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.

# 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.

De type checker controleert of het type B een gelijk is aan T of een subtype is van T.

# Query opbouwen

Een Query komt in Perspectives op twee plaatsen voor, namelijk als berekende Rol en als berekende Property. 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 er een subtype van zijn.

# Contextualiseren

Bij contextualiseren willen we een vrije variabele van een Aspect binden aan een concrete Rol of Property 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 supertype 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 een subtype zijn van de Property (beschrijving) die bij die variabele hoort (of eraan gelijk zijn).

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

# De relatie subtype

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

De omgekeerde van subtype is *supertype*. Beide verschillen van *gelijk.*

In dit hoofdstuk definiëren we precies wat het betekent dat S een subtype is van T.

## Subtype 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 subtype 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 is een subtype van Number.

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 subtype relatie.

Een FacettedValue is het Product van een SimpleValue met een Facet. De subtype 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.

## Subtype voor een Product type

Een Product type is een combinatie van typen.

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

## Subtype 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 een subtype van een Sum type T, als alle alternatieven van S gelijk zijn aan of een subtype zijn van een alternatief van T.

## Subtype 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 subtype 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 noemen Verplicht een subtype van NietVerplicht. Dus als T NietVerplicht heeft en S Verplicht en de andere twee onderdelen zijn gelijk, is S een subtype van T.

Op dezelfde wijze is Functioneel een subtype van NietFunctioneel.

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

## Subtype 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 subtype 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 subtype 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.

## Subtype 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 S een subtype is van T, moeten we dus onderzoeken of het ene Sum type een subtype 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 subtype van samengestelde typen, waar geldt dat corresponderende onderdelen in de juiste subtype 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 subtype).

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.

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)