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.

Overigens kan een type class variabele onderhevig zijn aan een *type class constraint*. Hierdoor wordt de ene type class een specialisatie van één of meer andere.

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 het gewone gebruik van een View (bij een Rol), ter plekke gedefinieerd zijn[[5]](#footnote-5).

De werking van een PropertyEisenPakket is dat elke RolInContext die we willen binden aan een SyntactischeRol de vereiste Properties moet dragen[[6]](#footnote-6). 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 erop toe of de aangegeven afbeelding klopt[[7]](#footnote-7).

## Rollen van Aspecten

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 Properties geven. Het PropertyEisenPakket van de Syntactische Rollen van de Acties in het Aspect kunnen we dan uitdrukken in termen van de Properties van de Rollen. 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.

## Rollen toevoegen of afbeelden

De modelleur kan een Rol van een 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.

## Properties toevoegen of afbeelden

Als we een Rol A verbinden met een bestaande RolInContext R, moeten we voor elke Property van de aspectrol A bepalen of we die toevoegen aan R, of afbeelden op een property van R.

We kunnen afbeelden op een lokale property van R, of op een property van de mogelijkeBinding van R.

# Rolbinding

We binden een Rol aan een Rol of een Context (waarbij we eigenlijk binden aan de BuitenRol van die Context). Als een Rol a gebonden is aan een Rol b, kunnen we bij a elke Property van b opvragen alsof het een Property van a was.

Op type-niveau, als Rol B de mogelijkeBinding is van Rol A, wordt elke Property die gedragen wordt door B, óók gedragen door A.

Het is alsof B gecontextualiseerd is in A!

Beschouwen we een Rol als een Product van Properties, dan zien we dat binding van Rollen precies voldoet aan de definitie van Aspect-zijn voor Product types.

Kortom, als Rol B de mogelijkeBinding is van Rol A, is B een Aspect van A.

Er zijn twee verschillen met het contextualiseren van een Rol X in A (d.w.z. dat we X als Aspect toevoegen aan A):

1. run time is de waarde van een Property van X gerepresenteerd bij de instantie van A, terwijl de waarde van een Property van B gerepresenteerd is bij de instantie van B;
2. het is niet mogelijk om een Property van B af te beelden op een Property van A. Als B een PropertyEisenPakket heeft, kunnen we daar niets mee doen op het moment dat we B tot mogelijkeBinding van A verklaren. Met andere woorden: alléén de Properties die B bijdraagt, zijn beschikbaar voor instanties van A.

# Contextualisering

Als we een Aspect A toevoegen aan een Context C, *contextualiseren* we A in C. We doen dat door A als aspect op te nemen bij C:

psp:Context usr:C

psp:aspect usr:A

## Rollen contextualiseren

Voor elke Rol Ra van A hebben we twee keuzes:

1. we voegen Ra toe aan C, of
2. we beelden Ra af op een rol Rc van C.

Een toevoeging vereist geen verdere representatie. Dat betekent dat een query die de voor C gedefinieerde Rollen opzoekt, recursief de aspecten van C moet aflopen op zoek naar rollen die niet zijn afgebeeld, bij contextualisering. Merk op dat deze rollen de namespace van hun Aspect hebben.

Afbeelden doen we als volgt:

psp:Context usr:C

psp:aspect usr:A

psp:rolInContext =>

psp:Rol $Rc

psp:aspectRol => usr:A$Ra

Als Rc de rol Ra als aspectrol heeft, is Ra afgebeeld op Rc.

Merk op dat we de rol Rc verder properties mogen geven zoals we willen. Ook mogen we de mogelijkeBinding bepalen, maar daar zijn we wel beperkt tot types die een specialisatie zijn van de mogelijkeBinding van de aspectrol Ra.

We mogen meerdere rollen (uit evenzovele Aspecten) afbeelden op Rc. Daarbij geldt dat de mogelijkeBinding van Ra beperkt wordt tot de doorsnede van de types van de mogelijke bindingen van de aspectrollen.

## Properties contextualiseren

Voor een Property van een gecontextualiseerde Rol hebben we dezelfde keuzes als voor een Rol zelf: toevoegen of afbeelden. Voor het afbeelden gebruiken we de rol psp:aspectProperty in de lokale definitie van de Property. Ook hier hebben we de mogelijkheid om properties uit meerdere Aspecten te contextualiseren naar één en dezelfde property van een Rol van Context C.

Maar een aspectproperty kunnen we ook verbinden met een Property van de mogelijkeBinding van een Rol. Oftewel: een Property Pra van de rol Ra van aspect A kunnen we afbeelden op een Property van de mogelijkeBinding van de Rc van de Context. We doen dat met de rol psp:bindingProperty:

psp:Context usr:C

psp:aspect usr:A

psp:rolInContext =>

psp:Rol $Rc

psp:aspectRol => usr:A$Ra

psp:rolProperty =>

psp:Property $P

psp:aspectProperty usr:A$Ra$Pra

psp:bindingProperty usr:B$P

psp:mogelijkeBinding => usr:B

Als we een getter construeren voor property usr:C$Rc$P, zien we aan het gebruik van psp:bindingProperty dat we de waarde op moeten halen uit de rolDAG, niet bij de rol zelf als lokaal gerepresenteerde property.

# Constraints op Rollen

Type checking vindt plaats in *type time*. Maar de modelleur wil soms condities opleggen aan rolbinding[[8]](#footnote-8), 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[[9]](#footnote-9).

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[[10]](#footnote-10).

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.

# Vereiste Aspecten

Met Aspecten kunnen we Aspecten bouwen op vergelijkbare manier als bij type classes mogelijk is door type class constraints in te zetten. 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[[11]](#footnote-11). 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[[12]](#footnote-12). 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. De Properties 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.

## Vereiste Aspecten: type class constraints en rdfs:subClassOf

Een Vereist Aspect is goed te vergelijken met een *type class constraint*. Deze laatste vereist dat een type class variable óók in een andere type class valt. Precies hetzelfde geldt voor een Vereist Aspect. In de tekst *Type Checking* werk ik de relatie *heeftAspect* uit. Deze relatie blijkt transitief te zijn en daarmee is heeftAspect goed te vergelijken met de relatie subClassOf van RDF Schema.

Bij rolbinding aan een rol a kijken we of de te binden rol b wel een instantie is van T, waar T de mogelijkeBinding is van A. Hier kunnen we vergelijken met rdf:type. b staat in de relatie rdf:type tot T als B gelijk is aan T, of als B een rdfs:subClassOf is van T.

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. Een View is opgebouwd uit PropertyReferenties. Een PropertyReferentie heeft als mogelijkeBinding een Property. Dit màg een ter plekke gedefinieerde Property zijn. [↑](#footnote-ref-5)
6. Zie de tekst *Rollen dragen Properties*. [↑](#footnote-ref-6)
7. 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-7)
8. Er lijken geen andere use cases te bestaan. [↑](#footnote-ref-8)
9. 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-9)
10. Zie de tekst *Type Checking* voor een definitie van ‘matchen’. [↑](#footnote-ref-10)
11. Existentieel, want we eisen dat er minstens één property moet bestaan die we eraan kunnen binden. [↑](#footnote-ref-11)
12. De objectrol is universeel gekwantificeerd. Wat we ook binden aan de objectrol, de bewering die volgt moet ervoor gelden. [↑](#footnote-ref-12)