Aspecten

|  |  |  |
| --- | --- | --- |
| Joop Ringelberg | 10-04-18 – 11-04-18 | Versie: 2 |

# Introductie: type classes

De functionele talen Purescript en Haskell kennen *type classes*. Een type class definieert een aantal members als types waarin een type class variabele voorkomt[[1]](#footnote-1). Een ADT kan lid van een type class worden door de implementatie te geven van de types van de type class. Members kunnen ook een default implementatie hebben.

Last, but not least, voor een type class kunnen *wetten* opgesteld worden. Dit zijn vergelijkingen (meestal gelijkheden) waaraan instanties moeten voldoen.

Type classes zijn het functionele alternatief voor *inheritance*. Inheritance is de transitiviteit van types. Types zijn niet transitief in de functionele talen, in tegenstelling tot bij de object-georiënteerde talen. Eén en hetzelfde datatype kan wel lid zijn van meerdere type classes. Op deze manier ‘verwerft een datatype *gedrag*’[[2]](#footnote-2), om het in OO-termen te zeggen. Dit is een krachtige vorm van compositie die wel doet denken aan *multiple inheritance*, maar dan zonder de nadelen.

De programmeur gebruikt type classes als constraints op type variabelen van functies. Zo verzekert hij zich ervan dat hij bij de implementatie van een functie gebruik kan maken van de members van de type class.

# Aspecten

Perspectives *Aspecten* zijn geïnspireerd op type classes.

Laten we een beginsel van Perspectives in herinnering nemen: *we modelleren alleen ten bate van Acties*. Een Context, Rol of Property die niet gerelateerd is aan een Actie, is nutteloos (sta erbij stil dat Raadplegen een Actie is). We huldigen dat beginsel in Aspecten door te eisen dat een Aspect minstens één Actie moet bevatten[[3]](#footnote-3). Aspecten zijn compositionele elementen die Acties bijdragen aan Contexten (verderop laat ik zien hoe Aspecten ook Rollen kunnen bijdragen).

## Acties stellen eisen

Een Actie heeft *syntactische rollen*: onderwerp, lijdend voorwerp[[4]](#footnote-4). Deze rollen zijn voorzien van een PropertyEisenPakket. Een *PropertyEisenPakket* is te vergelijken met een View. In essentie is het een opsomming van Properties, maar die kunnen, i.t.t. bij een View, ter plekke gedefinieerd zijn.

De werking van een PropertyEisenPakket is dat elke RolInContext die we willen binden aan een SyntactischeRol de vereiste Properties moet dragen[[5]](#footnote-5). De type checker ziet daarop toe.

## Compositie met Aspecten

We kunnen een Context uitbreiden door er een Aspect aan toe te voegen. Daarmee voegen we dus minstens één Actie toe aan deze Context. Dat heeft alleen zin, als we aangeven welke Rollen in die Context gebonden kunnen worden aan de Syntactische Rollen van de Actie. Deze afbeelding valt onder het begrip *contextualisering*. Zonder contextualisering zou de toegevoegde Actie vrij rondzweven in de Context en totaal nutteloos zijn, want geen enkele Actor zou hem kunnen uitvoeren.

De betreffende RolInContext moet wel de vereiste Properties dragen. De modelleur moet voor elke vereiste Property aangeven welke door de Rol gedragen Property daarmee correspondeert. De type checker ziet er op toe of de aangegeven afbeelding klopt[[6]](#footnote-6).

## Vereiste Rollen

Een Aspect bevat dus Acties. Het is een Context en die kan ook Rollen bevatten. Een Rol in een Aspect heeft alleen zin als hij optreedt in een Actie. Anders gezegd: als we in het Aspect aangeven aan welke Syntactische Rollen die Rol gebonden is.

Een Rol in een Aspect kunnen we ook een PropertyEisenPakket geven. Het PropertyEisenPakket van de Syntactische Rollen van de Acties in het Aspect kunnen we dan uitdrukken in termen van de properties die door de Rollen vereist worden. We nemen dan in het PropertyEisenPakket van de Actie een *verwijzing* op naar een vereiste Property van de Rol.

Dit heeft zin als er meerdere Acties in het Aspect zijn, die overlappende property-eisen stellen. Het verdient dan de voorkeur om zo’n vereiste property maar één keer op te schrijven (bij een Rol) en er in de Acties naar te verwijzen. Denk bijvoorbeeld aan Raadpleeg- en Beheer Acties die op dezelfde properties van hun object werken.

## Compositie met een Aspect met een Vereiste Rol

Contextualisering verloopt net iets anders als een Aspect een Rol vereist. We geven dan aan welke RolInContext correspondeert met de Rol van het Aspect. Daarbij moeten we, net zoals bij het contextualiseren van een Actie, óók voor elke door de AspectRol vereiste Property aangeven welke Property van de RolInContext ermee correspondeert.

## Vereiste Aspecten

Met Aspecten kunnen we Aspecten bouwen op vergelijkbare manier als dat de programmeur type constraints op zijn functies kan gebruiken. Bij de opbouw van een Aspect kunnen we veronderstellen dat de rollen van dat Aspect voldoen aan de eisen die gesteld worden door een *ander Aspect*.

De modelleur geeft de correspondentie aan tussen Rollen en Properties van het vereiste Aspect en het eisende Aspect.

# Aspecten vergeleken met type classes

Net zoals bij een type class, moeten we bij een Aspect datgene wat varieert, zien te hanteren. Als de programmeur een instantie van een type class maakt voor concrete datatypes, vervangt hij de variabelen van de type class door concrete datatypes. Vervolgens moet hij uitprogrammeren hoe die datatypes door de functionele members van de class moeten worden behandeld.

Als de modelleur een Aspect toevoegt aan een concrete Context, moet hij ook datgene wat varieert, vastpinnen op iets concreets. Dat is het contextualiseren. Hij moet

* aangeven welke RolInContext van de Context correspondeert met de vereiste rol van het Aspect;
* aangeven welke Property correspondeert met de door het Aspect vereiste Property.

Zo beschouwd zijn vereiste Rollen en Properties de variabelen van een Aspect. Contextualiseren is een binding van die variabelen aan concrete Rollen en Properties.

Anders dan bij type classes hoeft de modelleur geen implementatie te schrijven van Acties in de concrete Context. De Acties van een Aspect zijn geschreven in termen van operatoren die werken op àlle Rollen, Properties en Contexten. Contextualisering bepaalt op wèlke Rollen, Properties en Contexten ze worden toegepast.

Perspectives is dan ook geen programmeertaal.

## Formalisering van Actie in een Aspect

Een Actie in een Aspect heeft syntactische rollen. Deze formaliseren we als variabelen van de Actie. Een vereiste Property op zo’n rol formaliseren we als een existentieel gekwantificeerde variabele[[7]](#footnote-7). We gebruiken deze variabele om beweringen te doen over de vereiste Property, zoals dat hij functioneel is en een bepaald soort Range heeft.

Een Actie zelf is geen variabele. We beschrijven een Actie onder een bepaalde naam die we toevoegen aan een Context als we het Aspect contextualiseren. De Actie heeft vrije variabelen die we moeten binden aan Rollen en Properties in die Context.

∀ ?object [ ObjectVan(MijnActie, ?object)

∃ ?p [ PropertyVan(?object, ?p) ∧ isFunctioneel(?p) ∧ Range(?p, Number) ]]

Deze regel beschrijft een Actie met de naam MijnActie met één vereiste, functionele numerieke property op zijn objectrol[[8]](#footnote-8). Deze property heeft geen naam, maar wordt gerepresenteerd met een variabele, ?p. De objectrol zelf is eveneens een variabele, gerepresenteerd met de variabele ?object. Bij contextualiseren binden we ?object aan een RolInContext en ?p aan een Property daarvan.

## Formalisering van een Rol in een Aspect

Een Rol in een Aspect representeren we met een variabele. Het PropertyEisenPakket van zo’n Rol formaliseren we op precies dezelfde manier als bij een Aktie.

We kunnen nu echter de propertyvariabelen van de Rol gebruiken bij het beschrijven van de Actie. De Actie heeft dan niet zelf, over de Actie gekwantificeerde, existentiële variabelen.

# Aspecten die bijdragen

We kunnen een Aspect ook een Rol geven die niet zozeer Properties eist, maar bijdraagt. Zulke Properties *moeten* dan voorkomen in het PropertyEisenPakket van een Actie in hetzelfde Aspect, om het eerder genoemde beginsel van Perspectives te huldigen.

De modelleur kan zo’n Aspect op twee manieren contextualiseren:

1. door zo’n Rol over te nemen in een Context, of
2. door zo’n Rol te verbinden met een bestaande RolInContext. De betekenis daarvan is dat deze RolInContext óók de Properties van de AspectRol draagt.

Als de PropertyEisenPakketten van een Actie uit het Aspect volledig ingelost worden door Properties die bijgedragen worden door Rollen van het Aspect, is de contextualisering van zo’n Actie voltooid met het overnemen of afbeelden van de betreffende Rollen van het Aspect.

Een Rol hoeft niet òf bij te dragen, òf te eisen. Een mengvorm is goed mogelijk, waarbij sommige Properties vereist worden en andere bijgedragen. De contextualisering van zo’n Rol vereist dan de afbeelding van de vereiste Properties.

## Formalisering van een Rol die bijdraagt

Een Property die door een Rol van een Aspect bijgedragen wordt, beschrijven we in het Aspect met een naam. We beschrijven de eigenschappen van de Property in relatie tot die naam. Deze naam is géén vrije variabele die gebonden moet worden als we contextualiseren. Het is deze naam die óók door de Rol gedragen wordt (een naam met als namespace het Aspect, niet de Rol van de Context waar het Aspect aan is toegevoegd).

Als een Rol in een Aspect niet met een variabele wordt gerepresenteerd maar met een naam, nemen we die Rol automatisch over als we het Aspect toevoegen aan de Context.

Tenslotte: ook voor een Rol die met een naam in het Aspect is gerepresenteerd, kunnen we existentieel gekwantificeerde Properties beschrijven. Dit is de mengvorm waarbij een Rol zowel toevoegt, als eist.

# Constraints op Rollen

Type checking vindt plaats in *type time*. Maar de modelleur wil soms condities opleggen aan rolbinding[[9]](#footnote-9), die pas in *run time* gecontroleerd kunnen worden. Dergelijke condities hebben onveranderlijk te maken met de *waarde* van Properties. Denk bijvoorbeeld aan een minimum leeftijd[[10]](#footnote-10).

Perspectives ondersteunt dit met een Constraint op een Rol. Een *Constraint* is een berekende Property met een Boolean waarde. In run time moet de waarde van deze berekende Property gelijk zijn aan true (anders verhindert het run time systeem de rolbinding).

Een Rol heeft één Constraint. De Constraint van een vereiste Rol wordt bij contextualiseren automatisch gecombineerd met de Constraint op de Rol waarmee hij vereenzelvigd wordt. De combinatie berekent de logische EN van beide Constraints.

# Compositie van Acties

Twee Acties met matchende PropertyEisenPakketten en matchende Constraints op hun subject rollen, kunnen gecombineerd worden tot één Actie[[11]](#footnote-11).

In het eenvoudigste geval matchen de object rollen van beide Acties óók. Er ontstaat dan een Actie met de gecombineerde effecten van beide. De volgorde van compositie bepaalt de volgorde van de effecten.

Als de object rollen van de Acties niet gelijk zijn, ontstaat een Actie met meerdere object rollen. Mogelijk staan we dat niet toe.

Niet alle Acties kunnen zinvol gecomponeerd worden. Twee Raadpleegt Acties kunnen bijvoorbeeld niet gecomponeerd worden. Een Raadpleegt Actie heeft immers effect op een kanaal en het is niet voorstelbaar hoe deze effecten ‘gecombineerd’ kunnen worden.

# Aspecten voor functies

Een Context kan een *berekende rol* hebben (en een Rol kan een berekende property hebben). Zo’n berekening noemen we een Query. Deze twee *Query*-vormen zijn functies met het volgende algemene type:

Context -> Rol

Rol -> Property

Voor een specifiek type Context, bijvoorbeeld Aangifte, kan een Query met een specifieker type gemaakt worden, zoals dit:

Aangifte -> Aangever

Een Query wordt gecomponeerd uit elementaire stappen. Dit zijn functies van het systeem (primitieve query functies). De type checker van Perspectives ziet erop toe dat de modelleur combinaties maakt waarvan de types kloppen. Daarom zijn ook deze primitieve query functies beschreven in de type taal van Perspectives.

Het type van deze primitieve functies is opgebouwd uit hele elementaire types, zoals Context en Rol. Maar er zijn nog algemenere functies. Een goed voorbeeld is **type**, die op elk type kan werken en er het type van teruggeeft. Een ander voorbeeld is de functie **id**, die de string representatie van de unieke identificatie van een type teruggeeft. In Purescript zouden we beide functies als volgt typeren:

forall a b. type :: a -> b

forall a. id :: a -> String

De systematiek waarmee we in Perspectives types beschrijven, heeft echter geen *type variabele* (wel een type *class* variabele, d.w.z. Aspect rol- of property variabele). Er zijn geen (andere) use cases voor zo’n variabele. We beschrijven daarom deze functies aan de hand van parameterloze type classes, oftewel Aspecten zonder rollen, properties of acties:

type :: Type -> Type

id :: Type -> String

Het Aspect Type is het meest algemene Aspect dat we hebben. Elk type heeft het aspect Type. Op dezelfde manier maken we de Aspecten ContextAspect, RolAspect, PropertyAspect en nog enkele[[12]](#footnote-12).

1. In het algemene geval kunnen nul of meer type class variabelen voorkomen. [↑](#footnote-ref-1)
2. De *functionele members* van de type class vormen ‘het gedrag’. [↑](#footnote-ref-2)
3. Hier is één uitzondering op. Zie het hoofdstuk *Aspecten voor functies*. [↑](#footnote-ref-3)
4. We zullen hier respectievelijk *subject* en *object* gebruiken. [↑](#footnote-ref-4)
5. Zie de tekst *Rollen dragen Properties*. [↑](#footnote-ref-5)
6. Het is goed denkbaar dat we de modelleur ondersteunen bij deze afbeelding door geautomatiseerd naar mogelijke afbeeldingen van vereiste en beschikbare Properties te zoeken. [↑](#footnote-ref-6)
7. Existentieel, want we eisen dat er minstens één property moet bestaan die we eraan kunnen binden. [↑](#footnote-ref-7)
8. De objectrol is universeel gekwantificeerd. Wat we ook binden aan de objectrol, de bewering die volgt moet ervoor gelden. [↑](#footnote-ref-8)
9. Er lijken geen andere use cases te bestaan. [↑](#footnote-ref-9)
10. Perspectives zal een faciliteit bieden om in type time beperkingen op te leggen aan waarden van Properties. Dergelijke beperkingen heten *facetten*. Daarmee verplaatsen we een deel van deze problematiek naar type time, maar de noodzaak voor run time constraints nemen we er niet geheel mee weg. [↑](#footnote-ref-10)
11. Zie de tekst *Type Checking* voor een definitie van ‘matchen’. [↑](#footnote-ref-11)
12. We kunnen niet de naam “Context” gebruiken voor een Aspect, omdat we al een type met die naam hebben. Types en Aspecten gebruiken dezelfde namespaces. [↑](#footnote-ref-12)