Type checking

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

# Introductie

Er zijn drie bekende use cases waarbij het vergelijken van types een belangrijke rol speelt:

* bij rolbinding;
* bij het opbouwen van een query;
* bij contextualiseren van een Aspect.

Bij type checking in Purescript spelen type classes een grote rol. In Perspectives is die rol vervangen door Aspecten. De type checker onderzoekt de vorm van de types, en of het ene type het andere *als Aspect heeft*.

# Rolbinding

Bij rolbinding willen we een instantie b van een Rol B binden aan een instantie a van een Rol A. We zeggen dat b de *binding* wordt van a. a heeft vóór de rolbinding geen binding, erná wel.

Merk op dat b al dan niet zelf een binding kan hebben. Als b een instantie is van een *primitieve rol* zal hij nooit een binding krijgen; anders kan die binding later aangelegd worden.

Merk ook op dat A geen primitieve rol kan zijn, want dan kunnen we er niet aan binden. Daarom is A een combinatie, een product, van een rol en een ander type T. T is de waarde van de mogelijkeBinding van de beschrijving van de Rol A.

De type checker controleert of het type B gelijk is aan T of T als Aspect heeft.

# Query opbouwen

Een Query komt in Perspectives op drie plaatsen voor, namelijk als berekende Rol, berekende Property en in een query-applicatie expressie waarmee we een rol een binding geven. In het eerste geval correspondeert een Query een functie van Context naar Rol en in het tweede geval een functie van Rol naar Property[[1]](#footnote-1). Deze correspondentie wordt uitgewerkt in de implementatie van Perspectives, waar de Query expressie omgezet wordt in een Purescript functie. Deze uitwerking gebeurt in drie stappen:

1. de in Arc syntax geschreven expressie wordt ontleed en gerepresenteerd als een Purescript ADT;
2. deze abstracte syntax tree wordt gebruikt om de *beschrijving* van een compositie van Purescript functies op te zetten;
3. de beschrijving wordt omgezet in executeerbare code.

De type checker controleert of de beschreven functies wel gecomponeerd kunnen worden. Elke functie heeft het type forall a, b. a -> b (het zijn functies van één argument). De *range* b van functie n van de compositie moet gelijk zijn aan het *domain* a van functie n + 1, of het als Aspect hebben. Zie het hoofdstuk *Types van Functies* voor een uitgebreidere beschrijving.

# Contextualiseren

Contextualiseren is de vrije variabelen van een Aspect binden aan concrete Rollen of Properties van een Context.

Een property-variabele doet zich voor als een Property-beschrijving (in termen van contexten en rollen) die onderdeel is van een PropertyEisenPakket (van een RolInContext van het Aspect, of van een syntactische rol van een Actie van het Aspect).

Deze Property-beschrijving moet een Aspect zijn van, of gelijk zijn aan, een concrete Property. Oftewel, de concrete Property van de Context die we willen binden aan de property-variabele, moet de Property (beschrijving) die bij die variabele hoort als Aspect hebben (of eraan gelijk zijn).

Hetzelfde geldt, mutatis mutandis, voor een Rol-variabele.

Als we een Aspect in een Context hebben gecontextualiseerd, is die Context een *instantie* van het Aspect geworden. We zeggen ook wel dat de Context het Aspect heeft *opgenomen*, of dat de Context met het Aspect is *gecomponeerd*.

# De relatie heeftAspect

Intuïtief heeft S T als Aspect als S *gespecialiseerder* is dan T. Een andere intuïtie over heeftAspect is dat een i die een instantie is van S, óók een instantie van T moet zijn. Omgekeerd geldt dat niet.

De omgekeerde van heeftAspect is *isAspectVan*. Beide verschillen van *gelijk.*

In dit hoofdstuk definiëren we precies wat het betekent dat S T als Aspect heeft.

Nota Bene. De relatie heeftAspect is te vergelijken met de relatie rdfs:subClassOf. De relatie psp:type is te vergelijken met rdf:type.

## heeftAspect voor SimpleValue met facetten

SimpleValue is een Perspectives *kind* waarvan maar vier instanties zijn: String, Number, Date en Boolean. Elk van deze vier is dus een type. Er is geen heeftAspect relatie tussen één van hen (en ze zijn uiteraard allemaal verschillend van elkaar).

Een *facet* definieert een deelverzameling van de mogelijke waarden van een SimpleValue.

Een voorbeeld van een facet op Number is: > 10. Dit facet definieert alle getallen die groter zijn dan tien. De waarden van het type Number > 10 is een deelverzameling van de waarden van het type Number. Per definitie is daarom Number > 10 gespecialiseerder dan Number, oftewel: het heeft Number als Aspect.

Met dezelfde redenering zie je dat Number > 11 gespecialiseerder is dan Number > 10. Op deze wijze kunnen we types die een combinatie zijn van een SimpleValue en één of meer facetten, onderling ordenen met de heeftAspect relatie.

Een FacettedValue is het Product van een SimpleValue met een Facet. De heeftAspect relaties tussen FacettedValues gelden alleen voor twee van deze types die hetzelfde SimpleValue deel hebben. Verder wordt de relatie bepaald door de relatie van beider Facetten. Deze relatie, over de mogelijke Facetten per SimpleValue, wordt in een andere tekst uitgewerkt.

## heeftAspect voor een Product type

Een Product type is een combinatie van typen.

Een Product type S heeft een Product type T als Aspect, als elk onderdeel van T gelijk is aan een onderdeel van S, of er een Aspect van is. Merk op dat S meer onderdelen mag hebben dan T.

## heeftAspect voor een Sum type

Een Sum type is een combinatie van alternatieven, waar elk alternatief een type is. Een instantie van een Sum type is altijd een instantie van één van zijn alternatieve types.

Een Sum type S is heeft een Sum type T als Aspect, als alle alternatieven van S gelijk zijn aan een alternatief van T of dat alternatief als Aspect hebben.

## heeftAspect voor Property

Een Property beschrijven we met een context met één rol en twee externe property waarden. Zo’n context is isomorf met een ADT met drie delen:

data Property = Property Functioneel Verplicht SimpleValue

data Functioneel = Functioneel | NietFunctioneel

data Verplicht = Verplicht | NietVerplicht

data SimpleValue = String | Number | Date | Boolean

Dit is een Product type, dus we kunnen de definitie van heeftAspect voor Product types toepassen op Property. We moeten dus onderdeel voor onderdeel vergelijken.

Een S en een T hebben beide een waarde voor Verplicht. We zeggen dat Verplicht NietVerplicht als Aspect heeft. Dus als T NietVerplicht heeft en S Verplicht en de andere twee onderdelen zijn gelijk, heeft S het Aspect T.

Op dezelfde wijze is Niet-Functioneel een Aspect van Functioneel.

Voor SimpleValue kunnen we alleen gelijkheid of ongelijkheid vaststellen. Zodra Perspectives Facetten ondersteunt, vervangen we SimpleValue door FacettedValue. Deze types hebben wel interessante heeftAspect relaties.

## heeftAspect voor Context

De definitie van een bepaald type context, zoals Aangifte, representeren we – net zoals Property - met een context met rollen – dus een *instantie*. Intuïtief voelen we dat zo’n context isomorf moet zijn met een Record dat voor elke rol een veld heeft. Een Record is een Product type, zodat we ook voor definities van contexten de heeftAspect definitie van Product types kunnen toepassen. Maar anders dan bij een Property is het aantal rollen van een beschrijving van een context niet vast, maar variabel. De isomorfie is daarom minder vanzelfsprekend.

Een instantie van Context – de definitie van een bepaald type context – wordt in Purescript gerepresenteerd met een value van de ADT PerspectContext. We moeten dus laten zien dat PerspectContext in zijn algemeenheid isomorf is met een Product van types.

PerspectContext is een newtype van een Record. Voor onze doeleinden gaat het vooral om de rollen, die gerepresenteerd zijn met de velden binnenRol, buitenRol (beiden gebonden aan een RolID) en rolInContext.

Een Record is een Product type. So far, so good. Maar het veld rolInContext heeft zelf een structuur, want heeft het type StrMap (Array RolID). Nu is een StrMap a isomorf met een lijst van Tuple String a. Uit zo’n lijst kunnen we een Record bouwen waarbij de veldnamen de Strings van de Tuples zijn. Het type van elk veld is hetzelfde, namelijk Array RolID. We kunnen uit zo’n Record ook weer de lijst van Tuples produceren[[2]](#footnote-2). Kortom, het veld rolInContext is isomorf met een Record. Daar kunnen we de twee velden binnenRol en buitenRol aan toevoegen. Oftewel, elke instantie van PerspectContext is isomorf met één enkel record, met een unieke collectie velden (oftewel een uniek Product Type).

We nemen aan: de definitie van een context is gelijk aan een Product type, *up to isomorfism*.

Alle velden van zo’n record hebben een waarde die een type representeert (een type beschrijving, dus een context, c.q. de ID daarvan). We moeten dus deze typen vergelijken, als we een S en T vergelijken.

In de praktijk zullen we heeftAspect voor Context implementeren als een functie die de binnen- en buitenrollen van S en T vergelijkt, en dan voor elke sleutel in T de in beide StrMaps opgeslagen typen vergelijkt.

Nog concreter: T kan een Context met één Rol zijn. Dat doet zich voor als we een rolgetter construeren tijdens query-opbouw. Denk aan een functie die de rol Aangever wil teruggeven. Dat is een functie met als domein elke Context die de rol Aangever heeft. Oftewel, een Product type van één lid, Aangever. Kortom, hier komt de uitwerking van ‘heeftAspect’ erop neer dat we nagaan of S de rol Aangever heeft.

Op dezelfde manier kunnen we het nagaan of een rol een Property heeft, beschouwen als een ‘heeftAspect’ test.

## heeftAspect voor Rol

Ook een type rol wordt gerepresenteerd met een context, d.w.z. een instantie van de ADT PerspectContext. We volgen daarom dezelfde strategie als bij Context.

Daarbij kan het onderdeel mogelijkeBinding een Sum type als waarde hebben. Om vast te stellen of T een Aspect is van S, moeten we dus onderzoeken of het ene Sum type een Aspect is van het andere Sum type.

# Gelijkheid van van types en identificatie

Perspectives heeft een systeem van namespacing waardoor de modelleur elk type een unieke naam geeft. Daarom geldt bij het onderzoek of twee types aan elkaar gelijk zijn, dat als ze dezelfde naam hebben, we direct weten dat ze gelijk zijn.

Het omgekeerde geldt niet. Twee samengestelde typen kunnen geheel gelijk zijn opgebouwd, uit dezelfde onderdelen, en toch verschillende namen dragen.

De use case voor onderzoek naar gelijkheid van types is beperkt tot het onderzoek naar heeftAspect van samengestelde typen, waar geldt dat corresponderende onderdelen in de juiste heeftAspect relatie tot elkaar moeten staan *of aan elkaar gelijk moeten zijn.* De diverse algoritmes zullen altijd eerst gelijkheid van de onderdelen testen, omdat de vergelijking van namen zo snel en goedkoop is. Pas als deze test faalt, wordt de samenstelling van de onderdelen onderzocht (op gelijkheid of heeftAspect).

Een andere use case zou een discovery service zijn, waarmee een modelleur kan nagaan of een type al gedefinieerd is, of dat een door hem gemaakt type te componeren zou zijn uit andere, reeds bestaande types.

# heeftAspect is transitief

De definitie van heeftAspect die we bij Perspectives hanteren, is transitief. Dat wil zeggen dat als S T als Aspect heeft, en T heeft ST als Aspect, dan heeft S ST als Aspect. Immers:

* een Product type S heeft Product type T als Aspect, als elk onderdeel van T een Aspect is van (of gelijk is aan) een onderdeel van S;  
  T heeft ST als Aspect, als elk onderdeel van ST een Aspect van een onderdeel van T is of eraan gelijk is;  
  dan móet elk onderdeel van ST een Aspect van een onderdeel van S zijn (of eraan gelijk zijn).
* Een Sum type S heeft een Sum type T als Aspect, als alle alternatieven van S een alternatief van T als Aspect hebben of eraan gelijk zijn;  
  T heeft ST als Aspect, als alle alternatieven van T een alternatief van ST als Aspect hebben of eraan gelijk zijn;  
  dan móet elk alternatief van S een alternatief van ST als Aspect hebben of eraan gelijk zijn.

Voor de facetten geldt ook transitiviteit (in een andere tekst uitgewerkt).

Kortom, voor Contexten, Rollen, enz. geldt dat heeftAspect transitief is.

Om misverstanden te voorkomen: *psp:type is* ***niet*** *transitief!*

# Een beschreven hiërarchie van types

Het lege Product type (Product van nul onderdelen) is per definitie van ‘heeftAspect-voor-Product-types’ een Aspect van elk ander Product type. Omdat het een Aspect is van elk lid van een Sum, is het ook een Aspect van elk Sum type. Kortom, het is het meest algemene type[[3]](#footnote-3).

In Perspectives *beschrijven* we elk type met een Product type (n.l. een instantie van Context) (en dat is niet hetzelfde als de bewering dat elk type een Product type is! Ook een Sum type beschrijven we met een Product type).

We kunnen daarom een hiërarchie van lege types declareren, die volkomen binnen de grenzen van de definitie van heeftAspect blijft. Zo’n hiërarchie komt niet tot stand door structurele vergelijking van types, maar op basis van de relatie psp:aspect, waarmee we aangeven dat een type mede bestaat uit een Aspect. We doen dat voor simple values, die we niet van elkaar kunnen onderscheiden in termen van hun interne structuur:

psp:SimpleValue

psp:Number

psp:String

psp:Boolean

psp:Date

Deze hiërarchie gebruiken we om de functies te beschrijven waarmee we Queries samenstellen.

# Types van functies

We beschrijven eerst functies met één parameter. Zo’n functie beschrijven we als een Context met een rol *domain* en een rol *range*.

Speciale aandacht verdient een aantal elementaire functies:

identity :: forall a. a -> Array a

contextType :: PerspectContext -> ID

rolType :: PerspectRol -> ID

rolTypen :: PerspectContext -> Array ID

iedereRolInContext :: PerspectContext -> Array PerspectRol

binding :: PerspectRol -> PerspectRol

context :: PerspectRol -> PerspectContext

label :: PerspectContext -> String

Uit deze functies genereren we functies die op zichzelf een Query zijn en ook kunnen worden samengesteld tot grotere Queries. Ze accepteren ID's als parameterwaarde en geven Arrays van IDs of Arrays van Strings (die simpele waarden als numbers, dates, booleans en strings representeren) terug (waar ID's strings zijn).

Als we die strings interpreteren en de IDs dereferen, zien we functies met PerspectContext of PerspectRol als argument en Arrays van PerspectContext, PerspectRol, Strings, Booleans, Dates en Numbers als resultaat.

Hierboven heb ik laten zien hoe we de simple values representeren. Maar hoe beschrijven PerspectContext en PerspectRol? Oftewel, hoe typeren we de verzameling values die geconstrueerd zijn met de data constructor PerspectContext?

In de tekst *De anatomie van het type systeem* laat ik zien dat een context de typebeschrijving van een Context is, als hij het type psp:Context heeft, of psp:Context als Aspect heeft. We representeren daarom de PerspectContext data values in het Perspectives type systeem met psp:Context.

Met een vergelijkbare argumentatie representeren we PerspectRol data values in het Perspectives type systeem met psp:Rol.

## Functies met meerdere parameters

We noteren het type van een functie met één parameter als een context met twee rollen, domain en range. Een functie met twéé parameters representeren we met een extra rol.

In Purescript zouden we beiden als volgt noteren:

f :: forall a, b. a -> b

f :: forall a, b, c. a -> b -> c

De tweede functie kunnen we ook zo opschrijven:

f :: forall a, b, c. a -> (b -> c)

Hiermee benadrukken we dat een partiële toepassing van f op één argument, een functie van één parameter oplevert.

We kunnen dat niet op dezelfde wijze noteren in Perspectives. Omdat rolbinding een rol met naam van een binding voorziet, kunnen we de 'functie' in willekeurige volgorde van argumenten voorzien. We hebben geen notatie voor het type van een partieel toegepaste functie en dat is ook niet nodig.

## Voorbeelden: contextType en context

De functie contextType geeft het type van elke instantie van Context en kan gebruikt worden in een Query. Een dergelijke functie zou in Purescript of Haskell niet kunnen bestaan, omdat we types niet kunnen reflecteren in runtime. Maar we zouden zijn type als volgt kunnen benaderen, met twee type class constraints:

contextType :: Context a => Type b => a -> b

In Perspectives beschrijven we het type van contextType als een functie met een domain en range:

psp:Function psp:contextType

$domain => psp:Context

$range => <Type>

Maar wat vullen we in voor <Type>? We hebben geen representatie in het type systeem van Perspectives voor "Type". We weten wel dat een type onveranderlijk met een Context gerepresenteerd wordt. Maar niet elke Context is een Type!

Als we toch in plaats van 'Type' psp:Context zouden gebruiken, zouden we dus generaliseren. Gelukkig er zijn geen functies die alleen op een type-beschrijving werken (als argument vereisen). En functies die op een Context werken, werken natuurlijk óók op een type.

De generalisatie levert dus geen problemen op. Kortom, met onderstaande beschrijving kunnen we uit de voeten:

psp:Function psp:contextType

$domain => psp:Context

$range => psp:Context

Het type van de functie context beschrijven we in Perspectives zo:

psp:Function psp:context

$domain => psp:Rol

$range => psp:Context

Beschouw nu deze Query:

context >-> contextType

(de functie contextType volgt de functie context) Bij het opbouwen van deze Query controleert de type checker of de range van context een subType is van het domain van contextType (of eraan gelijk is).

Dus: heeft Context Context als Aspect of zijn ze gelijk? Ze zijn gelijk, dus in deze volgorde mogen deze functies tot een Query worden gecomponeerd.

Andersom niet. Want dan zou de range van contextType een aspect-hebber moeten zijn van het domain van context (of eraan gelijk zijn). Dus: dan zou Context een Aspect moeten zijn van Rol (of eraan gelijk zijn) En dat is natuurlijk niet zo.

1. Zowel Rollen als Properties kunnen relationeel zijn gedefinieerd. De Purescript functie waar we het over hebben, heeft dan ook een Array als resultaat. [↑](#footnote-ref-1)
2. Hier ben ik niet zeker van. [↑](#footnote-ref-2)
3. In Haskell en Purescript zou dit Unit zijn. [↑](#footnote-ref-3)