onderzoeksverslag Functioneel Paradigma

Jelmer van Vugt (618770) – 22-06-2022

Contents

[Afkortingen en begrippen 2](#_Toc105675225)

[1. Inleiding 3](#_Toc105675226)

[1.1 Onderzoeksvragen 3](#_Toc105675227)

[1.2 Challenge 3](#_Toc105675228)

[2. Methodologie 4](#_Toc105675229)

[2.1 Onderzoeksmethoden 4](#_Toc105675230)

[3. Onderzoeksresultaten 5](#_Toc105675231)

[3.1 Literatuuronderzoek 5](#_Toc105675232)

[3.1.1 Wat is imperatief en declaratief programmeren? 5](#_Toc105675233)

[3.1.2 Welke termen komen veel voor binnen het functionele paradigma? 6](#_Toc105675234)

[3.1.3 Wat is Haskell? 9](#_Toc105675235)

[3.2 Prototyping 11](#_Toc105675236)

[3.2.1 Hoe implementeer ik een Pythagoras fractal tree is Haskell? 11](#_Toc105675237)

[Fase 2 12](#_Toc105675238)

[3.3 Showroom 13](#_Toc105675239)

[3.3.1 Welke aspecten uit het functionele paradigma komen terug in de uitwerking van de fractal tree? 13](#_Toc105675240)

[4. Conclusie 16](#_Toc105675241)

[6. Literatuurlijst 17](#_Toc105675242)

# Afkortingen en begrippen

In dit hoofdstuk zijn enkele afkortingen en begrippen toegelicht die in dit onderzoeksverslag zijn gebruikt.

|  |  |
| --- | --- |
| Afkorting / Begrip | Toelichting |
|  |  |
| HOF | Higher-order function |
| FC | Functioneel programmeren |
| OOP | Object-georienteerd programmeren |

Tabel : Afkortingen en berippen

# 1. Inleiding

Dit document is het onderzoeksverslag dat deel uitmaakt van het beroepsproduct PO Paradigma uit het vak Algoritmes, Programmeertalen en Paradigma’s in het derde jaar Software Development zoals aangeboden op de Hogeschool van Arnhem en Nijmegen.

In dit onderzoeksverslag zijn de volgende hoofd- en deelvragen beantwoord.

## Onderzoeksvragen

Wat is functioneel programmeren?

* Wat is imperatief en declaratief programmeren?
* Welke termen komen veel voor binnen het functionele paradigma?
* Wat is Haskell?
* Hoe implementeer ik een Pythagoras fractal tree in Haskell?
* Welke aspecten uit het functionele paradigma komen terug in de uitwerking van de fractal tree?

## 1.2 Challenge

Als deel van het PO paradigma diende de student een zogeheten ‘challenge’ uit de kiezen en deze uit te werken in een functionele programmeertaal naar keuze. Ik heb ervoor gekozen om in de taal Haskell de Pythagoras fractal tree uit te werken. De uitwerking hiervan is tevens het prototype dat in deelvraag vier en vijf wordt behandeld.

Ik heb voor Haskell gekozen omdat ik hier voor het beroepsproduct al meerdere keren van heb gehoord maar nooit eerder ervaring mee heb opgedaan. Het was één van de eerste open-source ‘pure’ functioneel programmeertalen en lijkt mij daarom een mooie taal om mijn eerste ervaring met het functionele paradigma op te doen. De fractal tree lijkt mij hierbij een uitdagende maar haalbare opdracht om uit te werken.

# 2. Methodologie

In dit hoofdstuk worden de twee verschillende onderzoeksmethoden toegelicht die gebruikt zijn om de deelvragen te beantwoorden.

## 2.1 Onderzoeksmethoden

In dit kopje zijn de verschillende onderzoeksmethoden die binnen dit onderzoeksverslag zijn toegepast beknopt toegelicht.

|  |  |  |
| --- | --- | --- |
| Onderzoeksmethode | Toelichting | Toegepast binnen deelvraag |
|  |  |  |
| Literatuurstudie | Binnen een literatuurstudie wordt informatie vergaard uit leesbare artikelen. Deze informatie kan afkomstig zijn van websites, boeken, tijdschriften, kranten, et cetera. Uit deze gevonden artikelen kunnen selecte stukken worden gebruikt om op deze manier bij te dragen bij het beantwoorden van de deelvraag. | 1, 2, 3 |
| Prototyping | Binnen de prototyping onderzoeksmethode wordt een prototype ontwikkeld om een concept, design of probleem beter te begrijpen en/of de technische limieten en mogelijkheden de ontdekken. | 4 |
| Showroom | Binnen de showroom onderzoeksmethode wordt een gemaakt product – in dit geval een algoritme – aan de tand gevoelt en zijn bruikbaarheid, correctheid en functioneliteit ontleedt. | 5 |

Tabel : Onderzoeksmethoden

# 3. Onderzoeksresultaten

In dit hoofdstuk zijn de resultaten van elke deelvraag te vinden.

## 3.1 Literatuuronderzoek

Binnen deze paragraaf zijn de deelvragen behandeld die aan de hand van een literatuuronderzoek zijn uitgewerkt.

### 3.1.1 Wat is imperatief en declaratief programmeren?

Programmeertalen binnen het functionele – en het objectgeoriënteerde paradigma hebben in feite beide hetzelfde doel, namelijk: het bouwen van software. Ondanks dat beide paradigma’s hetzelfde einddoel hebben zit er veel verschil in hoe ze dit doel bereiken.

Het grootste verschil tussen de twee paradigma’s is dat objectgeoriënteerde talen imperatief zijn en functionele talen declaratief. Imperatieve programmeertalen beschrijven *hoe* een programma werkt en declaratieve *wat* een programma moet doen. *(Golliwitzer, 2020)* Om hier een beter beeld bij te illustreren pakken we voor het gemak het bouwen van een huis. Imperatieve programma’s zouden de volgende structuur hebben: bouw de fundering, voeg het skelet toe, installeer de utiliteiten, bouw de muren en werk het tot slot mooi af. Een aanroep om een huis te bouwen in een declaratief programma zou er als het volgende uitzien: Ik wil een huis met een open haard en drie verdiepingen, het maakt me niet uit hoe je dit doet. Dit wordt ook wel abstracte code genoemd.   
Talen die gebruik maken van het imperatieve paradigma worden ook wel procedurele talen genoemd. Voorbeelden hiervan zijn Java en C++. Talen die gebruik maken van het declaratieve paradigma worden ook wel functionele talen genoemd. Voorbeelden hiervan zijn Haskell en Lisp.

### 3.1.2 Welke termen komen veel voor binnen het functionele paradigma?

Onder dit kopje zijn een aantal veel voorkomende termen binnen het functionele paradigma onder elkaar gezet en toegelicht.

#### 3.1.2.1 Purity

In sommige functionele programmeertalen is het mogelijk voor functies om naast de returnwaarde te berekenen en terug te geven ook overige acties te ondernemen. Deze overige acties worden ook wel *side-effects* genoemd. Deze zijn side-effects genoemd omdat deze onverwachte acties mogelijk ongewenst het gedrag van andere functies kunnen beïnvloeden.

Om dit door side-effect verzoorzaakte mogelijke ongewenste gedrag te voorkomen is het concept ‘pure functions’ bedacht. Functies die worden beschouwd als ‘puur’ moeten voldoen aan de onderstaande eisen: *(Brassuer, 2014)*

* De functie moet altijd dezelfde waarde teruggeven gegeven dezelfde inputwaarden.
* De waardie die de functie mag alleen berekend worden door middel van variablen binnen zijn eigen scope. De lokale variabelen en eventuele parameters.
* De functie mag geen variabelen en/of objecten buiten zijn eigen scope aanpassen.

Omdat pure functies altijd dezelfde output moeten geven met dezelfde input zijn deze erg gemakkelijk te unit-testen. Ook hebben pure functies omdat deze geen interactie hebben met de buitenwereld amper impact op concurrency, wat erg voordelig is.

##### Lazy evaluation & Memoization

Lazy evaluation en memoization is een concept dat niet enkel bij talen binnen het functionale paradigma voorkomt. Omdat functionele talen echter grotendeels werken met pure functies simplificeert dit het wel.

Binnen het objectgeoriënteerde talen wordt vaak gebruik gemaakt van het concept dat ‘eager evaluation’ heet. Een compiler die werkt volgens het eager evaluation principe evalueert een variabele direct nadat het een tegenkomt; ook in de gevallen waar dit nog niet nodig is of compleet onnodig is. Tevens kan het voorkomen dat dezelfde intensieve evaluaties meerdere keren herhaalt worden, wat resources op een systeem kost en ten koste gaat van de performance. *(Bencevic, 2018)*

Functionele talen die werken met een combinatie van lazy evaluation en memoization trachten zoveel mogelijk onnodige evaluaties te voorkomen. Omdat pure functies gegeven dezelfde argumenten altijd dezelfde uitkomst hebben kunnen de uitkomsten van bepaalde – ervan uit gaande dat deze dezelfde parameters ontvangen – functies worden hergebruikt in plaats van opnieuw berekent te worden. Wat de performance enorm bevordert.   
Een voordeel dat lazy evaluation ook met zich meebrengt is de mogelijkheid om oneindig grote lijsten te kunnen gebruiken. Dit is mogelijk omdat een waarde in deze lijst pas wordt berekent wanneer deze nodig is.

##### Immutable data

Binnen veel functionele programmeertalen wordt er gebruik gemaakt van ‘immutable data’. Door veelal gebruik te maken van datastructuren die niet te wijzigen zijn is het schrijven van pure functies een stuk makkelijker. Ook brengt dit het voordeel met zich mee dat concurrency problemen veel minder voorkomend zijn omdat verschillende threads niet gelijktijdig dezelfde data kunnen gebruiken.

##### Monads

Ondanks dat pure functies de standaard zijn binnen het functionele paradigma kan het soms voorkomen dat een functie een actie moet verrichten die normaalgesproken voor een side-effect zou zorgen. Omdat side-effects zoveel mogelijk vermeden dienen te worden is hier een concept op bedacht, namelijk: monads.

Een monad fungeert als het ware als een container die door abstractie net doet alsof deze (mogelijke) side-effects niet bestaan. Denk hierbij aan bijvoorbeeld een I/O operatie. Doordat deze container een functie die (in sommige gevallen) een side-effect produceert omvat kan deze functie puur blijven. Dit stelt de ontwikkelaar in staat zich te focussen op de waardes die daadwerkelijk relevant zijn. *(Jitani, 2021).* Ook brengt de monad een set operaties met zich mee die uitgevoerd kunnen worden op deze container. Hierbij kan gedacht worden aan: map, return, join en bind.

#### 3.1.2.2 Higher order functions

Higher-order functions (HOFs) zijn functies die een functie gebruiken als een argument of een functie retourneren als return waarde. Een HOF kan dit ook beide doen. Een voorbeeld hiervan is een functie die zowel een lijst als een functie krijgt als argumenten. Vervolgens voert de functie op elk item binnen de lijst de meegegeven functie uit en geeft de nieuwe lijst terug.

Een van de voordelen die HOFs met zich meebrengen is het verminderen van repetitie binnen de code. Neem als voorbeeld een for lus. Deze bestaan altijd, exclusief de implementatie binnen de lus, uit dezelfde structuur. Het herhalen van deze lussen kan opgevangen worden met een HOF. Dit concept staat ook wel bekend als polymorfie voor functies.

#### 3.1.2.2 Currying

Naarmate functies een groter aantal parameters ontvangen worden deze al snel onleesbaar. Om dit probleem op te lossen is het zogeheten concept ‘currying’ verzonnen.

Currying is het proces waarin één functie die meerdere argumenten ontvangt wordt opgebroken in meerdere individuele functies die elk maar één parameter ontvangen en elkaar in serie aanroepen. Dit proces wordt net zolang herhaalt tot alle parameters zijn gebruikt. Elke functie berekent een waarde gebaseerd op zijn argument en geeft deze terug aan de functie boven hem in de hiërarchie tot het uiteindelijke resultaat wordt teruggegeven door de eerste functie in de serie. *(Haskell Wiki, 2020)*

Door één grote functie op te splitsen in verschillende sub-functies wordt de code makkelijker om te refactoren. Ook maakt dit de code makkelijker te lezen én te begrijpen voor de ontwikkelaar(s). Currying gaat erg goed samen met ander concepten binnen het functionele paradigma die leunen op dit concept. Een voorbeeld hiervan is ‘partial application’. In een notendop is partial application een concept waarin een ge-curryde methode aangeroepen kan worden zonder dat deze hoeft te voldoen aan alle parameters. Vervolgens kan het programma later de overige argumenten meegeven waarna de functie wordt uitgevoerd. *(Gupta, 2018)*

### 3.1.3 Wat is Haskell?

Haskell is een pure functionele programmeertaal. Het is en ‘statically typed’ en ontworpen om een zo breed mogelijk spectrum van applicaties te ondersteunen. Programma’s in Haskell worden altijd geschreven als wiskundige functies die geen ‘side-effects’ hebben. Door de puur functionele aard van Haskell wordt het veelal beschouwd als een veilige en betrouwbare taal, wat een van de grootste pluspunten van de taal is.

#### Beknopte geschiedenis

De eerste versie van Haskell is ontworpen in 1990 door de wiskundige Haskell Brooks Curry, wie wordt beschouwd als één van de grondleggers van functionele progammeertalen. *(Ionos, 2020)* Haskell is gebaseerd op Lambda calculus, wat een formele manier is om functies te bespreken. Dit is dan ook waarom het logo van Haskell grotendeels bestaat uit de Griekse letter Lambda. Haskell is van origine ontworpen om te werken met grootschalige industriële productieapplicaties en te gebruiken bij academisch onderzoek. Hedendaags wordt het echter voor ontzettend veel doeleinden buiten dit veld ingezet. Een aantal voorbeelden hiervan zijn ruimtevaart, social media, hardware design en nog veel meer.

#### Features

In de onderstaande tabel zijn de primaire features en/ of kenmerken van Haskell benoemd en is hier een kleine toelichting bij gegeven.

|  |  |
| --- | --- |
| Feature | Toelichting |
|  |  |
| Memory management | In talen zoals C moet de programmeur handmatig aan de hand van functies zoals *malloc* of *calloc* geheugen alloceren en na afloop weer vrijgeven.  Haskell neem deze verantwoordelijkheid bij de programmeur weg en alloceert automatisch geheugen, Initialiseert dit en geeft het aan de hand van een garbage collector ook weer vrij. |
| Leesbaarheid | Eén van de grote voordelen van functionele programmeertalen zijn hun leesbaarheid ten opzichte van hun imperatieve concurrenten. Dit maakt het vaker ook makkelijker te onderhouden. Het nadeel dat hier wel bij komt kijken is dat de implementatie een stuk lastiger is. |
| Herbruikbaarheid van code | Haskell ondersteunt polymorfisme binnen functies, wat de herbruikbaarheid hiervan ontzettend verhoogt. Hier is meer over verteld in de deelvraag [*“Welke termen komen veel voor binnen het functionele paradigma? – Higher order functions”*](#_3.1.2.2_Higher_order)*.* |
| Puur functioneel | Een stijl van programmeren waarbij in het programma alle computaties worden behandeld als een wiskundige functies. Binnen pure functionele programmeertalen hebben deze functies geen side-effects. |
| Statically typed | Een statically typed programmeertaal is een taal waar de typen van variabelen al bekend zijn gedurende het compileren. Dit helpt programmeurs om al in een eerder stadium (mogelijke) errors te identificeren. |
| Lazy evaluation | Zie[: *“Welke termen komen veel voor binnen het functionele paradigma? – Lazy evaluation & Memoization”.*](#_Lazy_evaluation_&) |
| Type inference | Type inference gebeurt wanneer een programma automatisch de datatypen binnen een expressie afleidt in compile-time zonder dat de programmeur deze expliciet hoeft te definiëren. |
| Multithreading | Haskell is een multithreaded taal. Dit houdt in dat het de mogelijkheid biedt om verschillende threads parallel aan elkaar verschillende gedeelten van het programma uit te laten voeren. Wat de performance van dit programma bevordert. |

Tabel : Features Haskell

## 3.2 Prototyping

Binnen deze paragraaf zijn de deelvragen behandeld die aan de hand van een prototyping onderzoek zijn uitgewerkt.

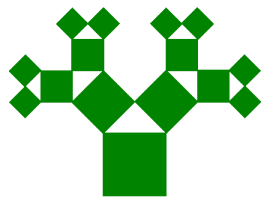
### 3.2.1 Hoe implementeer ik een Pythagoras fractal tree is Haskell?

Het implementeren van de Pythagoras’ fractal tree in Haskell bestaat uit drie fasen. In de eerste fase is de logica achter de fractal tree ontleedt, dit zodat deze kennis kon worden toegepast tijdens het implementeren van het algoritme. In de tweede fase is een keuze gemaakt voor een toolstack die het mogelijk maakt de fractal tree te implementeren. In de derde fase is het algoritme geschreven, deze is in de volgende deelvraag behandeld.

#### Fase 1

De Pythagoras boom is opgebouwd uit verschillende vierkantjes (fractals) die allemaal van één root fractal vandaan stammen. Bij elke nieuwe iteratie van de boom stammen er twee nieuwe ‘child’ fractals uit de onderliggende ‘parent’, deze moeten voldoen aan de volgende regels:

* Het formaat van de fractals wordt bepaald door de volgende formule:
* De linker fractal wordt 45 graden naar links geroteerd, de rechter 45 graden naar rechts.
* De y positie van beide fractals kan worden berekend door de volgende formule:
* De x positie van de linker fractal kan berekent worden door de volgende formule:
* De x positie van de rechter fractal kan worden berekend door de volgende formule:

In feite kan deze fractal tree tot in de oneindigheid door blijven itereren. Gebruikelijk is echter om deze na tien iteraties te stoppen. In de afbeelding hiernaast is een Pythagoras’ fractal tree afgebeeld na vier iteraties.

Figuur : Fractal Tree

### Fase 2

Voor het implementeren van de Pythagoras fractal tree in Haskell is een Cabal project met de Gloss library gebruikt.

Het is in Haskell mogelijk om zonder externe libraries illustraties op het scherm te tekenen. Dit kost echter relatief heel veel moeite om voor elkaar te krijgen en leek mij niet waar de focus van de opdracht op moest liggen. Om deze reden is gekozen om gebruik te maken van de externe library Gloss om de illustratie uit te werken.

Om Gloss te kunnen implementeren in het Haskell project is de package manager Cabal gebruikt. Deze dient ook gebruikt te worden om het project te executeren.

## 3.3 Showroom

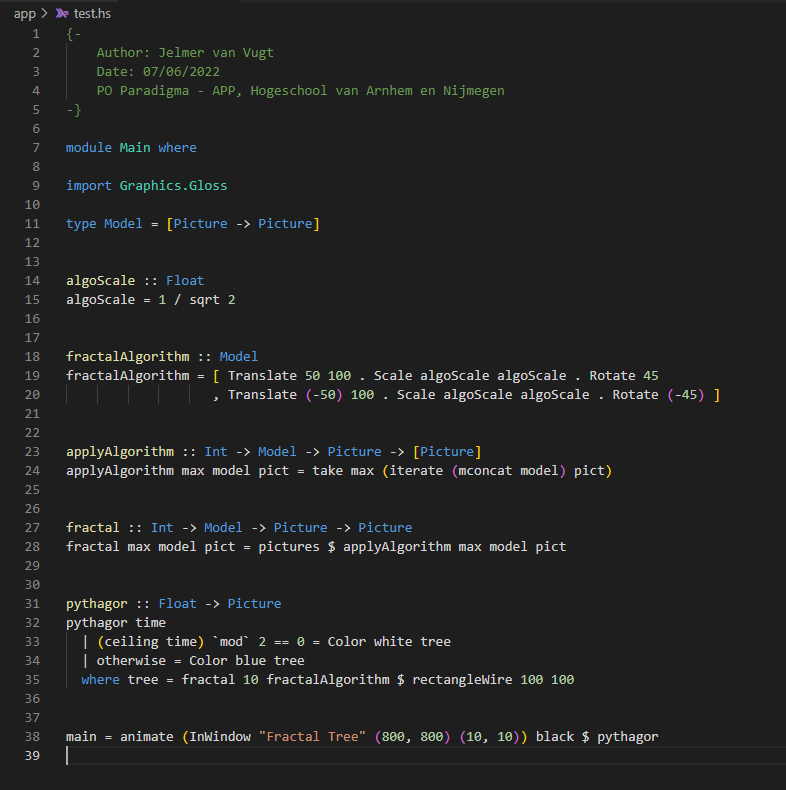
Binnen deze paragraaf zijn de deelvragen behandeld die aan de hand van een showroom onderzoek zijn uitgewerkt.

### 3.3.1 Welke aspecten uit het functionele paradigma komen terug in de uitwerking van de fractal tree?

Nadat in fase 1 duidelijk was geworden hoe de achterliggende logica van een Pythagoras fractal tree in elkaar zit en in fase 2 een toolstack is gekozen kon begonnen worden aan het programmeren. Ondanks dat ik een goed idee had over hoe een fractal tree in elkaar stak bleek de implementatie nog steeds erg lastig. Dit kwam vooral omdat ik nog geen eerdere ervaring had met Haskell en het functionele paradigma en ik hier ontzettend erg aan moest wennen. Toch is het uiteindelijk gelukt.

#### 3.3.1.1 Uitwerking

In onderstaand figuur is de uitwerking van de fractal tree in Haskell te zien. In het volgende kopje waar de functionele aspecten in deze oplossing zijn toegelicht wordt hier ook naar terug verwezen.



Figuur : Uitwerking Haskell

#### 3.3.1.2 Functionele aspecten

In dit kopje zijn verschillende functionele aspecten toegelicht die terugkomen in mijn uitwerking van de fractal tree.

##### Type inference

Ondanks dat ik er niet voor heb gekozen om type inference te gebruiken in mijn oplossing wil ik wel benoemd hebben dat dit had gekund. Voorbeelden hiervan zijn de type declaraties van de functies *applyAlgorithm* en *fractal* op lijn 23 en 27. Als deze er niet waren zou Haskell type inference toepassen en zou het programma nog steeds naar behoren werken.

Ik heb hier niet voor gekozen omdat het voor mij heel veel duidelijkheid schept tijdens het programmeren.

##### Higher order functions

Binnen mijn uitwerking worden op verschillende plekken HOFs toegepast. Een voorbeeld hiervan is de *iterate* functie op lijn 27 binnen de *applyAlgorithm* methode.

De iterate functie krijgt een functie als argument en een element waar deze op toepasbaar is. De bijbehorende type declaratie is: . Hier worden gebruikt om aan te tonen dat de functie polymorfisch is. De *iterate* functie creëert een oneindige lijst waar het eerste element wordt berekend door de functie toe te passen op het tweede argument, het tweede element door de functie toe te passen op het voorgaande resultaat, en zo verder.

In het geval van de *applyAlgorithm* methode worden de functies uit het *fractalAlgorithm* methode meegegeven die constant worden toegepast op het *pict* argument. Hierdoor wordt een oneindige lijst gemaakt die langzaam de fractal tree opbouwt. De *take* functie zorgt er echter voor dat de lijst niet daadwerkelijk oneindig wordt door de iterate functie te stoppen na *n* iteraties van de fractal. In dit geval is dat 10.

##### Function composition & function application

Een veel voorkomende term in Haskell die ik niet eerder heb toegelicht is function composition. Dit is tevens een veelvoorkomende term in de wiskunde, aangezien Haskell sterk leunt op wiskundige expressies is het dan ook niet raar deze terug te vinden in de taal. FC werkt als volgt:

Stel je hebt twee functies, en , waarbij de input van de output van is. Aan de hand van function composition kan dit worden herschreven als .

Binnen Haskell wordt function composition gedaan met de (.) infix operator. In mijn code wordt deze toegepast op lijn 19 en 20 door de *fractalAlgorithm.* Hier worden de drie functies *Rotate, Scale* en *Translate* in serie gezet door het toepassen van function composition.

Function application werkt in principe hetzelfde, hiervoor wordt de ($) infix operator gebruikt. Het verschil tussen de twee is dat de ($) een hogere prioriteit in de expressie heeft dan de (.) operator.

Een voorbeeld hiervan is de main functie op lijn 38. Hier wordt de ($) operator gebruikt om de uitkomst van de *pythagor* functie te chainen aan *black.* Hier dient *pythagor* een hoger prioriteit te hebben dan *black.* Als de (.) operator hier toegepast zou worden produceert dit een foutmelding, ($) dwingt te juiste ‘prescedence’ af.

##### Monad

In mijn uitwerking is tevens het gebruik van Monads terug te vinden in de *applyAlgorithm* en *fractal* functie. Beide functies maken namelijk gebruik van lists types, die binnen Haskell ook als Monads worden beschouwd.

##### Pure functions

Een Haskell programma zou niet compleet zijn zonder het gebruik van pure functies, binnen mijn uitwerking maak ik hier dan ook gebruik van. Zowel de *applyAlgorithm* als de *fractal* functie voldoen aan de drie eisen waarn een functie aan moet voldoen om ‘pure’ genoemd te worden. (Zie: [*“Welke termen komen veel voor binnen het functoinele paradigma?” – Purity”*](#_3.1.2.1_Purity)).

De *pythagor* daarentegen is geen pure functie. Dit komt omdat het gebruik maakt van de globale variabele *fractalAlgoritme.* Ondanks dat het de waarde hiervan niet wijzigt gebruikt het een variabele buiten zijn eigen scope, wat niet mag.   
Nu ik er achteraf naar kijk zou de *fractalAlgorithm* variabele opgeslagen kunnen worden in de *where-clause* van de *pythagor* functie, aangezien dit de enige plek is waar deze wordt gebruikt. Dit zou de functie wel puur maken en het programma van betere kwaliteit.

##### Immutable data

Expressies en variabelen binnen Haskell zijn standaard ‘immutable’. Dit houdt in dat Haskell de programmeur niet toestaat de waarde hiervan run-time te wijzigen.

Dit is ook terug te zien in mijn uitwerking, bijvoorbeeld de *algoScale* en de *fractalAlgorithm* variabelen.

# 4. Conclusie

Over het algemeen ben ik onzettend tevreden met het resultaat van dit beroepsproduct. Ik aan de hand van deze opdracht een eerste kennismaking gedaan met het functionele paradigma en heb hier ontzettend veel kennis over opgedaan. Tevens heb ik door het uitwerken van de Pythagoras’ fractal tree in Haskell een nieuwe taal geleerd en durf te zeggen dat ik deze op een basisniveau redelijk beheers.

Ik ben erg trots op het resultaat dat ik heb weten te leveren en vond het ontzettend interessant op meer te weten te komen over het functionele paradigma en hoe dit in relatie staat met object-geörienteerd programmeren. Het werken aan deze opdracht heeft ook nieuwe vragen bij mij opgedaan. Zo ben ik nog niet helemaal thuis in Monads en ben ik benieuwd hoe multithreading in Haskell wordt geimplementeerd. Dit zijn onderwerpen waar ik in de toekomst zeker nog eens naar ga kijken.

Al met al kan ik dit beroepsproduct niet anders beschouwen dan geslaagd.

# 6. Literatuurlijst

Golliwitzer, Z. (2020) *Imperative vs. Declarative Programming (procedural, functional, and OOP).* Geraadpleegd op 8 juni 2022, via:<https://medium.com/@zach.gollwitzer/imperative-vs-declarative-programming-procedural-functional-and-oop-b03a53ba745c>.

Brassuer, A. (2014) *Functional Programming: Pure Functions.* Geraadpleegd op 8 juni 2022, via: <https://www.sitepoint.com/functional-programming-pure-functions/#:~:text=A%20pure%20function%20is%20a,always%20return%20the%20same%20result>.

Bencevic, M. (2018). *What is Lazy Evaluation? — Programming Word of the*. Geraadpleegd op 8 juni 2022, via: <https://medium.com/background-thread/what-is-lazyevaluation-programming-word-of-the-day-8a6f4410053f>.

Jitani, V. (2022). *Monads explained.* Geraadpleegd op 8 juni 2022, via: <https://towardsdatascience.com/monads-from-the-lens-of-imperative-programmer-af1ab8c8790c>.

Haskell Wiki. (2020) *Currying.* Geraadpleegd op 23 oktober 2020, via: <https://wiki.haskell.org/Currying>.

Gupta, D. (2018) *Javascript – Currying VS Partial Application.* Geraadpleegd op 23 oktober 2020, via: <https://towardsdatascience.com/javascript-currying-vs-partial-application-4db5b2442be8>.

Ionos. (2020). *What is Haskell? A portrait of the Haskell functional programming language.* Geraadpleegd op 9 juni 2022, via: <https://www.ionos.com/digitalguide/websites/web-development/what-is-haskell/>.