Perspectives types als Purescript types

|  |  |  |
| --- | --- | --- |
| Joop Ringelberg | 03-02-19 | Versie: 2 |

# Introductie

In de ‘hogere’ modules van de Perspectives implementatie komen we queries tegen. Deze queries zijn gedefinieerd als combinaties van ObjectGetters, of van TripleGetters (deze laatste memoriseren resultaten in de dependency tracking cache). Een (basis) ObjectGetter beeldt een Context af op Rollen, of een Rol op Propertywaarden. Overigens, een Getter heeft met de *identifiers* van Contexten en Rollen als parameter of resultaat; nooit de representaties zelf.

Bij het combineren van ObjectGetters moeten we goed letten op de Perspectives-types van de zaken waarnaar die identifiers verwijzen. Zo moeten we fouten voorkomen, zoals deze: *neem van een Rol een propertyReferentie*.

Echter, de ‘lagere’ modules van Perspectives representeren die identifiers als niet-getypeerde Purescript Strings. Of een identifier nu een Context, Rol of waarde representeert – het is en blijft een String voor Purescript. Dat betekent dat de Purescript compiler ons niet helpt bij de compositie van ObjectGetters tot queries.

Bij complexere queries blijkt dat je snel type-fouten maakt. Daarom representeren we Perspectives types tot op zekere hoogte als Purescript types.

# Wat we van elkaar onderscheiden

Perspectives is een modelleertaal waarin we onbeperkt veel types kunnen beschrijven. De Purescript implementatie van Perspectives daarentegen is gesloten en eindig: we kunnen er slechts een beperkt aantal Perspectives types in representeren. De keuze is pragmatisch: het gaat om de ondersteuning van de programmeur(s) van Perspectives.

In Perspectives komen we alleen queries tegen die zijn opgesteld in termen van één de volgende basismodellen:

* model:Perspectives
* model:QueryAst

We representeren dus alleen de types die we in deze modellen tegenkomen.

## Contexten

Elke Context die we in Purescript code kunnen tegenkomen, heeft dus een type dat gemodelleerd is in één van de basismodellen. In termen van de data structuur PerspectContext: bij pspType mogen we de identifiers verwachten van alle contexten[[1]](#footnote-1) waarvan het type psp:Context is, of psp:Context als Aspect heeft[[2]](#footnote-2). Voor elk van de gedefinieerde typen Contexten in de basismodellen hebben we een Purescript newtype: View, Actie, Zaak, enz., maar ook TrustedCluster (type Zaak) en PBool (type SimpleValue). Eén van deze types is psp:Context zelf. In Purescript is psp:Context afgebeeld op ContextDef (‘de definitie van Context’). Het type ‘Context’ bestaat echter niet.

Bovendien hebben we een Class waarvan alle Purescript typen die een type Perspectives Context uit één van de basismodellen representeren, een instantie zijn. Een functie die werkt op een willekeurig Purescript Context type, geven we een type parameter die onderhevig is aan de beperking dat het een instantie van de Class ContextType moet zijn.

Maar wat typeren we met deze Purescript types? Welnu, de identifiers waarmee we Contexten identificeren. Oftewel: het id\_ veld van PerspectContext (maar zie “Waar maken we onderscheid?”).

## Rollen

Er zijn veel meer Rollen dan Contexten in de modellen. Maar veel Rollen zijn nauwelijks van elkaar onderscheiden, omdat ze geen eigen Properties hebben. Om die reden maken we in Purescript alleen een type voor een Rol, als die Rol eigen Properties heeft.

Met behulp van dat type definiëren we dan het type van de functie die zo’n Property oplevert. Bijvoorbeeld:

getVolgnummer :: propertyReferentie ~~> PNumber

propertyReferentie maken we in Purescript een instantie van de type class RolType (analoog aan ContextType). We hebben nu ook een functie nodig die een propertyReferentie oplevert:

getPropertyReferentie :: View ~~> propertyReferentie

Hier is View een instantie van ContextType en propertyReferentie een instantie van RolType.

We zijn altijd ook geïnteresseerd in de mogelijke binding van een Rol. Daarom maken we voor elke Rol een specifieke functie die die binding ophaalt. Daarbij behandelen we een Rol met Properties anders dan een Rol zonder properties. Met Properties:

propertyReferentie\_Property :: propertyReferentie ~~> Property

Zonder properties:

property\_Range :: Property ~~> SimpleValue

In het eerste geval nemen we van de View (een ContextType) eerst zijn propertyReferentie(s) (een RolType) en dan daarvan de binding (en dat is een Property). In het tweede geval nemen we van de Property (een ContextType) direct de binding van de rol range (en dat is een SimpleValue). We slaan de tussenliggende rol over omdat die geen eigen properties heeft die we kunnen opvragen.

Wat als we willen weten of de rol propertyReferentie functioneel is? propertyReferentie is een instantie van RolType en we hebben een algemene functie isFunctioneel:

isFunctioneel :: forall rt e. RolType rt => (rt ~~> PBool) e

Deze functie kunnen we toepassen op propertyReferentie.

En als we nu willen weten of range functioneel is? We hebben geen functie die van een Property de rol range oplevert, dus we kunnen isFunctioneel niet gebruiken. We kunnen een functie property\_Range\_isFunctioneel maken en die toepassen op Property. Dit is de body van die functie:

getUnqualifiedRol “range” >>> getUnqualifiedProperty “isFunctioneel”[[3]](#footnote-3)

# Waar maken we onderscheid?

Boven schreef ik dat de member “\_id” van PerspectContext (en PerspectRol) met een Purescript-representatie van Perspectives-types getypeerd wordt. Maar dat is onjuist. Het is contraproductief om op dit lage niveau types te onderscheiden. De typering in Perspectives termen wordt pas nuttig op het niveau van ObjectsGetters.

Sterker nog, als we b.v. PerspectContext typeren, komen we in problemen met de toestand van MonadPerspectives. Daarin houden we namelijk een cache bij van al uit de database gehaalde PerspectContexen. Die cache is een StrMap. Maar een StrMap heeft weliswaar een type-parameter, maar die kan – voor een bepaalde instantie – maar aan één type gebonden worden. Dus als we PerspectContext parametriseren met een type variabele, bindt de compiler daar op enig moment één bepaalde Purescript afbeelding van een Perspectives aan – en andere soorten PerspectContexten komen er niet meer in.

Kortom, op laag niveau houden we het op Strings.

Pas een ObjectsGetter zelf gaan we typeren – en maken er daarmee een TypedObjectsGetter van. We noteren dat met een infix-operator op type niveau:

contextType :: forall t e. ContextType t => (Context ~~> t) e

Maar hoe krijgen we dat voor elkaar? Met unsafeCoerce. Dat is simpelweg een opdracht aan de Purescript compiler om maar aan te nemen dat ons type klopt. We gebruiken daar een alias voor: typeWithPerspectivesTypes.

## De compiler heeft het soms moeilijk

We combineren TypedObjectsGetters met een infix operator: /~/. Deze operator werkt met een ‘tussentype’: gegeven twee getters f :: (a ~~> b) en g :: (b ~~> c) construeren we een getter (a ~~> c). Maar stel dat we een compositie f /~/ g maken, en dat

* f een type heeft met als resultaat een variabele o en
* g een type heeft met als argument een variabele s,

dan ontstaat een probleem. Dan klaagt de compiler over een type dat hij gedetecteerd heeft maar wat niet in de body van het type voorkomt. En over dat type wil hij meer weten.

Zelfs als beide variabelen gevuld moeten worden met een instantie van dezelfde class, is het niet goed. De compiler kan namelijk niet bepalen wèlke instantie. Je **moet** dan meer informatie geven over die variabele:

1. zeggen om welk type het gaat, of
2. een variabele in het type van de compositie opnemen en die beperken tot een class die gelijk is aan of een subClass is van beide classes (dus ‘ervoor uitkomen’ dat er een type variabele is in de compositie).

Daarbij kun je kiezen voor elk van beide kanten van de compositie. Dus je moet òfwel het resultaat van f specificeren, òfwel het argument van g.

De enige manier om dat te doen, is het type van f (als functie) of het type van g (als functie) te geven.

Hieronder een voorbeeld:

rolBindingDef :: forall c b rt e. Binding b => RolKind rt => ContextType c => (rt ~~> c) e

rolBindingDef = (binding :: (rt ~~> b) e) /-/ context

Als we de typering van de toepassing van binding weglaten, klaagt de compiler. Het type b komt niet voor in de body van het type van rolBindingDef. We moeten het declareren, zo nodig beperken met een type class, en gebruiken in de body van de functie.

1. Een type is in Perspectives beschreven met een Context (óók Rol- en Property types!). [↑](#footnote-ref-1)
2. Dit zijn dus Context-typen of Context-beschrijvingen of Context-definities. Verwar ContextDef niet met ContextType. ContextDef is de definitie van Context; ContextType is de Class waarvan alle context-typen lid zijn. [↑](#footnote-ref-2)
3. Omdat de eerste functie een type variabele oplevert en de tweede een type variabele neemt, zullen we deze functie met die type variabele moeten kwalificeren (zie hieronder). [↑](#footnote-ref-3)