threads (model : Model) =
    ThreadPool.empty

let app =
    {
        unpersist = Unpersist.instance
        threads = threads
        initial = { counter = 42 }
        update = update
        view = view
    }
```

Code 6: App of Counter

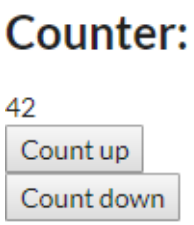


Figure 3: Resulting output of Code 6 of the Counter Application

## 4.1 Incremental updates

In the original Elm implementation, each message is handled via the update function, which in turn generates a new domain model. As a next step, the user interface needs to be updated to reflect changes of the domain model in the view. Most implementations compute the new user interface (represented as HTML DOM), and detect the modified parts in order to finally patch the existing DOM Tree. *Aardvark.Media* on the other hand uses fine-grained incremental changes (compute within the domain model). The concept behind incremental updates is, that when only some data or data points are changed, only these should update and not the complete application or view.

An example is, when there are changes of some values of a table, *Aardvark* only updates the div containers of the elements that are changed. Another example is, when there are data points in a graph, and the position of the camera is changed, *Aardvark* rewrite the view matrix in the GPU memory. *Aardvark* is build that it only perform incremental updates when it is necessary.

The update performance of incremental systems is independent of the total amount of data, it only depends on the rate of data that are changed. If a single point or a single div is changed, it will not use much CPU for the update. When every information is changed in every single frame, then there is no benefit in using incremental updates, it will be even slower as it still try to keep track of the changes. *Aardvark* applications are build by default, that they use incremental updates [9].

To perform these incremental changes, types are generated automatically, which provide the functionally that is needed:

- the change set of two different stages of an immutable object is computed (e.g. the old state and the new state after applying some operations)
- those changes are supplied to a mutable target object

For each immutable type *t*, which was marked as a *DomainType*, the *Aardvark.Compiler.DomainTypes* automatically creates a new type *mt* which has one operation:

*update (newImmutableValue : 't) : unit* [20]

In Code 7 the you can see the automatically generated File for the Model of Code 5.

```
1 [<AutoOpen>]
2 module Mutable =
3     type MModel(__initial : counterModel.Model) =
4         inherit obj()
5         let mutable __current :
              Aardvark.Base.Incremental.IModRef<counterModel.Model> =
              Aardvark.Base.Incremental.EqModRef<counterModel.Model>(__initial) :>
              Aardvark.Base.Incremental.IModRef<counterModel.Model>
6         let _counter = ResetMod.Create(__initial.counter)
7
8         member x.counter = _counter :> IMod<_>
9
10        member x.Current = __current :> IMod<_>
11        member x.Update(v : counterModel.Model) =
12            if not (System.Object.ReferenceEquals(__current.Value, v)) then
13                __current.Value <- v
14
15                ResetMod.Update(_counter,v.counter)
16
17        static member Create(__initial : counterModel.Model) : MModel =
              MModel(__initial)
18        static member Update(m : MModel, v : counterModel.Model) = m.Update(v)
```

```
19
20          override x.ToString() = __current.Value.ToString()
21          member x.AsString = sprintf "%A" __current.Value
22          interface IUpdatable<counterModel.Model> with
23              member x.Update v = x.Update v
```

Code 7: Model.g of Counter

The *Update* function compares the original model to the new model (v), check for equality and continue updating all included fields recursively.

With this translation it's possible to work on purely functional immutable data and feed the updated values to a mutable representation, which is updated automatically.

For some types, such as lists, *Aardvark* provide special implementations, which work more effective, when they are used in DomainTypes. Table 1 shows a summary of these types as immutable and mutable Types [20]. Details about the implementation are out of scope of this work. More informations regarding this are available on the *Aardvark.Media* page [14].

Table 1: Type for immutable and mutable Data for DomainTypes [20]

| | **Type** | **Incremental Type** |
|---|---|---|
| Persistent hash set | *hset<'a>* | *aset<'a>* |
| Persistent list | *plist<'a>* | *alist<'a>* |
| Persistent hash map | *hmap<'k, 'v>* | *amap<'k, 'v>* |
| Option type | *option<'a>* | *MOption<'a>* with active patterns: *MSome(v)* and *MNone* |
| Union types | *U = A1 \| ... \| An* | Mutable types, active patterns: *MA1 ... MAn* |
| All other types | *'t* | *IMod<'t>* |

Each incremental type is aware of its own change and the change of its dependencies. These types ensure, that when there are changes to the model, only relevant parts of the views are automatically updated.

There are builders for the adaptive datastructures *alist*, *amap* and *aset* and since these are adaptive context, it is possible to unpack mods inside these datastructures with a *let!* [15].

# 5 Implementations

In the next sections there are some examples of applications that were build with the *Aardvark.Media* platform. There are only code excerpts in this chapter. The complete code can be viewed on Github at https://github.com/Sayshea/aardvark.media/

## 5.1 To Do List

The first application is a To So List, where you can add or delete open task and mark them as done.

### 5.1.1 Model

The Model consists of *hmap* of *MyTasks*, where a GUID (Global Unique Identifies) is the Key of the map (see Code 8). A *MyTask* itself contains of the name of the Task, a Timestamp and a bool to differentiate if the task is already completed or not. The separation of Types is based on the Domain Driven Design (DDD, for more information see [25]).

There is an additional string in the Model (*TaskList*), the *activeCount* which counts the number of active Tasks and transform it to a Text. This is only one option to perform such a counter. Another way is to directly count the number from the *AMap* (see Line 22 of Code 10). Both versions are valid and give the correct number. For the first version you first have to update the model and then perform the count, to get the correct number of tasks.

```
1 type MyTask =
2     {
3         name : string
4         createDate : DateTime
5         completed : bool
6     }
7
8 [<DomainType>]
9 type TaskList =
10    {
11        tasks : hmap<string, MyTask>
12        activeCount : string // to have easier access for active todos (could be
              computed adaptively from tasks amap provided by mmodel)
13    }
```

Code 8: Model for the To Do List

### 5.1.2 Update

The update function (see Code 9) can add new tasks, delete tasks and add or remove them to the completed list. All manipulations are handled via the GUID, which is the key value for the *hmap*.

The *updateCompleted* function sets the completed value to true or false corresponding to the message, that was sent from the view. The message sent to the update function consists of a string as *message*, the *guid* and a *task*. The *message* is generated by a JavaScript call, so the resulting type is a string and the match is with a string and not a bool.

The *updateCount* function is called with a new model (*m'*) and calculates the the number of tasks that are not completed and returns a string.

```
let updateCompleted tasks guid task c =
    let updatedTask = function | Some t -> { t with completed = c} | None ->
        failwith ""
    HMap.update guid updatedTask tasks

let updateCount tasks =
    let i = tasks |> HMap.filter (fun _ t -> t.completed = false) |> HMap.count
    match i with
    | 1 -> "1 item left"
    | a -> (string a) + " items left"

let update (model : TaskList) (msg : TodoMessage) =
    let realUpdate model msg =
        match msg with
        | AddToTodoList s ->
            let task = { name = s; createDate = DateTime.Now; completed = false }
            let guid = System.Guid.NewGuid() |> string
            { model with tasks = HMap.add guid task model.tasks }
        | DeleteElement guid -> { model with tasks = HMap.remove guid model.tasks
            }
        | AddToCompletedList (message, g, task) ->
            match message with
            | "true" -> { model with tasks = updateCompleted model.tasks g task
                true }
            | _ -> { model with tasks = updateCompleted model.tasks g task false }
    let m' = realUpdate model msg
    { m' with activeCount = updateCount m'.tasks }
```

Code 9: Update function for the To Do List

### 5.1.3 View

The main part of the view function (see Code 10), is the *todoListGui*, the tasks are sorted by the creation date and then listed as a table.

```
let view (model : MTaskList) =
    let todoListGui =
        let sortedList = model.tasks |> AMap.toASet |> ASet.sortBy (fun (_,t) ->
            t.createDate)
        alist {
            for (guid,task) in sortedList do
                yield tr [][
                    yield td [][
                        yield inputCheckbox (fun s -> AddToCompletedList (s,
                            guid, task)) task.completed
```

```
9                       ]
10
11                      yield td [
12                          (match task.completed with
13                              | true -> attribute "style"
                                    "text-decoration:line-through"
14                              | _ -> attribute "style" "text-decoration:none")
15                      ][text task.name]
16
17                      yield td [] [button[onClick (fun _ -> DeleteElement guid)]
                            [text "Delete"]]
18                  ]
19          }
20
21      // most efficient adaptive variant (compared to explict storage of count in
            model)
22      let count = model.tasks |> AMap.filter (fun k v -> not v.completed) |>
            AMap.count
23
24      body [] [
25          h1 [] [text "Todos"]
26          input [
27              attribute "placeholder" "What needs to be done?"
28              onChangeResetInput(fun s -> AddToTodoList s)
29          ]
30          Incremental.table AttributeMap.empty todoListGui
31          br []
32          Incremental.text ( model.activeCount |> Mod.map string )
33          br []
34          Incremental.text (count |> Mod.map (function 0 -> "no items" | 1 -> "one
                item" | n -> sprintf "%d items" n))
35      ]
```

Code 10: View function for the To Do List

With the current state of *Aardvark.Media*, there were two problems which I encountered. The check boxes were not working properly and for the input field I wanted that the entry gets deleted, after someone hits enter or it looses the focus. I added the functionality to a Staging file (see Code 11). The JavaScript of the checkbox watches to *"event.target.checked"* and not the *value* which is the normal behaviour of the onchange function. Another feature for a checkbox is, that when a checkbox was already checked, it should be checked in the view again. At HTML5, whenever the checked attribute is there, it will count as a checked checkbox. For this reason I replace the attribute checked with the value when it is not checked. For the input field an additional event is sent to the client, where the value is set to nothing.

```
1 let onChangeCheckbox (cb : string -> 'msg) =
2     onEvent "onchange" ["event.target.checked"] (List.head >> cb)
3
4 let inputCheckbox (funct : string -> 'msg) (check : bool) =
```

```
5     input[
6         attribute "type" "checkbox"
7         (match check with
8             | true -> attribute "checked" "checked"
9             | _ -> attribute "value" "")
10        onChangeCheckbox (funct)
11    ]
12
13 let onChangeResetInput (cb : string -> 'msg) =
14    onEvent "onchange" ["event.target.value, event.target.value = ''"] (List.head
          >> Pickler.json.UnPickleOfString >> cb)
```

Code 11: Functions for additional functionality

### 5.1.4 Output

Figure 4 shows the output of the application.

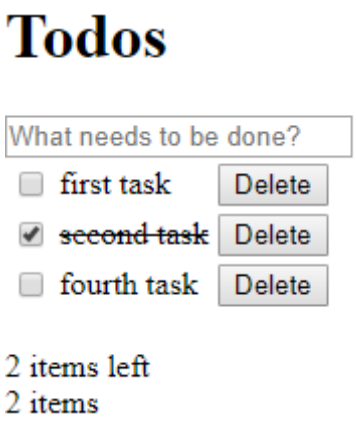


Figure 4: Output of the To do List

### 5.1.5 Advantages and disadvantages

Advantages

- The model uses a Domain Driven Design.

Disadvantages:

- It is a specialised application.
- Some tricks with JavaScript were needed to get the desired effect

## 5.2 3 dimensional Vector Composition

This example shows a 3 dimensional vector, which is build out of 3 counters. These counter are very similar to the counter in Code 5 and Code 6.

### 5.2.1 Counter

The Model uses a float (see Code 12) to store the value.

```
[<DomainType>]
type Model = { value : float }
```

Code 12: Model for one counter

Code 13 shows the update and view function of the counter. The update function contains a match for setting the value to a specific value to normalize the vector. The view function takes an additional attribute which is a label for the directions of the vector.

```
let update (m : Model) (a : Action) =
    match a with
        | Increment -> { m with value = m.value + 0.1 }
        | Decrement -> { m with value = m.value - 0.1 }
        | Set f -> { m with value = f }

let view (m : MModel) (direction : string) =
    tr[] [
        td[] [a [clazz "ui label circular Big"] [text direction]]
        td[] [a [clazz "ui label circular Big"] [Incremental.text (m.value |>
            Mod.map(fun x -> sprintf "%.2f" x))]]
        td[] [
            button [clazz "ui icon button"; onClick (fun _ -> Increment)] [text
                "+"]
            button [clazz "ui icon button"; onClick (fun _ -> Decrement)] [text
                "-"]
        ]
    ]
```

Code 13: Update and view function for the counter

### 5.2.2 Composed application for the vector

In Code 14 you can see, that the Vector Model uses the *Model* of the counter. The model use the Model of the counter, which is in the *NumericModel* namespace.

```
[<DomainType>]
type VectorModel = {
    x : NumericModel.Model
```

```
    y : NumericModel.Model
    z : NumericModel.Model
}
```

Code 14: Model for the composition of the Vector

Code 15 shows the update function for the 3 dimensional Vector application. For every direction an own *Action* is needed that references the *NumericControl.Action* of the counter, where the increment and decrement are dealt.

```
let update (m : VectorModel) (a : Action) =
    match a with
        | UpdateX a -> { m with x = NumericControl.update m.x a }
        | UpdateY a -> { m with y = NumericControl.update m.y a }
        | UpdateZ a -> { m with z = NumericControl.update m.z a }
        | Normalize ->
            let v = V3d(m.x.value,m.y.value,m.z.value)
            { m with
                x = NumericControl.update m.x (Set v.Normalized.X);
                y = NumericControl.update m.y (Set v.Normalized.Y);
                z = NumericControl.update m.z (Set v.Normalized.Z) }
        | Reset -> initialValues
```

Code 15: Update function for the application for the composition of the Vector

The *view* function (see Code 16 calls the *NumericControl.view* from the counter where the action is mapped to the update of the 3 dimensional Vector.

```
let view (m : MVectorModel) =
    require Html.semui (
        body[attribute "style" "margin:10"][
            table [] [
                NumericControl.view m.x "X:" |> UI.map UpdateX
                NumericControl.view m.y "Y:" |> UI.map UpdateY
                NumericControl.view m.z "Z:" |> UI.map UpdateZ
                tr[][
                    td[attribute "colspan" "2"][
                        div[clazz "ui buttons small"][
                            button [clazz "ui button"; onClick (fun _ ->
                                Normalize)] [text "Norm"]
                            button [clazz "ui button"; onClick (fun _ -> Reset)]
                                [text "Reset"]
                        ]
                    ]
                ]
            ]
        ]
    )
```

Code 16: View function for the application for the composition of the Vector

### 5.2.3 Output

Figure 5 shows the output of the Application.

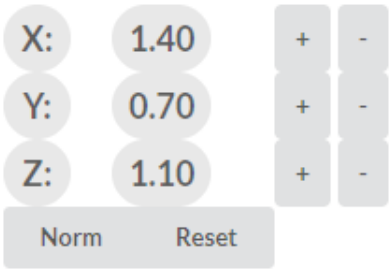


Figure 5: Output of the 3 dimensional Vector

### 5.2.4 Advantages and disadvantages

Advantages:

- It is easy to implement and understand
- the actions for one dimension are defined one time and can be called for every direction.
- It is easy possible to add additional things to a direction, like a name.
- It is possible to use types like *V3d* that give some additional features like Normalizing of a vector.

Disadvantages:

- There is still repetitive code for all the different directions
- For more dimensions there would be even more repetitive code

## 5.3 n dimensional Vector Composition with a List

This example use several counters to build an n dimensional vector. It is possible to add or remove dimensions, to reset the vector or to normalise it.

The counter that is used, is the same as in the example for the 3 dimensional vector. There is only a small change in the view function, as it does not take an additional string for the label.

### 5.3.1 Composed application for the n dimensional vector

In this example the *VectorModel* uses a *plist* of *NumericModels* for the *vectorList* (see Code17).

```
1 [<DomainType>]
2 type VectorModel = {
3     vectorList : plist<NumericModel.Model>
4     numDim : int
5 }
```

Code 17: Model for the composition of the n dimensional Vector

In comparison to the 3 dimensional Vector the *Action* only needs one *Update* with an additional parameter for the Index (see Code 18). The other additional actions here are for adding, deleting or resetting dimensions.

While performing the update the model still use the old one, because it is an immutable data type. For calculating the number of Dimensions the *Count* value is still the number from the old model, so it has to be corrected with plus or minus one depending whether a dimension should be added or removed. If the number of dimensions is not needed outside of the view functions, it is possible to evaluate it there and directly compute it (see Code 19 Line 7).

```
1 let update (m : VectorModel) (a : Action) =
2     match a with
3         | AddDimension ->
4             { m with
5                 vectorList = PList.append { value = 1.0 } m.vectorList
6                 numDim = m.vectorList.Count + 1
7             }
8         | DelDimension ->
9             let i = m.vectorList.Count - 1
10            let j =
11                match i with
12                | -1 -> 0
13                | a -> a
14            { m with
15                vectorList = PList.removeAt i m.vectorList
16                numDim = j
17            }
18        | Update'(i,a) ->
19            let v =
20                let v1 = PList.tryGet i m.vectorList
21                match v1 with
22                    | Some v' -> v'
23                    | None -> failwith ""
24            let v1 = NumericControl.update v a
25            { m with vectorList = PList.set i v1 m.vectorList }
26        | Normalize ->
27            let z = PList.toList m.vectorList
28            let z'= List.fold (fun norm v -> norm + (NumericControl.getValue v) *
                  (NumericControl.getValue v) ) 0.0  z
29            { m with vectorList = PList.map (fun vector -> NumericControl.update
                  vector (Set ( (NumericControl.getValue vector)/z'))) m.vectorList}
```

```
30          | Reset -> {m with vectorList = PList.map (fun vector ->
                NumericControl.update vector (Set 1.0)) m.vectorList }
31          | ResetAll ->
32              {m with
33                  vectorList = PList.empty
34                  numDim = 0
35              }
```

Code 18: Update function for the application for the composition of the n dimensional Vector

The *view* function (see Code 19) use an *alist*, as this is the mutable type of the *plist*. The individual directions are displayed by mapping over the *alist*.

```
1 let view (m : MVectorModel) =
2     let models =
3         m.vectorList |> AList.mapi (fun i vector ->
4             NumericControl.view vector |> UI.map (fun a -> Update'(i,a))
5         )
6
7     let countedDim = AList.count m.vectorList
8
9     require Html.semui (
10        div[][
11            h1 [] [text "n-dim Vectors"]
12            text "Add or remove dimensions: "
13            button [onClick(fun _ -> AddDimension)] [text "+"]
14            button [onClick(fun _ -> DelDimension)] [text "-"]
15
16            Incremental.table AttributeMap.empty models
17
18            button [onClick(fun _ -> Normalize)] [text "Normalize"]
19            button [onClick(fun _ -> Reset)] [text "Reset Values"]
20            button [onClick(fun _ -> ResetAll)] [text "Delete all Dimensions"]
21            br []
22            text "Number of dimensions (from model): "
23            Incremental.text (m.numDim |> Mod.map (fun x -> sprintf "%i" x))
24            br []
25            text "Number of dimensions (counted): "
26            Incremental.text (countedDim |> Mod.map (fun x -> sprintf "%i" x))
27        ]
28    )
```

Code 19: View function for the application for the composition of the n dimensional Vector

### 5.3.2 Output

Figure 6 shows a screen shot of the application.

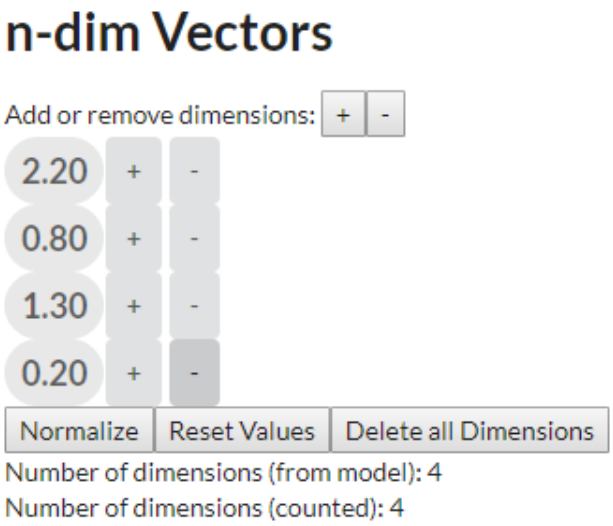


Figure 6: Output of the n dimensional Vector

### 5.3.3 Advantages and disadvantages

Advantages:

- There is no repetitive code.
- The number of dimensions does not matter.
- As you work with a *plist*, you can use the features of a *plist*, like *map*, *removeAt*, *set* and so on.

Disadvantages:

- As the directions of the vector are generated, it is harder to add specific additional informations like a name.
- You have to pay more attention to what you do, because when you update the model, you are still working with the old version of the data and this can lead to confusions, for example when you count the number of elements of a list. Another approach would be to first update the list and return a new model and count the number of elements in the new model and update the model again for the number of dimensions.

## 5.4 Tree view

This example is a Tree View. The application allows to add and remove nodes and leafs. The values for the new generated elements are randomly generated. The application implements multi select of nodes and leafs and a drag and drop functionality.

The tree view is based on the zipper concept [7]. For the every element a path is generated inside the view function and passed to the update function, when an elements get modified.

### 5.4.1 Model

The Model consists of a *TreeModel* which contains a Tree, a *plist* of paths for selected items, a bool to store, if the control key is pressed and an optional value for a path, if something is dragged (see Code 20). The tree itself consists of Nodes and Leafs. A Leaf consists of a Value, which here is a string for his name and an integer for the value and a property, where it is stored, if it is selected or not. The Node consists of a Value, which here is a string for the name and a string for a unit, a property, which contains variables if the node is selected and collapsed, and a *plist* for the children of the type tree.

```
[<DomainType>]
type NodeValue =
    {
        name : string
        unit : string
    }

[<DomainType>]
type Properties =
    {
        isExpanded : bool
        isSelected : bool
    }

[<DomainType>]
type Tree =
    | Node of NodeValue * Properties * plist<Tree>
    | Leaf of LeafValue * LeafProperties

[<AutoOpen;
    CompilationRepresentation(CompilationRepresentationFlags.ModuleSuffix)>]
module Tree =
    let node v p c = Node (v,p,c) //{ value = v; children = c; properties = p }
    let leaf v p = Leaf (v,p)//{ value = v; properties = p }

[<DomainType>]
type TreeModel =
    {
        data: Tree
        selected : plist<list<Index>>
        ctrlDown : bool
        drag : Option<list<Index>>
    }
```

Code 20: Shortened Model for the tree view

### 5.4.2 Update

Code 21 shows the function that is responsible for updates in the tree itself. In the first step it reverses the path, that the first element of the list, is the children from the root node and then it traverse recursively through the path until the list is empty. At this point it performs the function that is passed to the function on the leaf or node. Code 22 shows an example how such a function could look like.

```
let updateAt (p : list<Index>) (f : Tree -> Tree) (t : Tree) =
    let rec go (p : list<Index>) (t : Tree)  =
        match p with
            | [] -> f t
            | x::rest ->
                match t with
                    | Leaf (l, p) -> Leaf (l, p)
                    | Node(l,p,xs) ->
                        match PList.tryGet x xs with
                            | Some c -> Node(l,p, PList.set x (go rest c) xs)
                            | None   -> t
    go (List.rev p) t
```

Code 21: UpdateAt function for the update of the tree view

The *flipSelected* function (see Code 22) changes the value of the *isSelected* bool, to represent, if the node or leaf is selected or not. It calls the previously described *updateAt* function. The function that is passed to the updateAt function is a pattern matching function, which is a shorter version of the match expression (see [12] for detailed information). It has to differentiate if the element is a leaf or a node.

```
let flipSelected path t b =
    updateAt path (
        function
            | Leaf (l, p) -> Leaf (l, {p with isSelected = b })
            | Node (l, p, xs) -> Node (l, { p with
                                                isSelected = b
                                                isExpanded = p.isExpanded }, xs)
        ) t
```

Code 22: flipSelected functions for the update of the tree view

Code 23 shows a shortened version of the *update* function and some messages are described exemplary.

*AddNode* (and *AddLeaf*) gets the path information from the parent element, where the new element should be generated. The new element is generated with some random informations and appended to the *plist* of children with the *updateAt* function.

A key press is a JavaScript event, which is tracked in the body of the application (see Code 24). The event triggers a message for the update (*Keydown*). When the key press is the left or the right Ctrl key, this is stored in the *ctrlDown* value.

For the multi select functionality the *Selected* message needs to differentiate, if the *Ctrl* key is pressed or not. If the key is pressed it calls the *flipSelected* function and add the path to the *selected plist*. When the key is not pressed, it first iterates through the *oldSelected plist* and remove the selection from every element from the list and generate a new *selected plist* with the new path as element.

```
let rec update (m:TreeModel) (msg : Message) =
    match msg with
    | AddNode path ->
        let genUnit =
            let rnd = Random()
            let arr = [|"%"; "W"; "V"; "MHz"; "GB"|]
            arr.[rnd.Next(arr.Length)]
        let newNode = Tree.node { name = "Generated Node"; unit = genUnit }
            defaultProperties PList.empty
        { m with
            data =
                updateAt path (
                    function | Leaf _ -> failwith "You can't add anything to a
                        leaf!"
                             | Node (l, p, xs) -> Node (l, p, PList.append
                                 (newNode) xs)
                ) m.data
        }
    | AddLeaf path -> [...]
    | Selected path ->
        match m.ctrlDown with
        | false ->
            let oldSelected = m.selected
            { m with
                data = selectUpdate oldSelected (PList.ofList [path]) m.data
                selected = PList.ofList [path]
            }
        | true ->
            { m with
                selected = PList.append path m.selected
                data = flipSelected path m.data true
            }
    | ToggleExpaneded path -> [...]
    | DeleteItem path -> [...]
    | Keydown Keys.LeftCtrl -> { m with ctrlDown = true }
    | Keyup Keys.LeftCtrl -> { m with ctrlDown = false }
    | Keydown Keys.RightCtrl -> { m with ctrlDown = true }
    | Keyup Keys.RightCtrl -> { m with ctrlDown = false }
    | DragStop target -> [...]
    | DragStart source -> { m with drag = Some source }
    | _ -> m
```

Code 23: Shortened update function for the tree view

### 5.4.3 View

Code 24 shows the *view* function for the tree view. In the body of the application the *onKeyDown* and *onKeyUp* JavaScript events are added to register, if someone presses one of the Ctrl keys.

For viewing the tree, the *traverseTree* function (see Code 25) is called with an empty path.

```
1 let view (m: MTreeModel) =
2     require Html.semui (
3         body [ onKeyDown (fun usedKey -> Keydown usedKey); onKeyUp (fun usedKey
              -> Keyup usedKey) ][
4             h1 [][ text "TreeView" ]
5             Incremental.div (AttributeMap.ofList [clazz "ui list"]) (traverseTree
                  [] m.data)
6         ]
7     )
```

Code 24: View function for the tree view

The *traverseTree* function (see Code 25) is a recursive function that generates the view of the tree. As the code is inside an *alist* it is possible to transform the model from an *IMod<MTree>* to a *MTree* with a *let!*. After that it is possible to pattern match with the *MLeaf* and *MNode* to differentiate if the element is a leaf or a node.

The children variable (see Line 13 of Code 25) use the *collecti* function of the *alist*. The *collecti* function applies a function to every element of the *alist*, while giving additional informations for the *Index* of the element. After that, the results are concatenated to a combined *alist*. Inside the function the path of the children is generated by adding the Index in front of the current value of the path. With this new path the *traverseTree* function is called for the children.

```
1 let rec traverseTree path (model : IMod<MTree>) : alist<DomNode<Messages>> =
2     alist {
3         let! model = model
4         match model with
5         | MLeaf (l, p) ->
6             let! isSelected = p.isSelected
7             yield (leafViewText l p path)
8         | MNode (v, p, c) ->
9             let! isSelected = p.isSelected
10            let nodeAttributes = [...]
11            let leafAttributes = [...]
12
13            let children = AList.collecti (fun i v -> traverseTree (i::path) v) c
14
15            match path with
16            //the root Element shouldn't be visible, it's only here, to have
                  several nodes at the top level
17            | [] -> [...]
18            | _ ->
19                yield div [ clazz "item"] [
```

```
20                      Incremental.i nodeAttributes AList.empty
21                      div [ clazz "content" ][
22                          div [ clazz "header" ] [(nodeViewText v p path)]
23                          Incremental.div leafAttributes children
24                      ]
25                  ]
26      }
```

Code 25: Shortened traverseTree function for the tree view

### 5.4.4 Output

Figure 7 shows the output of the application.

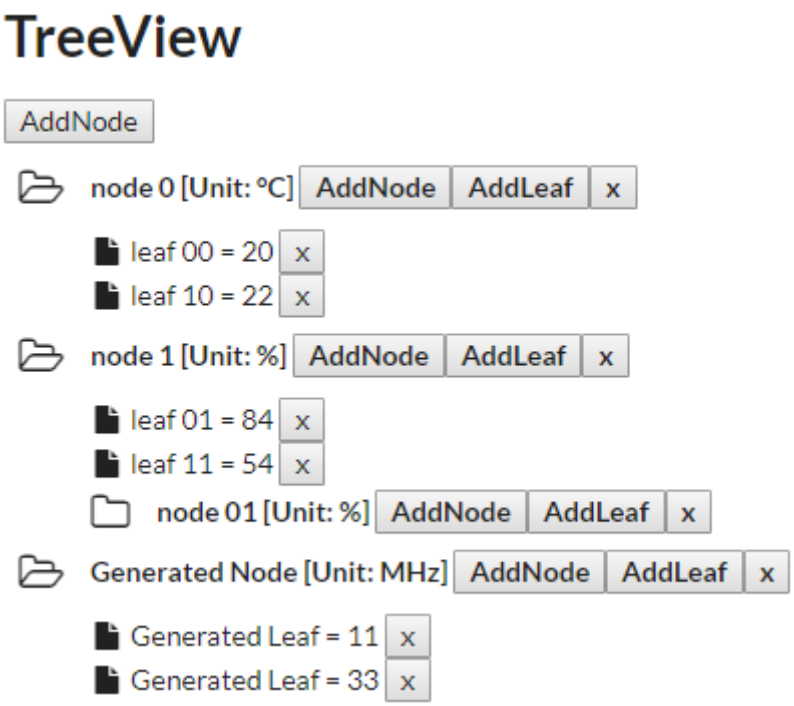


Figure 7: Output of the tree view

### 5.4.5 Advantages and disadvantages

Advantages

- It is easy possible to define different Node Types or Leaf Types.
- Is is highly configurable, how the tree view will look like and what is possible to do there.

Disadvantages:

- The implementation for a multi-select Drag and Drop can get complex, as when you move a parent node, and you have children of the node selected, you would loose the path to the children, when you first move the parent node.
- For every update on a Leaf or Node, it has to traverse the complete path of the element and updates every parent element in the model.
- For the actions you rely on JavaScript Events and there are no combinations like pressing the Control Key and pressing the Left Mouse Button, so you need a separate Event for pressing the Control and store this in a value and pressing the Left Mouse Button and then checking, if the value was set or not.

## 5.5 Generic Tree View

This example is a Generic Tree View. The intention behind this approach was, to build a reusable tree view, which has its own model, update and view functions and gets called from another application, like the vector examples before.

The Tree View uses a less complex Model than the previous Tree View (see Section 5.4). By default it can only toggle if Leafs or Nodes are shown below a Node but a user can define what is shown and if there are events like buttons.

### 5.5.1 Generic Tree View Application

In Code 26 is the Model of the Generic Tree View. The Type *InnerTree<'a>* consists of *InnerLeafs* and *InnerNodes*. It is the intern structure which is used by the application to build the Tree View. The *TModel* stores which Nodes are collapsed in a *hset* of paths. A path is build by the application (see Code 28 Line 8) from the right to the left by adding an Integer to a List.

```
1 [<DomainType>]
2 type TModel = {
3     collapsed : hset<list<int>>
4 }
5
6 type InnerTree<'a> = InnerLeaf of 'a | InnerNode of 'a * list<InnerTree<'a>>
```

Code 26: Model of the Generic Tree View

The *Message* in the application (see Code 27) is a generic type of type *'userMessage* which is specified from the application, that calls this Module. It takes a *UserMessage* or a *Toggle* from the internal Module. The *UserMessage* in the *update* currently only returns the model, as it is a simple Tree View. This can be further development, if it is necessary that user to be able to interact more with the model. With Toggle it is possible to collapse or expand the nodes.

```
type Message<'userMessage> =
    | Toggle of list<int>
    | UserMessage of 'userMessage

let update ( m : TModel) (msg : Message<'userMessage>) =
    match msg with
    | Toggle path ->
        match (HSet.contains path m.collapsed) with
        | true -> { m with collapsed = HSet.remove path m.collapsed }
        | false -> { m with collapsed = HSet.add path m.collapsed }
    | UserMessage userMsg -> m
```

Code 27: Update function of the Generic Tree View

The view function takes an *IMod* of an *InnerTree*, a function, how the elements are shown and an MTModel where it is stored, which elements are collapsed. It returns a *DomNode<Message<'userMessage»* which can be included in an application (see Code 31 Line 20).

The *traverseTree* function builds recursively the tree by calling itself by using *List.mapi* over every element of the List (see Code 28 Line 8). During this mapping, the path for the children is generated by adding the integer index in front of the current value of the path.

```
let rec traverseTree (path:List<int>) (model : InnerTree<'a>) (ui : 'a ->
    list<int> -> DomNode<'userMessage>) (m : MTModel) :
    alist<DomNode<Message<'userMessage>>> =
    alist {
        match model with
        | InnerLeaf a ->
            yield ((leafVText a path ui m) |> UI.map UserMessage)
        | InnerNode (a, c) ->
            let children : alist<DomNode<Message<'userMessage>>> =
                (List.mapi ( fun i x -> traverseTree (i::path) x ui m) c)
                |> AList.ofList
                |> AList.concat

            yield div [ clazz "item" ] [
                Incremental.i (nodeAttributes path m) AList.empty
                div [ clazz "content" ] [
                    div [ clazz "header" ] [(nodeVText a path ui m) |> UI.map
                        UserMessage]
                    Incremental.div (leafAttributes path m) children
                ]
            ]
    }

let view (tree : IMod<InnerTree<'a>>) (ui : 'a -> list<int> ->
    DomNode<'userMessage>) (m : MTModel) : DomNode<Message<'userMessage>> =
    let tree =
```

```
23          tree |> Mod.map (fun tree ->
24              traverseTree [] tree ui m
25          )
26
27      Incremental.div AttributeMap.empty <|
28              alist {
29              let! tree = tree
30              yield Incremental.div (AttributeMap.ofList [clazz "ui list"]) tree
31          }
```

Code 28: View function of the Generic Tree View

The functions *nodeVText* and *leafVText* (see Code 29) take the function from the user and display the informations. It is necessary to transform the *'userMessage* to a *Message<'userMessage>* as the update function can only handle these. This is performed by the *traverseTree* function (see Code 28 line 5 and 15). It maps the *DomNode<'userMessage>* to a *DomNode<UserMessage<'userMessage»* with the *UI.map* function.

```
1 let leafVText a path (ui : 'a -> list<int> -> DomNode<'userMessage>) (m :
      MTModel) =
2     let leaftext =
3         alist {
4             yield Incremental.span (AttributeMap.empty) <| alist {
5                 yield i [ clazz "file icon" ] []
6                 yield ui a path
7             }
8         }
9     Incremental.div (AttributeMap.ofList [ clazz "content" ]) leaftext
10
11 let nodeVText a path (ui : 'a -> list<int> -> DomNode<'userMessage>) (m :
      MTModel) =
12     let nodetext =
13         alist {
14             yield ui a path
15         }
16     Incremental.div (AttributeMap.ofList [clazz "content"]) nodetext
```

Code 29: leafVText and nodeVText function for the Generic Tree View

### 5.5.2 Example Application, that uses the Tree View

For the example application a simple model for a tree view was used. The model contains the *cTree* which represents the tree that should be displayed and a *treeModel*, which uses the *TModel* of the *TreeViewModel* (see Code 30).

```
1 type UserTree = UserLeaf of string | UserNode of string * list<UserTree>
2
3 [<DomainType>]
```

```
4 type Model = {
5     [<TreatAsValue>] // we want mod<tree> not mod<usertree>
6     cTree : UserTree
7
8     treeModel : TreeViewModel.TModel
9 }
```

Code 30: Model for the Application, that calls the Generic Tree View

The *Msg* type contains *Clicked* and *TreeMsg* events. The *Clicked* message is used to define the action that should be used when someone clicks on the button (see Code 31 Line 20). As this message is mapped to a *TreeMsg* (see Line 25) in the update function it won't match with the *Clicked* event, it will use the *TreeMsg (TreeVApp.Message.UserMessage (Clicked s))* action (see Line 8). This can be verified by checking the Console Output, as both use a *printfn* (see Figure 8).

```
inner: Node 10
inner: Leaf 010
inner: Leaf 110
```

Figure 8: Output of the console of the generic Tree View, when the buttons are clicked

For passing the the tree to the *TreeVApp* it must be converted to an *InnerTree*. This is done at the *createTreeViewTree* function. Inside the view the view of the *TreeVApp* is called (see Line 11). It takes the *InnerTree*, a function for handling what should be printed in the tree view and the *MTModel*.

```
1 type Msg =
2     | Clicked of string
3     | TreeMsg of TreeVApp.Message<Msg>
4
5 let update (m : Model) (msg : Msg) =
6     match msg with
7     | Clicked str -> printfn "%A" str; m
8     | TreeMsg (TreeVApp.Message.UserMessage (Clicked s)) -> printfn "inner: %s"
          s; m
9     | TreeMsg msg -> { m with treeModel = TreeVApp.update m.treeModel msg }
10
11 let rec createTreeViewTree (a : UserTree) : InnerTree<string> =
12     match a with
13         | UserNode(s,children) -> InnerNode(s, children |> List.map
              createTreeViewTree)
14         | UserLeaf(s) -> InnerLeaf s
15
16 let view (m: MModel) =
17     let innerTree =
18         m.cTree |> Mod.map createTreeViewTree
19
```

```
20      let tv = TreeVApp.view innerTree (fun s path -> button [onClick (fun _ ->
            Clicked s)] [text s]) m.treeModel
21
22      require Html.semui (
23          body [attribute "style" "margin:10"] [
24              h1 [][text "Generic TreeView"]
25              tv |> UI.map TreeMsg
26          ]
27      )
```

Code 31: Application, that calls the Generic Tree View

### 5.5.3 Output

Figure 9 shows an example of an generated Tree View. It is possible to collapse or expand the nodes, by clicking on the folder symbols and when you click on a button you get a console output with the name of the node or leaf.

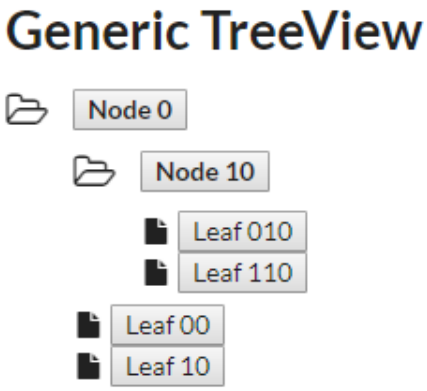


Figure 9: Output of the generic Tree View

### 5.5.4 Advantages and disadvantages

Advantages

- It is possible to pass every data which can be represented as tree and build a tree view with it.
- It is possible to customize the view by changing the function that is passed to the tree view.
- The Generic Tree View application is reusable and composable.

Disadvantages:

- It is necessary, that you know the model of the generic tree view application and that you generate a model from your data and pass them to the tree view application.
- As with the previous tree view example, all data are loaded, even when they are not shown. With a large dataset this can result in longer loading times.

# 6 General discussion

The purpose of this work was to use functional concepts to develop software with a high degree of reusability. It has been shown that composition is a viable way to write reusable software.

In the last section different graduations of composition were used. Functional composition was used in every program. It is a basic part of functional programming. More complex functions are build by chaining or piping simpler functions together.

Type composition was used for the models and the messages. Examples for discriminated unions are messages, while using these at the view function only one type of the message can match. Models are often record types, as you often want to combine different informations for your application. The Model of the Tree View use both types (see Code 20), as a tree consists of leafs and nodes which is represented by a discriminated union. Leafs and nodes consists of values and properties, which are record types.

Composition of applications was used for the 3 dimensional Vector, the n-dimensional Vector and the Generic Tree View. The vector examples are simpler examples as the composed application is a counter with limited functionality. The Generic Tree View is a more complex example as the illustration of a tree is general more complex and it uses generic types to display the data.

All examples that are included in this work are still relative simple applications, but all of them show that composition is a powerful tool to build complex and reusable applications in a functional language.

As long as the applications are pure, it is possible to create more complex applications with composition using the same mechanism described in this work for lower level applications. Future work could prove this by building even larger applications from the same components used here.

# Bibliography


[1] AMPATZOGLOU, A., A. KRITIKOS, G. KAKARONTZAS and I. STAMELOS: *An empirical investigation on the reusability of design patterns and software packages*. The Journal of Systems & Software, 2011, Vol.84(12), pp.2265-2283.

[2] BACKFIELD, J.: *Becoming functional*. O'Reilly, Sebastopol, Calif., 1. ed. ed.

[3] CZAPLICKI, E.: *An Introduction to Elm*, 2018. [Online] Available at: https://legacy.gitbook.com/read/book/evancz/an-introduction-to-elm [Accessed 13.4.2018].

[4] CZAPLICKI, E.: *Elm: Concurrent FRP for Functional GUIs*, 30.3.2012. [Online] Available at: http://elm-lang.org:1234/papers/concurrent-frp.pdf/ [Accessed 14.5.2018].

[5] ELMISH: *Elmish: Elm-like abstractions for F# applications targeting Fable.*. [Online] Available at: https://github.com/elmish/elmish [Accessed 9.5.2018].

[6] FACEBOOK INC.: *Flux - Application Architecture for building user interfaces*, 2014-2015. [Online] Available at: https://facebook.github.io/flux/ [Accessed 8.5.2018].

[7] HASKELL WIKI: *Zipper*, 6.3.2014. [Online] Available at: https://wiki.haskell.org/Zipper [Accessed 15.5.2018].

[8] HUTTON, G.: *Frequently Asked Questions for comp.lang.functional*, Nov 2002. [Online] Available at: http://www.cs.nott.ac.uk/~pszgmh/faq.html#functional-languages [Accessed 20.4.2018].

[9] MAIERHOFER, S.: *Incremental Rendering*, 28 Jan 2017. [Online] Available at: https://github.com/aardvark-platform/aardvark.docs/wiki/Incremental-Rendering [Accessed 17.4.2018].

[10] MICROSOFT DOCS: *Map.Add<'Key,'Value> Method (F#)*, 18.8.2017. [Online] Available at: https://msdn.microsoft.com/en-us/visualfsharpdocs/conceptual/map.add%5B'key%2C'value%5D-method-%5Bfsharp%5D [Accessed 14.5.2018].

[11] MICROSOFT DOCS: *Map.iter<'Key,'T> Function (F#)*, 18.8.2017. [Online] Available at: https://msdn.microsoft.com/en-us/visualfsharpdocs/conceptual/map.iter%5B'key%2C't%5D-function-%5Bfsharp%5D [Accessed 15.5.2018].

[12] MICROSOFT DOCS: *Match expressions*, 19.4.2018. [Online] Available at: https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/match-expressions [Accessed 15.5.2018].

[13] MOSKALA, M.: *MVC vs MVP vs MVVM vs MVI*, 22.11.2017. [Online] Available at: https://academy.realm.io/posts/mvc-vs-mvp-vs-mvvm-vs-mvi-mobilization-moskala/ [Accessed 15.5.2018].

[14] ORTNER, T.: *Aardvark.Media*, 2.2.2018. [Online] Available at: https://github.com/aardvark-platform/aardvark.docs/wiki/Aardvark.Media [Accessed 18.4.2018].

[15] ORTNER, T.: *Learning Aardvark.Media #3*, 2.2.2018. [Online] Available at: https://github.com/aardvark-platform/aardvark.docs/wiki/Learning-Aardvark.Media-%233 [Accessed 15.5.2018].

[16] PETRICEK, T. A. and J. C. SKEET: *Real-world functional programming : with examples in F# and C#*. Manning, [Place of publication not identified].

[17] PETRICEK, T. A. and J. C. SKEET: *Real-world functional programming : with examples in F# and C*. Manning, [Place of publication not identified].

[18] REDUX: *What is Redux?, When to use it? Here's the Beginners Guide to Redux.*. [Online] Available at: https://redux.js.org/ [Accessed 11.5.2018].

[19] REIMANN, D.: *The Elm Architecture*, 20. Dec 2016. [Online] Available at: https://dennisreimann.de/articles/elm-architecture-overview.html [Accessed 23.4.2018].

[20] VRVIS: *Automatic diffing for immutable types using Aardvark.Compiler.DomainTypes*. [Online] Available at: https://rawgit.com/wiki/aardvark-platform/aardvark.docs/docs/media/DomainTypeGeneration.html [Accessed 17.4.2018].

[21] VRVIS: *Aardvark Platform*, 20.3.2018. [Online] Available at: https://github.com/aardvark-platform/aardvark.docs/wiki [Accessed 13.4.2018].

[22] WLASCHIN, S.: *The Power of Composition - A review of why composition is a fundamental principle of functional programming*, 17. Jan 2018. [Online] Available at: https://fsharpforfunandprofit.com/composition/ [Accessed 30.4.2018].

[23] WLASCHIN, S.: *F# for fun and profit*, 2016. [Online] Available at: https://www.gitbook.com/book/swlaschin/fsharpforfunandprofit/ [Accessed 19.4.2018].

[24] WLASCHIN, S.: *Immutability*, 2016. [Online] Available at: https://fsharpforfunandprofit.com/posts/correctness-immutability/ [Accessed 14.5.2018].

[25] WLASCHIN, S.: *Domain Driven Design - Slides and video from my talk "Domain Modeling Made Functional with the F# Type System"*, 5. July 2017. [Online] Available at: https://fsharpforfunandprofit.com/ddd/ [Accessed 26.4.2018].

# List of Figures

Figure 1 Diagram of the Elm Architecture (Source: [19]) . . . . . 5
Figure 2 Structure and Data Flow model of Flux (Source: [6]) . . . . . 6
Figure 3 Resulting output of Code 6 of the Counter Application . . . . . 10
Figure 4 Output of the To do List . . . . . 16
Figure 5 Output of the 3 dimensional Vector . . . . . 19
Figure 6 Output of the n dimensional Vector . . . . . 22
Figure 7 Output of the tree view . . . . . 27
Figure 8 Output of the console of the generic Tree View, when the buttons are clicked . . 31
Figure 9 Output of the generic Tree View . . . . . 32

# List of Tables

Table 1 Type for immutable and mutable Data for DomainTypes [20] . . . . . 12

# List of Code


Code 1 Example for immutable data structures in F# (source: [10]) . . . . . 4
Code 2 Example for a composition in F# (source: [22]) . . . . . 7
Code 3 Example for a composition with piping in F# (source: [22]) . . . . . 7
Code 4 Example for a composition of Types (source: [22]) . . . . . 8
Code 5 Model of Counter . . . . . 9
Code 6 App of Counter . . . . . 9
Code 7 Model.g of Counter . . . . . 11
Code 8 Model for the To Do List . . . . . 13
Code 9 Update function for the To Do List . . . . . 14
Code 10 View function for the To Do List . . . . . 14
Code 11 Functions for additional functionality . . . . . 15
Code 12 Model for one counter . . . . . 17
Code 13 Update and view function for the counter . . . . . 17
Code 14 Model for the composition of the Vector . . . . . 17
Code 15 Update function for the application for the composition of the Vector . . . . . 18
Code 16 View function for the application for the composition of the Vector . . . . . 18
Code 17 Model for the composition of the n dimensional Vector . . . . . 20
Code 18 Update function for the application for the composition of the n dimensional Vector 20
Code 19 View function for the application for the composition of the n dimensional Vector 21
Code 20 Shortened Model for the tree view . . . . . 23
Code 21 UpdateAt function for the update of the tree view . . . . . 24
Code 22 flipSelected functions for the update of the tree view . . . . . 24
Code 23 Shortened update function for the tree view . . . . . 25
Code 24 View function for the tree view . . . . . 26
Code 25 Shortened traverseTree function for the tree view . . . . . 26
Code 26 Model of the Generic Tree View . . . . . 28
Code 27 Update function of the Generic Tree View . . . . . 29
Code 28 View function of the Generic Tree View . . . . . 29
Code 29 leafVText and nodeVText function for the Generic Tree View . . . . . 30
Code 30 Model for the Application, that calls the Generic Tree View . . . . . 30
Code 31 Application, that calls the Generic Tree View . . . . . 31