Bachelorscriptie 2011
-CleanDoc-

Wouter Lockefeer – 0545228

Definities

Term	**Definitie**
Clean	Clean 2.1
CleanDoc	Het te ontwikkelen documentatiesysteem voor Clean-code.
Entity	Noemer voor values, types en classes.
Haddock	Haddock 2.8.0
Haskell	Haskell 98

Inhoud

Inleiding

Clean is de functionele programmeertaal die sinds 1984 ontwikkeld wordt aan de Radboud Universiteit Nijmegen. Hoewel de ontwikkelaars en gebruikers vol lof zijn over de kracht en schoonheid van de taal, valt de taal nog in het niet bij de populariteit van andere programmeertalen. Een veelgehoorde klacht onder studenten is dat de documentatie van Clean niet altijd volledig, op orde of up to date is. Een reden hiervoor kan zijn dat het voor programmeurs niet makkelijk wordt gemaakt om (automatisch gegenereerde) documentatie op te stellen voor hun applicatie of library.

De ontwikkelaars en gebruikers van Clean aan het Radboud zelf gaven aan behoefte te hebben aan een geautomatiseerd documentatiesysteem vergelijkbaar met JavaDoc. Met het bestaan van een dergelijke CleanDoc kan een makkelijke, heldere en uniforme manier om documentatie aan te leggen worden bevorderd, hetgeen kan leiden tot een betere algehele documentatie voor met name de vele libraries voor de taal.

Probleemstelling

Een goede programmeertaal moet niet alleen efficiënte programma's opleveren, maar ook handig zijn in gebruik. Één van de eigenschappen die bijvoorbeeld Java tot een populaire taal maakt, is *JavaDoc* [1]. Comments in de Java-code die van speciale structuur voorzien zijn, worden door JavaDoc geparsed en omgezet in documentatie in HTML of eventueel een ander formaat. De programmeertaal Clean [3] heeft geen dergelijk systeem en wordt op wereldschaal relatief weinig gebruikt.

De functionele programmeertaal Haskell [4][6] lijkt erg sterk op Clean en heeft een eigen documentatiesysteem genaamd Haddock [2][15]. Zeker nu de nieuwe Clean compiler in staat is om een Haskell-dialect te compilen [7], lijkt het voor de hand te liggen dit documentatiesysteem over te nemen voor Clean. Het spreekt echter voor zich dat er wel degelijk verschillen zijn tussen beide talen en dat dit gevolgen heeft voor de implementatie en werking van een dergelijk documentatiesysteem.

Dit leidt tot de volgende onderzoeksvraag:

Wat zijn de belangrijkste verschillen tussen Clean en Haskell met betrekking tot de ontwikkeling van een documentatiesysteem voor Clean geïnspireerd door Haddock en hoe kan dit documentatiesysteem verder verbeterd worden?

Het resultaat van dit onderzoek zal bestaan uit een lijst van richtlijnen voor het ontwikkelen van CleanDoc met Haddock als voorbeeld.

Theoretisch kader

De meeste programmeertalen hebben minstens één, maar vaak zelfs meerdere tools om commentaar in een bepaalde syntax om te laten zetten in beter leesbare documentatie in bijvoorbeeld PDF- of HTML-formaat. Door het gebruik van zulke automatische documentatiesystemen te bevorderen, worden gebruikers aangespoord om een betere documentatie voor hun programma's en libraries aan te leggen.

In de meeste gevallen bestaan de systemen uit een parser, die commentaar uit de

programmeercode haalt en de informatie die dit commentaar bevat, omzet naar een leesbaarder document, vaak voorzien van automatisch gegenereerde informatie en waar mogelijk hyperlinks. Speciale syntax kan gebruikt worden om extra informatie over bijvoorbeeld betekenis en opmaak aan te geven.

Methode

Om een antwoord te vinden op de gestelde onderzoeksvraag is gekozen voor een opdeling in drie hoofdstukken. In het eerste hoofdstuk worden de eigenschappen van Haskell's documentatiesysteem Haddock besproken. In het tweede hoofdstuk worden de overeenkomsten en verschillen tussen Clean en Haskell bekeken en vervolgens vanuit de invalshoek van de ontwikkeling van CleanDoc belicht. In het derde hoofdstuk worden suggesties voor toevoegingen en verbeteringen besproken.

Tot slot wordt in hoofdstuk 4 een opsomming van richtlijnen opgesteld als samenvattend resultaat van dit onderzoek. Hoofdstuk 5 vormt de conclusie/discussie bij dit onderzoek.

1. Functionaliteit van Haddock
Aan de hand van de documentatie van Haddock worden de eigenschappen van het documentatiesysteem belicht.

2. Vergelijking tussen Clean en Haskell
Aan de hand van literatuur worden Clean en Haskell met elkaar vergeleken. De verschillen tussen de talen worden belicht, mede aan de hand van de eigenschappen van Haddock zoals gevonden in hoofdstuk 1.

3. Aanpassingen & verbeteringen
Suggesties voor toevoegingen en verbeteringen aan het concept van Haddock worden besproken.

4. Resultaat: van Haddock naar CleanDoc
De uiteindelijke lijst van richtlijnen wordt opgeleverd om met Haddock als voorbeeld CleanDoc te realiseren.

5. Conclusie & Discussie
Een terugblik op het onderzoek, waarbij de onderzoeksvraag, het proces en het resultaat worden samengevat.

Hoofdstuk 1 – Functionaliteit van Haddock

Sinds 2002 bestaat Haddock, een documentatiesysteem voor Haskell dat standaard geïntegreerd is in de Haskell Package. Het systeem is ontwikkeld, omdat er behoefte was aan een simpele, lightweight manier om documentatie aan code toe te voegen. Haddock is hoofdzakelijk geïnspireerd door IDoc[9], HDoc[10] en Doxygen[11].

Haddock is ontwikkeld met zes richtlijnen in gedachten:

- Hou implementatie en documentatie dicht bij elkaar om te bevorderen dat de documentatie up to date blijft.
- Genereer automatisch standaarddocumentatie uit de broncode.
- Hou de syntax simpel, overzichtelijk en beknopt.
- Hou de documentatie los van implementatiedetails. Het gaat om wát de code doet, niet hóe de code het doet.
- Voorzie de documentatie van hyperlinks waar mogelijk.
- Hou het simpel om output in meerdere formaten makkelijk mogelijk te maken.

Deze richtlijnen zijn redelijk standaard voor een documentatiesysteem en zijn niet specifiek gericht op Haskell. Voor deze scriptie is onderzocht Haddock versie 2.8.0, hierna kortweg Haddock, gebruikt voor Haskell 98, hierna kortweg Haskell.

Automatische documentatie

Haddock is in staat om documentatie te genereren uit bepaalde delen van Haskell-code, zonder dat hiervoor door de gebruiker geschreven Haddock-comments nodig zijn. De Haskell-code waaruit Haddock deze informatie kan onttrekken, zijn: function type signatures, type declarations en class declarations.

Bijvoorbeeld, bekijk de volgende code:

```
inc :: Int -> Int
inc x = x + 1
```

Dit resulteert in de automatische documentatie voor function `inc`:

```
inc :: Int -> Int
```

Haddock neemt simpelweg de type signature over, maar creëert ook hyperlinks in het geval dat de argumenten types zijn die geëxporteerd worden door de modules waar Haddock de documentatie van genereert.

In het geval van type classes genereert Haddock in de documentatie van de betreffende type class ook een lijst van alle instanties van deze type class.

```
class Clickable c where
  onClick :: c -> Action
```

In het geval dat Clickable twee instanties heeft, levert dit de volgende alfabetische lijst van instanties op onderaan de pagina van de Clickable type class.

```
Instances
  Clickable Button
  Clickable TextArea
```

Ook voor deze instanties geldt dat de namen van de entities van hyperlinks voorzien worden.

Handgemaakte documentatie

Gebruikers kunnen zelf documentatie toevoegen aan delen van hun code. Haddock comments beginnen met `-- |` wanneer de te documenteren code volgt op de comment en `-- ^` wanneer de te documenteren code voorafgegaan is aan de comment.

Top-level declarations

Top-level taalconstructies die van commentaar voorzien kunnen worden, zijn:

- type signatures van top-level functions
- data declarations
- newtype declarations
- type declarations
- class declarations

Neem bijvoorbeeld onze eerdere code, nu voorzien van een Haddock comment:

```
-- |Verhoog een integer met 1.
inc :: Int -> Int
inc x = x + 1
```

Of, alternatief:

```
inc :: Int -> Int
-- ^Verhoog een integer met 1.
inc x = x + 1
```

Beide voorbeelden resulteren in dezelfde documentatie voor function `inc`:

```
inc :: Int -> Int
Verhoog een integer met 1.
```

Optioneel kan de comment uit meerdere regels bestaan:

```
-- |Verhoog een integer met 1.
-- Extra regel.
inc :: Int -> Int
inc x = x + 1
```

Of alternatief:

```
{-|
  Verhoog een integer met 1.
```

```
  Extra regel.
-}
inc :: Int -> Int
inc x = x + 1
```

Resulteren beide in:

```
inc :: Int -> Int
```
Verhoog een integer met 1.
Extra regel.

Onderdelen van declarations

Soms is het nuttig om specifieke onderdelen van declarations van documentatie te voorzien. Dit is mogelijk bij:

- class methods
- data constructors en data records
- function arguments

Class method:

```
class C a where
  -- | Documentatie voor method 'f' van class 'C'.
  f :: a -> Int
```

Data constructor:

```
data T a b
  -- | Documentatie voor constructor 'C' van data 'T'.
  = C a b
```

Data record:

```
data R a b =
  C { -- | Documentatie voor field 'a' van data 'R'.
      a :: a,
      -- | Documentatie voor field 'a' van data 'R'.
      b :: b
    }
```

Function argument:

```
inc  :: Int  -- ^ De integer die verhoogd moet worden.
     -> Int  -- ^ De input, verhoogd met 1.
```

Modules en export lists

Code wordt in Haskell opgedeeld in modules. Een module kan een export list bevatten, die aangeeft welke code door de module geëxporteerd wordt. Wanneer er geen export list is, worden alle entities geëxporteerd.

Een module kan een algemene beschrijving hebben boven de module header:

```
-- | Documentatie voor module Foo.
module Foo where
```

Dit resulteert in een apart stukje documentatie getiteld `Description` bovenaan de module-documentatie.

Wanneer een module een export list heeft, wordt normaliter enkel van de geëxporteerde entities documentatie gegenereerd. De export list bepaalt dan ook de volgorde waarin de entities voorkomen in de documentatie. Headings kunnen worden aangegeven met de code `-- *` , waarbij het aantal asterisken de diepte van de heading aangeeft.

Neem bijvoorbeeld deze module met export list:

```
-- | Documentatie voor module M.
module M (
  -- * Modules
  module L,
  -- * Classes
  C(..),
  -- * Types
  -- ** Data types
  T1, T2,
  -- ** Records
  R1, R2,
  -- * Functions
  f, g
) where
```

Dit zou resulteren in de documentatie met headings, sub-headings, de declarations van de entities en de bijgevoegde omschrijvingen van de entities. Alle entity-namen en module-namen hyperlinken naar hun eigen documentatie-pagina of naar een positie binnen de overzichtspagina van de module.

In het geval dat de export list ontbreekt, genereert Haddock documentatie van alle entities die voorkomen in de module.

Soms kan het handig zijn extra documentatie toe te voegen, die niet specifiek hoort bij een entity. Dit kan in de export list door middel van een comment met `-- |` . Wanneer tekst lang en slordig wordt binnen een export list, mag deze buiten de export list gedefinieerd worden met de combinatie `-- $`, zoals hieronder:

```
-- | Documentatie voor module Foo.
module M (
  -- $extra
  ) where

-- $extra
-- Dit is extra documentatie, vrij gedefinieerd buiten de export
-- list.
```

De code `-- $extra` refereert naar de documentatie buiten de export list.

Tot slot kan een module nog attributen hebben, gedefinieerd in een pragma bovenaan de module file als volgt:

```
{-# OPTIONS_HADDOCK hide, prune, ignore-exports, not-home #-}
```

De vier attributen zijn optioneel en hebben de volgende betekenis.

- `hide`: Genereer geen aparte documentatie voor deze module, maar genereer wel documentatie van entities van deze module die door andere modules geëxporteerd worden.
- `prune`: Genereer geen documentatie van entities waar geen door de gebruiker gecreëerde Haddock-comments bij staan.
- `ignore-exports`: Negeer de export list en doe alsof deze niet bestaat. Headings mogen nu aangegeven worden in de body van de module.
- `not-home`: Deze module is niet de home module voor haar entities, behalve wanneer een entity niet door een andere module geëxporteerd wordt.

Opmaak en referenties

Tot slot zijn er allerlei manieren beschikbaar waarmee de opmaak en functionaliteit van de documentatie aan te passen is.

- Paragrafen: witregels binnen comments geven scheiding van paragrafen aan.
- Code blocks (monospaced):
  `@ code @` Een code block dat opmaak kan bevatten.
  `> code` Een code block dat letterlijk genomen wordt. Voor élke regel moet een `>` staan.
- Nadruk: met `/ commentaar /` krijgt commentaar nadruk.
- Referentie naar entity: `'entity'`
- Referentie naar module: `"module"`
- Opsommingslijst:
  ```
  * item1   of    - item1
  * item2         - item2
  ```
- Genummerde lijst:
  ```
  1. item1  of    (1) item1
  2. item2        (2) item2
  ```
- Definitielijst:
  ```
  [@entity1] Beschrijving van @entity1@.
  [@entity2] Beschrijving van @entity2@.
  ```
- URL's: Een URL kan naar een webpagina verwijzen als volgt: `<URL>`
- Anchors: `#label#` genereert een hyperlink naar de anchor met id `label`. Voor anchors naar een andere module is de syntax `"module#anchor"` .

Opties

De lijst van extra opties is niet direct van belang voor dit onderzoek en is bewust buiten beschouwing gelaten. Het betreft allerlei kleine extra opties voor bijvoorbeeld verschillende output-formaten met gespecificeerde stijlen.

Hoofdstuk 2 – Vergelijking tussen Clean en Haskell

Hoewel Clean en Haskell op het eerste oog veel op elkaar lijken, zijn er als vanzelfsprekend belangrijke verschillen tussen beide talen. Niet alleen syntactisch, maar zeker ook qua functionaliteit is elke taal uniek. In dit hoofdstuk worden overeenkomsten en verschillen tussen beide talen besproken, in het bijzonder wanneer deze significante gevolgen hebben voor de ontwikkeling van CleanDoc.

Voor dit hoofdstuk zijn met name twee artikelen veel gebruikt. Ten eerste *Haskell to Clean Translation (2004)* [8], waarin Matthew Naylor onderzoekt hoe een vertaler gemaakt kan worden van Haskell 98 naar Clean 2.1. Hierbij maakt hij een diepgaande vergelijking tussen de talen. Ten tweede *Exchanging Sources Between Clean and Haskell (2010)* [7], waarin besproken wordt hoe de nieuwste Clean compiler dialecten van Clean en Haskell gemixt kan compilen. Ook hier worden verschillen tussen de talen besproken, met als toevoeging uitleg over hoe de dialecten van Clean en Haskell een brug tussen de talen vormen.

Modules

In Clean worden declarations in `.dcl` bestanden opgeslagen, terwijl de implementaties ervan in `.icl` bestanden staan. In Haskell staan de namen van de te exporteren entities in de (optionele) export list, bovenaan het module-bestand. De type declarations en implementations staan buiten de export list. Wanneer de export list ontbreekt, worden alle geïmplementeerde entities ook geëxporteerd.

In `.dcl` bestanden in Clean kunnen sommige definities staan die niet voorkomen in de implementation module. Dit zorgt ervoor dat, als deze definities van documentatie voorzien moeten kunnen worden, CleanDoc het moet toestaan dat commentaar gezet mag worden in de declaration module. Voor CleanDoc is het dus noodzakelijk dat commentaar zowel in de `.dcl` als in de `.icl` mag staan.

Voor abstracte datatypes staat slechts de linkerkant van de type-regel genoemd in het `.dcl` bestand en de volledige regel in het `.icl` bestand. Afhankelijk van waar het commentaar staat, zal CleanDoc hier dus kiezen hoe deze definitie wordt weergegeven in de documentatie. Als er geen commentaar staat, raad ik aan standaard de versie van de `.dcl` te nemen, aangezien het een abstract datatype betreft en het aannemelijk is dat de programmeur zo min mogelijk over de implementatie ervan wil laten zien aan de gebruiker.

Afgezien van deze verschillen lijken Clean's declaration module en Haskell's export list sterk op elkaar. Beide kunnen gebruikt worden om headings en overige documentatie te noteren, alswel om de volgorde van de te documenteren entities aan te geven.

```
// Clean (.dcl file)
power :: Int Int -> Int

-- Haskell
Module Math(
  power
) where
```

Functions

Functions lijken in Clean en Haskell sterk op elkaar. Een belangrijk verschil is echter dat in Clean de ariteit van functions expliciet duidelijk is in de definitie, terwijl dit in Haskell niet zo is. Bekijk bijvoorbeeld de volgende function signatures van `power` in Clean en Haskell.

```
// Clean
power :: Int Int -> Int

-- Haskell
power :: Int -> Int -> Int
```

Beide functions leveren als resultaat de eerste parameter tot de macht van de tweede parameter op. De function signature van `power` in Clean geeft duidelijk weer dat de ariteit van deze functie 2 is. De function signature in Haskell heeft, zoals elke function signature in Haskell, een ariteit van 1, ook al verwacht de function duidelijk twee parameters voor het zijn resultaat oplevert.

Clean biedt, in tegenstelling tot Haskell, de mogelijkheid tot het gebruik van strictness annotations in function declarations, wat wil zeggen dat (delen van) functions eager geëvalueerd kunnen worden waar nodig. Eventueel kan bij het voorkomen van stricte parameters hier in de documentatie extra nadruk op gelegd kan worden. Ook kent Clean uniqueness typing, wat bij het volgende punt verder besproken wordt.

Bij function signatures in Clean is, in tegenstelling tot in Haskell, de ariteit van de function expliciet weergegeven. Dit stelt een documentatiesysteem in staat deze ariteit expliciet te benadrukken in de documentatie. In Haddock is het noodzakelijk dat dergelijke informatie expliciet handmatig wordt toegevoegd in comments. Dit is dus een voordeel voor CleanDoc.

CleanDoc zal dus wat betreft functions overeenkomen met Haddock, met als toevoeging dat extra informatie over de ariteit van de function automatisch kan worden gedocumenteerd. Het is bovendien mogelijk om de argumenten van de function van eigen documentatie te voorzien zoals in Haddock. Dit kan handiger in CleanDoc, omdat CleanDoc al kan zien hoeveel parameters er zijn en wat de functie oplevert. Dit zou er in de documentatie als volgt uit kunnen gaan zien voor de function `power`:

```
Function:       power
Parameters (2): Int      Het grondtal.
                Int      De exponent.
Result:         Int      Het grondtal tot de macht van de
                         exponent.
power :: Int Int -> Int
```

Uniqueness typing

Clean kent uniqueness typing. Types en variable types kunnen voorzien worden van een uniqueness attribute: `*` . De compiler dwingt dan af dat deze types en variable types niet gekopieerd kunnen worden en destructief ge-update worden. De mogelijkheid om grote objecten zoals records en arrays destructief te kunnen updaten, hetgeen betekent dat er geen volledige kopie van deze objecten gemaakt hoeft te worden, zorgt voor mogelijk vele malen efficiëntere code in Clean ten opzichte van Haskell.

```
// Clean
// Een function met uniqueness typing.
changeInfo :: *BigObject -> *BigObject
```

De bovenstaande function ontvangt een unieke BigObject-waarde, die destructief ge-update kan worden. Het is veel efficiënter een dergelijk groot object niet te hoeven kopiëren om er een (kleine) aanpassing aan te doen.

In Haskell bestaat geen uniqueness typing en zodoende doet Haddock er ook niks mee. Voor CleanDoc kan extra functionaliteit verzonnen worden om uniqueness typing weer te geven. Het is echter niet direct duidelijk hoe uniqueness typing extra aandacht zou moeten krijgen in CleanDoc. Op zijn minst is het mogelijk om een aparte stijl toe te kennen aan unieke entities. Verder zijn er wellicht mogelijkheden om CleanDoc te laten herkennen wanneer functies efficiënter (kunnen) zijn dankzij destructieve updates van (mogelijk) grote objecten.

Macro's

De verschillen tussen beide talen zijn syntactisch wat macro's betreft. Echter, aangezien macro's niet als zodanig bestaan in Haskell en wel in Clean, kan CleanDoc expliciet aangeven bij een macro dat het een macro betreft.

```
// Clean
double x :== x + x

-- Haskell
{-# INLINE double #-}
double x = x + x
```

Macro's worden bij het compilen vervangen en zijn dus efficiënte inline code. Het is echter zo dat macro's geen type-declaratie hebben. CleanDoc moet dus een uitzondering maken voor macro's en de documentatie van de macro boven de macro zelf toestaan.

Newtypes

In Clean bestaan geen newtypes. In plaats van een newtype kan een algebraïsche data declaration met een strict argument gebruikt worden, maar dit is minder efficiënt dan een newtype, omdat deze data declarations niet tijdens het compilen verwijderd worden.

In Haskell bestaan wel newtypes. In tegenstelling tot type synonyms kunnen newtypes gebruikt worden als instance van een type class en kunnen ze recursief zijn. Bovendien worden newtype constructors tijdens het compilen verwijderd, omdat ze slechts voor de programmeur betekenis hebben, maar niet voor het programma tijdens run-time.

```
// Clean
:: Natural = MakeNatural !Int

-- Haskell
newtype Natural = MakeNatural Int
```

Wegens gebrek aan newtypes in Clean, hoeft CleanDoc hier niks mee te doen.

Type classes

In Clean kunnen type classes met meerdere parameters gebruikt worden. Default members zijn mogelijk op class en instance niveau door middel van macro's, maar default members op class niveau kunnen niet opnieuw gedefinieerd worden op instance niveau.

In Haskell hebben type classes slechts één parameter. Default members zijn slechts mogelijk op class niveau, maar kunnen op instance niveau opnieuw gedefinieerd worden.

```
// Clean
class Eq a where
     (==) :: a a -> Bool
     (/=) :: a a -> Bool
     // Defaults voor de equality class als macro's.
     (/=) x y :== not (x == y)

// Default op instance niveau. Wordt gebruikt wanneer geen
// matchende instance gevonden kon worden.
instance Eq a where
     _ == _ = False

-- Haskell
class Eq a where
     (==) :: a a -> Bool
     (/=) :: a a -> Bool
     -- Default members voor de quality class.
     x == y = not (x /= y)
     x /= y = not (x == y)

-- Herdefiniëring op instance niveau mogelijk.
instance Eq Int where
     x == x = True
     x == y = False
```

Wederom geldt voor de ariteit van deze member functions dat bij Clean de ariteit van functies duidelijk is, terwijl dit bij Haskell niet zo is.

In Haddock-documentatie wordt bij types vermeld van welke type classes deze types instances zijn. Dezelfde functionaliteit is gewenst in CleanDoc, zonder aanpassingen.

De verschillen tussen type classes in beide talen zijn klein en ondanks deze verschillen praktisch equivalent. De verschillen zijn niet direct van invloed op de ontwikkeling van CleanDoc.

Monads

Beide talen kennen een implementatie van monads. In beide talen betreft het een class met twee members, `return` en `>>=`. Bij Haskell is het echter in veel gevallen noodzakelijk monads te gebruiken, aangezien de taal geen uniqueness typing kent.

Records

In Clean worden records geïmplementeerd als algebraïsch datatype zonder constructor

met één alternatief. Record fields van verschillende records mogen overlappende field names hebben. Omdat Clean toestaat dat unieke objecten destructief worden ge-update, kan erg efficiënt met grote objecten gewerkt worden.

```
// Clean
:: Dog = { name :: String, age :: Int }
:: Cat = { name :: String, age :: Int, likesMilk :: Bool }
```

De code laat zien dat verschillende record types dezelfde field names mogen hebben, maar elke record type heeft maar 1 alternatief.

In Haskell zijn records een algebraïsch datatype met constructor met meerdere alternatieven mogelijk, zolang de field names van hetzelfde type zijn bij elk alternatief. In tegenstelling tot Clean produceert Haskell voor elke field label een function met dezelfde naam, die het record type als argument ontvangt en als resultaat de field value oplevert.

```
-- Haskell
data Pet = Dog { name :: String, age :: Int }
         | Cat { name :: String, age :: Int, likesMilk :: Bool }

-- Niet toegestaan nu wegens het bestaan van function 'name':
-- data Child = Son { name :: String }
--            | Daughter { name :: String }
```

Echter, doordat voor elke field name een nieuwe functie wordt gedefinieerd, kunnen twee verschillende record types geen overlappende field names hebben. De field values van een record kunnen als volgt opgevraagd worden:

```
// Clean
myDog.name

-- Haskell
name myDog
```

Het gedrag van records met meerdere alternatieven kan alsnog als volgt in Clean nagebootst worden:

```
// Clean
:: Pet = Dog dog | Cat cat
```

Let hierbij op dat `Dog` en `Cat` beide een field name `foo` mogen hebben met voor elk een eigen type, in tegenstelling tot Haskell.

In beide talen betreft dit algebraïsche datatypes. Voor CleanDoc kan het gedrag van Haddock simpelweg overgenomen worden, aangezien er geen relevante verschillen zijn voor het documentatiesysteem.

Arrays

Clean biedt native support voor arrays. Arrays kunnen simpel weergegeven worden met de `{elementType}`-notatie.

```
// Clean
names :: *{String}
names = { "Tom", "Dick", "Harry" }
```

Arrays worden in Clean standaard destructief ge-update. Dit maakt arrays in Clean bijzonder efficiënt, doordat voor mutaties geen volledige kopie van de array gemaakt hoeft te worden. Arrays kunnen lazy, strict en unboxed zijn en gebruikt worden in array patterns. Elementen selecteren uit arrays gaat praktisch hetzelfde als bij records, met een index in plaats van een field name:

```
// Clean
names.[0] // "Tom"
```

Het updaten van arrays gaat evenwel op dezelfde manier als bij records:

```
// Clean
{ names & [0] = "Sally" } // { "Sally", "Dick", "Harry" }
```

In Haskell is er beperkte support voor arrays via de standaard module `Array`. Arrays zijn in Haskell immutable en er is geen support voor array patterns en array selections. Er kan met de `//` function wel een nieuwe array uit een oude worden gemaakt, met aanpassingen, maar bij grote arrays is dit bijzonder inefficiënt, aangezien de array volledig opnieuw aangemaakt wordt.

Dynamic typing

Clean biedt, in tegenstelling tot Haskell, de mogelijkheid tot dynamic programming. Dit houdt in dat waardes, samen met hun type, ' ge-wrapped' kunnen worden in een zogenaamde dynamic value. Door middel van pattern matching kan daar later weer een waarde uit geëxtraheerd worden. Een groot voordeel van dynamic typing is dat dergelijke dynamic values weggeschreven naar en gelezen uit files kunnen worden.

```
// Clean
// unwrap een dynamic value door middel van pattern matching:
unwrap :: Dynamic -> String
unwrap (0 :: Int) = “Nul.“
unwrap (n :: Int) = “Een integer.“
unwrap (f :: Int->Int) = “Function van integer naar integer.“
unwrap ((x,y) :: (Int,Int)) = “Twee integers.“
unwrap _ = “Dynamic value is niet herkend.“
```

Er zijn enkele kleine aanpassingen voor CleanDoc te verzinnen met betrekking tot dynamics. Deze zullen aan het eind van dit hoofdstuk besproken worden.

Generic functions

Haskell heeft een speciaal `deriving` keyword, waarmee voor elk type automatisch een instantie van een klein aantal veel voorkomende type classes (zoals `Eq` en `Ord`) te laten genereren is.

Clean gaat hier een stap verder en implementeert generic programming, wat programmeurs in staat stelt een dergelijke automatische afleiding mogelijk te maken voor

elke type class, hetzij standaard of zelf gedefinieerd.

```
// Clean
:: myType = Foo | Bar
derive gEq myType
derive gOrd myType
derive gMyClass myType

-- Haskell
data myType = Foo | Bar
  deriving (Eq, Ord)
```

Er zijn enkele kleine aanpassingen voor CleanDoc te verzinnen met betrekking tot generics. Deze zullen aan het eind van dit hoofdstuk besproken worden.

Overige kleine verschillen

- Haskell biedt support voor integers van arbitraire grootte met het `Integer` type. De normale `Int` type in Clean en Haskell zijn 32-bits. Voor dit onderzoek is dit verschil irrelevant.
- Comments worden in Clean genoteerd met `//` i.p.v. `--` . Dit is een klein syntactisch verschil voor dit onderzoek.
- In Haskell kunnen entities gekwalificeerd worden met hun module-naam om namespace collision te voorkomen met de syntax myModule.myEntity . Dit is niet mogelijk in Clean. Dit is slechts een kleine syntactische kwalificatie en voor dit onderzoek dus niet bijzonder relevant.

Richting CleanDoc

Een aantal verschillen tussen Clean en Haskell is noemenswaardig voor dit onderzoek, aangezien er in CleanDoc aandacht aan besteed zou kunnen of moeten worden. Het betreft Uniqueness Typing, Arrays, Dynamic Typing en Generic Typing. Deze vier aspecten komen niet of minder native voor in Haskell en kunnen dus in CleanDoc extra aandacht krijgen. Ook zijn declaration modules en export lists niet geheel hetzelfde. Kort belichten we voor deze aspecten enkele mogelijkheden.

Uniqueness Typing

Unieke entities kunnen in CleanDoc van aparte stijl voorzien worden. Eventueel is extra automatische documentatie mogelijk in verband met efficiëntie, wanneer uniqueness van toepassing is op (waarschijnlijk) grote objecten.

Arrays

Arrays kunnen in CleanDoc van aparte stijl voorzien worden. Eventueel zijn de verschillende types arrays nog visueel van elkaar te onderscheiden.

Dynamic Typing

Veelvoorkomende functies bij dynamic typing zijn wrap- en unwrap-functies. Een wrapper pakt iets in als een dynamic value en een unwrapper haalt met pattern matching een waarde uit een dynamic value. Speciale `@wrap` en `@unwrap` annotaties zijn mogelijk voor

dergelijke functies, om ze speciaal weer te kunnen geven en zoekbaar te maken in documentatie.

Hierbij moet benadrukt worden dat voor deze aspecten van Clean nieuwe functionaliteit in CleanDoc met name voort zal komen uit behoeften vanuit de Clean community. Het is nog niet duidelijk wat Clean-programmeurs hier het liefst wel of niet zien gebeuren.

Generic Typing

Bij documentatie van type classes kan expliciet vermeld worden welke types instances zijn geworden van deze type classes door afleiding met behulp van generics. Bij bijvoorbeeld de `gEq` generic type class zou een lijst staan met alle types die een instance zijn van `gEq`, waarbij de types die via generics zijn afgeleid een aparte stijl hebben of in een aparte lijst staan.

Modules

In CleanDoc moet het mogelijk zijn zowel in `.dcl` als in `.icl` bestanden documentatie te plaatsen. De volgorde van de entities in de documentatie wordt bepaald door de volgorde in de declaration module.

Hoofdstuk 3 – Aanpassingen & Verbeteringen

In dit hoofdstuk worden verdere verbeteringen voorgesteld om het te ontwikkelen documentatiesysteem nog beter te maken. Deze suggesties gaan verder dan Haddock in functionaliteit en geven een indicatie van wat er mogelijk is voor CleanDoc.

Hoogle

Hoogle [12][16] is een zoekmachine voor Haskell-code. Door een tekst-query in te voeren, kan gezocht worden naar functions en types. Standaard-libraries van Haskell zitten ook standaard in de zoekmachine, maar ook in eigen code kan gezocht worden met behulp van Hoogle. Haddock biedt de mogelijkheid om van Haskell-modules text files te genereren die door Hoogle gebruikt kunnen worden om in te zoeken.

In Hoogle zijn text searches en type searches mogelijk. Wanneer er symbolen (zoals ->) in de query voorkomen of wanneer expliciet :: in de query wordt gebruikt, doet Hoogle een type search, waarbij resultaten van dat type gezocht worden. Wanneer er enkel tekst in de query zit, doet Hoogle een text search en zoekt namen die overeenkomen.

Bijvoorbeeld, wanneer de volgende queries worden uitgevoerd op de standaard-libraries van Haskell, levert dit de volgende resultaten op.

Query	**Resultaat**
get time	getCPUTime :: IO Integer getClockTime :: IO ClockTime getCurrentTime :: IO UTCTime ...
:: Int	cINT_SIZE :: Int dOUBLE_SIZE :: Int iNT64_SIZE :: Int ...
Int -> Int -> String	buf_toStr :: BufferOp a -> a -> String ...

Voorbeelden van één text en twee type searches op de standaard-libraries van Haskell met Hoogle.

Goocle

Natuurlijk is het handig om een dergelijk systeem ook mogelijk te maken voor Clean-code. Een 'Goocle' kan op basis van Hoogle gemaakt worden, waarbij snel gezocht kan worden in Clean-modules. Op basis van de eerder genoemde verschillen tussen Clean en Haskell (zie hoofdstuk 2) kan het concept van Hoogle, net zoals Haddock, betrekkelijk simpel worden omgezet naar een Clean-variant.

Clean heeft een nog onbesproken voordeel. Clean heeft een eigen IDE, waaraan nog steeds actief gewerkt wordt. Dit maakt het in het bijzonder goed mogelijk om documentatie- en programmeertools direct te integreren in de IDE zelf.

Een bekende IDE voor de programmeertaal Java is Eclipse [13]. Deze IDE is voorzien van ongelooflijk veel hulpmiddelen die in de Clean IDE veelal ontbreken. Één van deze

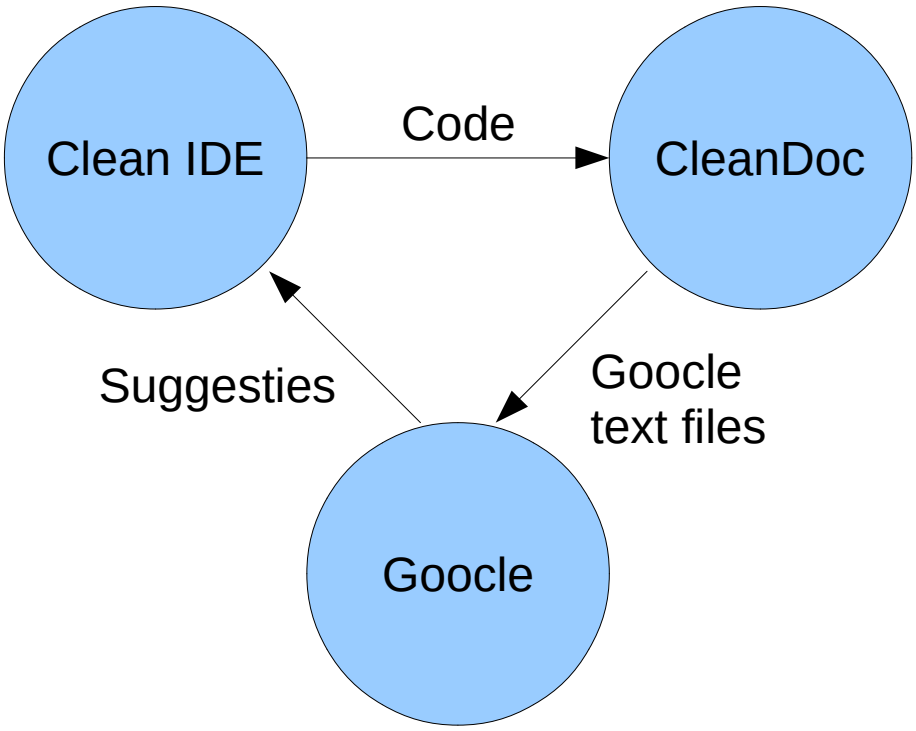
voordelen omvat de real-time programmeerhulp in de vorm van code-suggesties. Neem het volgende voorbeeld waarbij Java wordt geschreven.

```
// Java
public class Main{
  public int getFoo(){
    return 2;
  }
  public int getBar(){
    return this._
  }
}
```

Na het typen van de punt achter `this` geeft Eclipse een lijst met functions die de class Main implementeert (in dit geval `getFoo` en `getBar`) en geeft ze als automatisch in te voegen mogelijkheden. Dit maakt het voor programmeurs bijzonder makkelijk om snel functies binnen classes te vinden en voorkomt dat ze fouten maken.

Een zelfde soort functionaliteit zou een goede toevoeging kunnen zijn voor de Clean IDE. Dit is handig te koppelen aan de “Goocle”-zoekmachine. Tijdens het programmeren kan de huidige code door CleanDoc omgezet worden in door Google leesbare tekstbestanden. Zodoende heeft Goocle altijd de huidige code om in te zoeken. De Clean IDE kan in real-time de code die op dat moment geschreven wordt, voeden als query aan Goocle. Wanneer Goocle resultaten vindt, kan de IDE suggesties doen voor de code die geschreven wordt.

Het is echter wel nodig dat de Clean IDE op de hoogte is van de types waar het suggesties voor vraagt. Hierdoor moet er eigenlijk in real-time type-checking verricht worden. Dit probleem zal niet dieper belicht worden in dit onderzoek.



Figuur 1: Goocle als generator van code-suggesties. Eerst levert de Clean IDE de code aan CleanDoc. CleanDoc verwerkt de code en levert text files op die leesbaar zijn voor

Goocle. Tot slot levert Goocle suggesties op voor de code waar op dat moment aan geschreven wordt en geeft deze door aan de Clean IDE.

Het dient opgemerkt te worden dat CleanDoc met deze functionaliteit de functie van een documentatiesysteem overstijgt. Echter, het is goed mogelijk om CleanDoc gebruikt te laten worden als programmeerhulp, doordat Clean een eigen IDE heeft. Hierdoor is het ook logischer om extra opties, zoals het hier omschreven genereren van suggesties aan programmeurs in real-time, toe te voegen aan het concept van CleanDoc.

Zoekfuncties in de Clean IDE

Met de implementatie van Goocle is het ook mogelijk om de zoekfunctie van de Clean IDE zelf uit te breiden met de functionaliteit van Goocle. De huidige zoekfunctie is beperkt en als Goocle al gebruikt wordt voor het genereren van code-suggesties, is er geen reden om het niet ook te gebruiken voor de gewone zoekfunctie.

Documentatie van implementatie

Bij Haddock en veel andere documentatiesystemen ligt de nadruk op het generen van documentatie voor gebruikers die zelf niet meer aan de betreffende code gaan sleutelen. In het geval van libraries is dit een goede aanname, aangezien programmeurs slechts de door libraries geëxporteerde entities gebruiken, zonder bijzonder geïnteresseerd te zijn in de implementatiedetails.

Er zijn echter gevallen te verzinnen waarbij documentatie gewenst is waarbij ook de exacte implementatie getoond wordt. Stel dat meerdere programmeurs werken aan een project. Bij het lezen van de documentatie van (al dan niet volledig uitgewerkte) code, kan het voor programmeurs van belang zijn delen van de implementatie bij bepaalde entities te kunnen zien, bijvoorbeeld wanneer de programmeur die de code schreef, twijfelt of een deel van de code wel goed genoeg is.

Het zou dus handig zijn als het in CleanDoc mogelijk was met behulp van annotaties aan te geven dat een stuk implementatie weergegeven moet worden in de documentatie. De volgende voorbeeldcode laat zien hoe dit zou kunnen werken.

```
// | Merge two lists of integers.
merge:: [Int] [Int] -> [Int]
merge f [] = f
merge [] s = s
// | @imp
merge f=:[x:xs] s=:[y:ys] // Could this be done better?
| x<y = [x:merge xs s]
| x==y = merge f ys
| otherwise = [y:merge f ys]
// | @/imp
```

Hierbij is gekozen om implementatiedetails weer te geven tussen de speciale annotaties `@imp` en `@/imp` . Deze code zou dus resulteren in de volgende documentatie.

```
merge:: [Int] [Int] -> [Int]
```
Merge two lists of integers.

Implementation details

```
...
merge f=:[x:xs] s=:[y:ys] // Could this be done better?
| x<y = [x:merge xs s]
| x==y = merge f ys
| otherwise = [y:merge f ys]
…
```

Alternatief is het mogelijk om @imp toe te voegen in de comment boven de functie om de volledige implementatie in de documentatie weer te geven.

```
// | Merge two lists of integers. @imp
merge:: [Int] [Int] -> [Int]
```

Natuurlijk is de syntax voor deze eigenschap slechts een suggestie en weinig van belang. Een uiteindelijke implementatie kan kiezen voor welke annotatie dan ook gewenst is.

Deze optie betekent dat de vierde streefregel van Haddock geschonden wordt: *“Hou de documentatie los van implementatiedetails. Het gaat om wát de code doet, niet hóe de code het doet.”* Echter, dit is een slechts een extra keuzemogelijkheid voor de gebruiker en kan in sommige situaties nuttig zijn.

In het bijzonder sluit deze optie aan bij de eerste streefregel van Haddock: *“Hou implementatie en documentatie dicht bij elkaar om te bevorderen dat de documentatie up to date blijft.”* Zou deze optie niet bestaan, dan zouden programmeurs hun implementatie handmatig in comments moeten benadrukken, waarbij ze het risico lopen dat deze comments ooit niet meer gelijk lopen met de huidige code. Met de voorgestelde optie is dat geen probleem meer.

Geïntegreerde documentatiehulp

Haddock is een documentatiesysteem dat los staat van Haskell editors. Er is geen dusdanig geïntegreerde support voor Haddock dat de editors hulp bieden bij het toevoegen van Haddock-annotaties in code. Met de mogelijkheid tot nauwe integratie van CleanDoc in de Clean IDE, is deze mogelijkheid er wel.

Er zijn veel handige opties te bedenken om het toevoegen van CleanDoc-annotaties simpeler te maken. Hier zullen een aantal voorbeelden gegeven worden.

```
// | Handige functie. Niet te verwarren met de functie 'b_'
foo :: Int -> Int
```

Wanneer de cursor achter de b staat, kan CleanDoc, zeker met de hulp van Goocle, een alfabetische lijst genereren van entities beginnend met character 'b'. Wanneer de cursor weggehaald wordt van deze entity reference in de CleanDoc comment, kan CleanDoc, wederom met de hulp van Goocle, checken of de entity echt bestaat. Zo niet, kan met bijvoorbeeld rode kronkellijntjes aangegeven worden dat de entity reference onjuist is. Dit kan handig zijn bij typefouten of wanneer later ergens een functienaam veranderd wordt.

```
// | Merge two lists of integers.
merge:: [Int] [Int] -> [Int]
merge f [] = f
```

```
merge [] s = s
merge f=:[x:xs] s=:[y:ys] // Could this be done better?
| x<y = [x:merge xs s]
| x==y = merge f ys
| otherwise = [y:merge f ys]
```

In deze eerder genoemde merge function is een deel van de implementatie geselecteerd. Met (bijvoorbeeld) een druk op een knop in een CleanDoc werkbalk of een sneltoets kan deze voorzien worden van de @imp … @/imp annotatie. Dergelijke geautomatiseerde handelingen kunnen het toevoegen van annotaties simpeler en foutloos maken.

In het algemeen kan CleanDoc checken of de syntax van CleanDoc comments op orde is en geen fouten bevat. Dergelijke functionaliteit voorkomt fouten in documentatie en is daarom bijzonder nuttig.

Tags

De meeste code bevat handige naamgevingen voor entities, zodat relaties tussen entities sneller duidelijk zijn en de werking van entities handig is af te lezen. Zo is het bijvoorbeeld in een oogopslag duidelijk dat `isEnabled` een functie is die een Boolean zal opleveren die iets zegt over of een object enabled is.

Het blijft echter altijd mogelijk dat er logische indelingen zijn in code die het niveau van modules en naamgeving overstijgen. Wellicht wil de programmeur aangeven welke functies getest of bewezen zijn, of door welke programmeur ze geschreven zijn. Voor dergelijke informatie is het handig om door de gebruiker verzonnen tags toe te staan. Neem de volgende code als voorbeeld.

```
// | Voeg wat foo toe aan een integer. @wouter@ @tested@
foo :: Int -> Int
```

In dit voorbeeld is de CleanDoc comment bij de function `foo` voorzien van twee tags: `@wouter@` en `@tested@`, die kunnen betekenen dat deze functie geschreven is door Wouter Lockefeer en ook getest is.

Goocle kan vervolgens uitgebreid worden om ook te kunnen zoeken op tags. Ook de door CleanDoc gegenereerde documentatie kan per tag een aparte pagina genereren met daarin alle entities die de betreffende tag hebben. Bovendien kan de Clean IDE suggesties doen voor tagnamen door bestaande tagnamen alfabetisch gesorteerd te tonen wanneer de programmeur een tag typt.

JavaScript in HTML output

Wellicht de meest gangbare output van documentatiesystemen is HTML. De output wordt voorzien van hyperlinks waar mogelijk zodat binnen en tussen HTML-bestanden links kunnen bestaan. Echter biedt HTML dankzij JavaScript veel meer mogelijkheden. JavaScript is haar status als langzame taal behoorlijk ontgroeid en kan met gemak flinke zoekopdrachten verwerken en allerlei HTML animaties uitvoeren, zeker met handige frameworks als jQuery [14].

Een dergelijk zoeksysteem als dat van Goocle is ook te realiseren in JavaScript, wat het mogelijk maakt zonder bezit van de Clean IDE, of zelfs direct op een website, Clean-

documentatie te plaatsen voorzien van de uitgebreide zoekfuncties van Goocle. Dit biedt lezers de mogelijkheid sneller en handiger door de documentatie te zoeken dan wat met enkel hyperlinks mogelijk is.

Dankzij de geïntegreerde mogelijkheden van JavaScript om de HTML DOM (Domain Object Model) te manipuleren, eventueel voorzien van animaties met behulp van bijvoorbeeld jQuery, kan het uiterlijk van de documentatie ook een stuk gelikter dan wat we vaak van documentatie gewend zijn.

Simpele opties zoals het weergeven en verbergen van stukken van de documentatie (door middel van [+] en [–] knoppen, zoals vooral bekend van wiki-pagina's) zijn makkelijk toe te voegen en het is zelfs mogelijk instellingen te onthouden voor de gebruiker. Zo kunnen mensen zelf bepalen welke delen van de documentatie ze willen zien.

Hoofdstuk 4 – Resultaat: van Haddock naar CleanDoc

Het doel van dit onderzoek was om uit te vinden in hoeverre het mogelijk is om een documentatiesysteem te laten ontwikkelen voor Clean, geïnspireerd door Haskell's Haddock. De voorgaande hoofdstukken werken daarheen. In dit hoofdstuk vatten we kort de resultaten samen en geven daarmee een lijst van richtlijnen om vanuit de huidige hulpmiddelen te gaan werken aan CleanDoc.

Functionaliteit van Haddock

Haddock beschikt over een kern van basale functionaliteiten voor een documentatiesysteem. De meeste eigenschappen die genoemd zijn in hoofdstuk 1, zijn zonder meer direct toepasbaar op Clean en kunnen dus overgenomen worden voor CleanDoc.

De verschillen:

- Support voor newtypes is niet nodig in CleanDoc. Clean kent geen newtypes.
- Modules en export lists. Clean gebruikt hiervoor `.dcl` en `.icl` files, zoals omschreven in hoofdstuk 2.
- Clean werkt met `//` comments in plaats van `--` comments. Een miniem syntactisch verschil.

Clean en Haskell

De meeste verschillen tussen Clean en Haskell zijn oppervlakkig of totaal niet van toepassing op documentatie. Doordat de talen zo sterk op elkaar lijken en Haskell praktisch vrijwel een subset is van Clean, kan Haddock redelijk direct worden overgenomen bij het maken van CleanDoc.

De verschillen:

- Uniqueness typing. Hoewel dit het belangrijkste verschil tussen Clean en Haskell is, zijn er praktisch geen directe gevolgen voor CleanDoc gezien vanuit Haddock. Uniqueness typing is in CleanDoc wellicht te benadrukken, maar het bestaan ervan is geen halszaak.
- Arrays. Efficiënter, uitgebreider en bovenal native in Clean. Wederom is extra benadrukking ervan mogelijk in CleanDoc, maar heeft het geen grote, directe invloed op het concept van Haddock.
- Dynamic typing. CleanDoc kan extra visuele nadruk leggen op dynamic typing en annotaties invoeren voor `@wrap` en `@unwrap` voor wrapper en unwrapper functies.
- Generic programming. Hoewel een sterke eigenschap van Clean, hoeft dit geen noemenswaardige impact te hebben op CleanDoc. Wederom is er wellicht extra visuele nadruk mogelijk voor generics.
- Modules. In Clean moet het mogelijk zijn documentatie te plaatsen in zowel declaration als implementation modules.

Verbeteringen

Hoogle, de zoekmachine voor Haskell-code is redelijk simpel om te zetten tot een Clean-variant: “Goocle”. Om een stap verder te gaan, kan Goocle, als onderdeel van de Clean IDE, real-time een database onderhouden van code, waardoor het ook real-time suggesties kan opleveren voor de code die geschreven wordt, zoals bijvoorbeeld de Java

IDE Eclipse dit doet. Dankzij Goocle is ook de standaard zoekfunctie van de Clean IDE uit te breiden.

Een extra functie voor het documentatiesysteem kan zijn om delen van de implementatie te kunnen aanduiden om weer te geven in de documentatie. Soms kan het van belang zijn delen van de implementatie toe te lichten. Dankzij deze toevoeging blijft dergelijke code-documentatie ook makkelijk up to date.

CleanDoc kan binnen de Clean IDE hulp bieden bij het schrijven van documentatie-annotaties. Door bijvoorbeeld entity references en syntax te checken, kunnen fouten in de documentatie voorkomen worden. Dit is goed mogelijk wegens de mogelijkheid tot nauwe integratie van CleanDoc in de Clean IDE.

Door een systeem toe te voegen voor tag-annotaties, wordt aan programmeurs de vrijheid geboden om specifieke indelingen in de code door tags (ook wel labels genoemd) te verzinnen.

Tot slot kan de harde output van CleanDoc, mits dit HTML-bestanden zijn, dankzij JavaScript voorzien worden van allerhande functionaliteit, zoals een interactieve interface en zoekfuncties.

Samenvatting

Clean en Haskell zijn talen die sterk op elkaar lijken. Bovendien zijn de meeste verschillen tussen de talen verschillen op hoog niveau, waardoor ze weinig tot geen invloed hebben op de kern van documentatiesysteem Haddock. Praktisch gezien kan Haddock bijna direct worden gebruikt voor Clean. Voor CleanDoc rest er echter wel de mogelijkheid om unieke eigenschappen te implementeren voor de unieke eigenschappen van Clean, zoals in het bijzonder uniqueness typing.

Ook valt er los van de talen heel wat te verbeteren aan het concept van Haddock, zoals een nauwe integratie in en samenwerking met de Clean IDE. In dit onderzoek zijn een aantal voorbeelden genoemd: code-suggesties, automatische controles voor CleanDoc comments, betere zoekfuncties binnen de Clean IDE, documentatie van implementatiedetails, tags en extra functionaliteit in HTML output.

Hoofdstuk 5 – Conclusie & Discussie

Aan het begin van dit onderzoek was het slechts duidelijk dat er behoefte was aan een documentatiesysteem voor Clean-code. Dit onderwerp heb ik gekozen met hulp van Pieter Koopman en Peter Achten, die me destijds drie mogelijke onderwerpen met betrekking tot Clean aanboden.

Aanvankelijk ben ik begonnen met een vergelijking tussen Clean en Haskell, om op basis van de verschillen tussen de talen te onderzoeken hoe het concept van Haddock omgevormd zou kunnen worden tot een documentatiesysteem voor Clean, genaamd CleanDoc. Hiervoor heb ik vooral veel moeten lezen over Clean en Haskell, omdat ik weinig bekend ben met functioneel programmeren. De onderzoeksvraag was destijds als volgt:

Wat zijn de belangrijkste verschillen tussen Clean en Haskell met betrekking tot de ontwikkeling van een documentatiesysteem voor Clean geïnspireerd door Haddock?

In retrospect bleek deze onderzoeksvraag op zichzelf niet interessant genoeg voor de volledige scriptie. De verschillen tussen Clean en Haskell zijn betrekkelijk klein en de meeste verschillen zijn niet bijzonder van belang in het kader van de gestelde onderzoeksvraag. Toen heb ik in overleg besloten de onderzoeksvraag te verbreden met de volgende toevoeging:

Wat zijn de belangrijkste verschillen tussen Clean en Haskell met betrekking tot de ontwikkeling van een documentatiesysteem voor Clean geïnspireerd door Haddock en hoe kan dit documentatiesysteem verder verbeterd worden?

Dankzij de open toevoeging kon er meer interessants toegevoegd worden aan het onderzoek. Aan de hand van Haddock en een korte studie naar andere documentatiesystemen voor code heb ik toen verschillende verbeterpunten voor CleanDoc kunnen aankaarten. Pieter heeft me toen nog in de richting van Hoogle gestuurd, wat spannende mogelijkheden gaf.

Persoonlijk ben ik tevreden met het resultaat. De vergelijking tussen Clean en Haskell heeft een academisch niveau, ondanks dat dergelijke vergelijkingen al eerder zijn gedaan. De suggesties voor verbeteringen zijn nuttig en kaarten aan welke mogelijkheden er zoal zijn om CleanDoc tot een geavanceerd, state-of-the-art documentatiesysteem te maken.

Er waren zeker struikelblokken tijdens dit onderzoek. Ten eerste kostte het me veel tijd om een goed begrip te krijgen van de werking van Clean en Haskell om een beeld te krijgen van de (soms subtiele) verschillen tussen de talen. Ik had gehoopt dat dit minder tijd zou kosten. Mijn begrip van de functionele talen is nog zeker niet perfect, maar goed genoeg voor het leeuwendeel van dit onderzoek.

Ten tweede viel het me zwaar om de verschillen tussen Clean en Haskell te vertalen tot aanpassingen op het concept van Haddock. In de meeste gevallen voelde het alsof Haddock praktisch voldoende was om te werken voor Clean. Ondanks dat sommige van de voorgestelde aanpassingen niet van wereldniveau zijn, zijn ze voor de volledigheid wel toegevoegd.

Ten derde was de nieuwe toevoeging op de onderzoeksvraag erg breed en onmogelijk

sluitend te beantwoorden. Ik kon slechts suggesties doen voor aanpassingen op basis van de eigenschappen van andere documentatiesystemen. Ik kan natuurlijk niet claimen dat deze lijst compleet is, aangezien het een lijst suggesties is. Er is natuurlijk nog veel meer mogelijk, maar de genoemde verbeteringen geven een goed beeld van wat mogelijk is en tippen enkele punten aan die mij persoonlijk erg relevant leken.

In retrospect was het een lastig onderzoek, met name door de abstracte stof. Het was soms moeilijk om antwoorden te verzinnen. Het was ook zo dat ik van tevoren totaal niet wist in hoeverre er verschillen zouden zijn tussen Clean en Haskell die invloed zouden hebben op de ontwikkeling van CleanDoc. Hier kwam minder uit dan ik gehoopt heb.

In ieder geval is naar voren gekomen dat met weinig moeite een systeem als Haddock is te realiseren voor Clean en dat er veel plek is voor verbetering aan het concept. Het is mijn hoop dat er in de toekomst ook daadwerkelijk gewerkt zal worden aan CleanDoc, omdat ik documentatie als programmeur bijzonder belangrijk vind. Als dit mocht gebeuren, dan wens ik de Clean community veel succes toe en hoop ik dat mijn scriptie nog als inspiratie kan dienen.

Literatuur & Referenties

[1] Sun Microsoystems - Javadoc (http://java.sun.com/j2se/javadoc/)

[2] Simon Marlow 2002 - Haddock, A Haskell Documentation Tool

[3] R. Plasmeijer & M. Van Eekelen 2002 – Clean Language Report 2.1 (http://clean.cs.ru.nl/download/Clean20/doc/CleanLangRep.2.1.pdf)

[4] http://www.haskell.org/haskellwiki/Language_and_library_specification

[5] Peter Achten 2007 - Clean for Haskell98 Programmers

[6] Simon Marlow – Haskell 2010 Language Report

[7] John van Groningen, Thomas van Noort, Peter Achten, Pieter Koopman, and Rinus Plasmeijer 2010 - Exchanging Sources Between Clean and Haskell - A Double-Edged Front End for the Clean Compiler (http://www.cs.ru.nl/~thomas/publications/groj10-exchanging-sources-between.pdf)

[8] Matthew Naylor 2004 – Haskell to Clean Translation

[9] Idoc (http://www.cse.unsw.edu.au/~chak/haskell/idoc/)

[10] Hdoc (http://www.infosun.fim.uni-passau.de/cl/staff/groesslinger/)

[11] Doxygen (http://www.stack.nl/~dimitri/doxygen/)

[12] Hoogle (http://www.haskell.org/haskellwiki/Hoogle)

[13] Eclipse (http://www.eclipse.org/)

[14] jQuery (http://jquery.com/)

[15] Haddock User Guide (http://www.haskell.org/haddock/doc/html/)

[16] N. Mitchell 2008 - Hoogle Overview (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.155.4142&rep=rep1&type=pdf)