Typen from first principles

|  |  |  |
| --- | --- | --- |
| Joop Ringelberg | 28-11-17 – 11-04-18 | Versie: 3 |

# Introductie

Deze tekst heb ik geschreven om de manier van redeneren vast te leggen waarmee uit een klein aantal beginselen het gebouw van typen en metatypen opgetrokken kan worden. Tegelijkertijd beschrijft deze tekst een taal, de *Context Role Language* (CRL), waarin typen en instanties beschreven kunnen worden.

# Uitgangspunt

We willen een Perspect model in tekst kunnen opschrijven. We hebben een minimale syntax gedefinieerd waarmee we alle aspecten van een context en een rol daarin, met properties voor beiden, kunnen beschrijven. Vertrekkend uit een expressie waarin al deze aspecten zijn verwerkt, kunnen we beredeneren welke types nodig zijn en hoe we die representeren.

Het uitgangspunt is:

:Aangifte :a1

extern :urgentie = 7

intern :aantekening = “Dit is een hopeloos geval.”

:aangever => :Jansen

:betrouwbaarheid = 6

Deze tekst telt vijf typen expressies:

1. type declaratie: :Aangifte :a1;
2. extern context property assignment: extern :urgentie = 7;
3. intern context property assignment: intern :aantekening = “…”;
4. rolbinding: :aangever => :Jansen;
5. rol property assignment: :betrouwbaarheid = 6.

Er zijn nog enkele typen expressies die we nodig hebben. Deze komen aan bod als we het voorbeeld uitbreiden.

# Afleidingsregels

Uit de type declaratie expressie volgt natuurlijk wat een type is en wat een instantie.

Een type declaratie impliceert een type (I)

Een binding-expressie is altijd ingebed in een expressie die een Context beschrijft. Hetzelfde geldt voor een extern property assignment en een intern property assignment.

intern, extern of => impliceert een context (II)

Een rolProperty (of een query, zoals we verderop in de tekst zullen zien) impliceert een rol:

een niet-gedecoreerde property assignment of het query sleutelwoord impliceert een rol (III)

De combinatie van een type declaratie en een binding-expressie impliceert dat het type een :rolInContext rol heeft die gevuld is met de rol die gebonden wordt:

:Type <perspectName>

:role => <perspectName>

impliceert dat :Type een rol :rolInContext heeft, gevuld met :role (IV)

**Corollary**: :rolInContext => :Role legt op dat in instanties (van :Type) :role => <perspectName> geschreven moet worden.

Dezelfde redenering kunnen we toepassen op de drie property assignment expressies. Bij elkaar gezet:

|  |  |
| --- | --- |
| Bevat de instantie-expressie: | Dan geldt voor type: |
| :Type <perspectName> | :Context :Type |
| :role => :Filler | :rolInContext => :role |
| :prop = value | :rolProperty => :prop |
| intern :prop = val | :internalProperty => :prop |
| extern :prop = val | :externalProperty => :prop |
| query <operator>+ | :Query |

Een type dat rollen, intern of extern properties definieert voor zijn instanties, definieert een type context. Het type van zo’n type is per constructie Context. (V)

Een type dat (rol)properties of een query definieert voor zijn instanties, definieert een type rol. Het type van zo’n type is per constructie RolInContext. (VI)

# Afleiding van :Context

Het uitgangspunt was:

:Aangifte :a1

extern :urgentie = 7

intern :aantekening = “Dit is een hopeloos geval.”

:aangever => :Jansen

:betrouwbaarheid = 6

Hieruit kunnen we nu met de afleidingsregels het type :Aangifte afleiden.

:Context :Aangifte

:externalProperty => :urgentie

:internalProperty => :aantekening

:rolInContext => :aangever

Daaruit volgt op dezelfde manier de definitie van :Context:

:Context :Context

:rolInContext => :externalProperty

:rolInContext => :internalProperty

:rolInContext => :rolInContext

Merk op dat deze definitie consistent is in die zin dat de regel :rolInContext => :rolInContext volgens regel IV impliceert dat het type van :Context een rol in context heeft die gevuld moet worden met :rolInContext. Het type van :Context is :Context en die heeft precies zo’n regel, namelijk diezelfde regel!

# Uitbreiding van het uitgangspunt

We willen ook nog een en ander zeggen over de diverse properties, namelijk:

* is een property functioneel?
* is een property verplicht?
* wat is de range van een property?

Nemen we de publieke contextproperty urgentie als voorbeeld:

:Property :urgentie

extern :isFunctioneel = true

extern :isVerplicht = false

:range => :Number

Waarom :Property? Wel, :urgentie moet een type hebben en dit is een redelijke naam. Waarom extern properties? De alternatieven stuiten op problemen: intern is onzinnig op dit niveau (intern voor wie?) en als we een rolproperty zouden definiëren, volgt daaruit onmiddellijk dat :Property zelf een rol moet zijn – maar waar is het dan een rol van? Kortom, :Property moet een context zijn – en dan moeten zijn properties publiek zijn.

Moeten we :urgentie niet als :externalProperty definiëren? Nee. :urgentie speelt de rol :externalProperty bij :Aangifte en daarmee is al vastgelegd om wat voor property het gaat.

Waarom een rol voor :range? We willen voor Perspectives een typechecker bouwen. Daartoe willen we het type van een expressie representeren. We hebben ervoor gekozen de types van Rollen en Contexten met Contexten te representeren. Een expressie die evalueert tot een property-waarde, heeft natuurlijk ook een type – namelijk de range. Wel zo handig als we dat type dan óók met een Context representeren. Maar dan moet :range een rol zijn, want een property kunnen we niet aan een Context (buitenrol) binden.

# Afleiding van :Property, :IsFunctioneel, :IsVerplicht en :Range

Uit de uitbreiding van het uitgangspunt:

:Property :Urgentie

extern :isFunctioneel = true

extern :isVerplicht = false

:range => :Number

volgt één op één de definitie van :Property:

:Context :Property

:externalProperty => :isFunctioneel

:externalProperty => :isVerplicht

:rolInContext => :range

Merk op dat deze definitie volledig consistent is met de definitie van :Context (want die heeft een rol in context gevuld met :externalProperty èn een rol in context gevuld met :rolInContext).

:isFunctioneel, :isVerplicht en :range volgen nu. We willen namelijk ook in staat zijn om aan te geven dat het in Perspect verplicht is om deze twee properties toe te passen; dat ze alle drie functioneel zijn; en wat hun range, respectievelijk mogelijkeBinding is.

:Property :IsFunctioneel

extern :isFunctioneel = true

extern :isVerplicht = true

:range => :Boolean

:Property :IsVerplicht

extern :isFunctioneel = true

extern :isVerplicht = true

:range => :Boolean

:Range heeft een mogelijke binding:

:Rol :Range

extern :isFunctioneel = true

:mogelijkeBinding => :Context

Merk op dat we niet kunnen aangeven dat de rol :Range verplicht is.

# :Rol

Het voorbeeld laat structuur zien van de rol :aangever:

:Aangifte :a1

…

:aangever => :Jansen

:betrouwbaarheid = 6

Bovendien willen we kunnen beperken waar :aangever aan gebonden kan worden. Laten we zeggen dat we die bindingen willen beperken tot instanties van het type :Persoon, of rollen die daarmee gevuld zijn.

:Rol :aangever

:rolProperty => :betrouwbaarheid

:mogelijkeBinding => :Persoon

Bij de definitie van :Context ontstond :rolInContext. We gebruiken hier echter de algemenere naam :Rol, omdat we óók binnen- en buitenrollen hebben. :rolProperty volgt rechtstreeks uit de afleidingsregels.

Nu kunnen we :Rol afleiden:

:Context :Rol

:rolInContext => :rolProperty

:rolInContext => :mogelijkeBinding

Later zullen we :rolInContext verder uitbreiden.

Kunnen we iets zeggen over :mogelijkeBinding? Eigenlijk niets:

:Context :mogelijkeBinding

is voldoende, zoals bij :internalProperty en :externalProperty.

# :internalProperty, :externalProperty en :rolProperty

Ik heb de behandeling van deze drie uitgesteld. Deze types zagen we voor het eerst bij :Context. Maar er valt eigenlijk niets over hen te zeggen. Bij verstek representeer ik ze als contexten, maar eigenlijk kan je alles kiezen:

:Context :internalProperty

:Context :externalProperty

:Context :rolProperty

# Uitbreiding van het uitgangspunt

Bij een rol hoort een view[[1]](#footnote-1). Nu we een definitie hebben van :aangever, kunnen we er een view aan toevoegen:

:Rol :aangever

:rolProperty => :betrouwbaarheid

:mogelijkeBinding => :Persoon

:view => :aangeverBetrouwbaarheidsPerspectief

We moeten nu de definitie van :Rol uitbreiden:

:Context :Rol

:rolInContext => :rolProperty

:rolInContext => :mogelijkeBinding

:rolInContext => :view

# Afleiding van :view

Hoe beschrijven we het :aangeverBetrouwbaarheidsPerspectief? Als volgt:

:view :aangeverBetrouwbaarheidsPerspectief

:propertyReferentie => :betrouwbaarheid

:volgNummer = 1

En hieruit volgt de definitie van :view zelf:

:Context :view

:rolInContext => :propertyReferentie

en dan de rol :propertyReferentie en :volgnummer:

:Rol :propertyReferentie

:rolProperty => :volgnummer

:mogelijkeBinding => :Property

:Property :volgnummer

extern :isFunctioneel = true

extern :isVerplicht = true

extern :range = Number

# Cardinaliteit van rollen

Een aangifte kan door meerdere aangevers gedaan worden. Oftewel: net zoals bij een property willen we van een rol kunnen aangeven wat zijn cardinaliteit is. Breiden we het voorbeeld van :aangever uit:

:Rol :aangever

extern :isFunctioneel = false

:rolProperty => :betrouwbaarheid

:mogelijkeBinding => :Persoon

Dienovereenkomstig breiden we :Rol nogmaals uit:

:Context :Rol

:rolInContext => :rolProperty

:rolInContext => :mogelijkeBinding

:rolInContext => :view

:externalProperty => :IsFunctioneel

:Context ondersteunt reeds :externalProperty, dus die hoeven we niet uit te breiden.

# Query

Een rol in een context kan ook berekend worden. De berekening noemen we een query. Voor queries ontwerpen we een speciale syntax voor Arc (nog niet bekend op het moment van schrijven). In CRL wordt een query gerepresenteerd met typen die erven van :Query, zoals :compose. Bijvoorbeeld:

:rolInContext =>

:compose :mijnBerekendeRol

:operand => :aangever

:operand => binding

Rol- en propertynamen (beginnend met een kleine letter) zijn operands voor :compose. binding is ook een operand. Daarnaast is er een klein repertoire aan combinatoren om queries te combineren, zoals filter. Queries worden elders uitgebreider beschreven.

We hoeven :Rol niet nogmaals uit te breiden, want we kunnen een query dus gewoon opschrijven als de inline gedefinieerde binding van een rolInContext.

# Inline type declaratie bij rolbinding

Een type declaratie kan opgenomen worden na de binding-pijl. Op die manier kunnen we rollen en properties, die immers precies in respectievelijk één context en één rol voorkomen, inline opnemen.

:Context :view

:rolInContext =>

:Rol :propertyReferentie

:rolProperty => :volgnummer

:mogelijkeBinding => :Property

# Syntactische ontleding en semantische constraints

Uit de definities (zie Appendix 2) volgen restricties. De rol :rolProperty mag bijvoorbeeld alleen gebonden worden in de beschrijving van een rol. Dit blijkt uit regel IV, die van toepassing is op deze situatie:

:Type <perspectName>

:role => <perspectName>

Regel IV stelt dat :Type een rol :rolInContext heeft, gevuld met :role. Beschouw het voorbeeld: :rolProperty => :betrouwbaarheid. :role is dan :rolProperty en het enige type waarin de expressie :rolInContext => :rolProperty voorkomt is :Rol. En alleen dat type beschrijft een rol.

We noemen dit een *semantische constraint*. Semantische constraints volgen dus uit de toepassing van de afleidingsregels op types. Welke types? Alle types! Er is wat dit betreft geen verschil tussen de types in Appendix 2 en types die de modelleur aanmaakt.

Semantische constraints moeten we onderscheiden van syntactische constraints. Deze laatste volgen uit de grammatica voor correcte Perspect expressies. Dit zijn de enige correcte expressies:

1. type declaratie: :Aangifte :a1;
2. externe context property assignment: extern :urgentie = 7;
3. interne context property assignment: intern :aantekening = “…”;
4. rolbinding: :aangever => :Jansen;
5. rol property assignment: :betrouwbaarheid = 6.
6. inline type declaration: :role => :Type <perspectName>....

Er is (syntactische) samenhang tussen deze expressies, die alleen blijkt uit de grammatica zelf (zie hieronder).

De ontleder hanteert alleen de grammatica en geeft aan of de tekst voldoet aan de syntactische constraints. Bovendien levert de ontleder een Abstract Syntax Tree (AST). Deze AST is louter opgesteld in termen van contexten en rollen.

De grammatica luidt:

expression = context | role

context = typeDeclaration

externContextPropertyAssignment\*

internContextPropertyAssignment\*

rolBinding\*

role = typeDeclaration

rolePropertyAssignment\*

typeDeclaration = perspectName perspectName

externContextPropertyAssignment = 'extern' propertyName '=' simpleValue

internContextPropertyAssignment = 'intern' propertyName '=' simpleValue

rolBinding = roleName '=>' (perspectName | expression)

rolePropertyAssignment\*

rolePropertyAssignment = propertyName '=' simpleValue

roleName = lowerCaseChar anyChar\*

propertyName = lowerCaseChar anyChar\*

perspectName = upperCaseChar anyChar\*

Uit regel II volgt de definitie van context: iets is een context als het een externe of interne context property heeft, of als het een rolbinding heeft. Op dezelfde manier volgt de definitie van rol uit regel IV: iets is een rol, als het een rolproperty of een query heeft.

Contexten en rollen samen noemen we expressie[[2]](#footnote-2). Een rolBinding is op een perspectName, of op een expression (inline type declaration). De andere grammatica regels volgen direct uit het voorbeeld dat we als uitgangspunt namen.

De ontleder levert dus een serie contexten en rollen als de ontlede tekst grammaticaal correct is. In zo’n tekst zou een rolproperty als onderdeel van een context definitie gebonden kunnen worden (en daarmee de constraint geschonden worden die we in het begin van dit hoofdstuk beschreven), zonder dat de ontleder er bezwaar tegen maakt.

De semantische controle toont zo’n geschonden constraint aan (b.v. door in de editor de betreffende expressie te markeren). Of het systeem probeert te voorkomen dat de modelleur zo’n constraint schendt.

# De type-ketting is niet transitief

Een Context heeft een type. Maar een type representeren we met een Context en die heeft natuurlijk (weer) een type. Zo ontstaat een ketting van types. Context heeft zichzelf als type, dus daar eindigt de ketting.

‘Type’ is in Perspectives géén transitieve relatie. Met andere woorden, als A een B is en B een C, is A geen B! Vergelijk dit met de relatie ‘kind’: Arie is een kind van Bea en Bea is een kind van Chris. Maar Arie is dan géén kind (maar een kleinkind) van Chris.

In Perspectives is de relatie ‘type’ niet transitief omdat een type een beschrijving is van de structuur van een verzameling instanties. Er is geen enkele reden om te veronderstellen dat een type ook een beschrijving zou zijn van de structuur van de instanties van die instanties (zouden die bestaan).

Desalniettemin kunnen we de type-ketting wel gebruiken. In Perspectives controleren we bijvoorbeeld of een rol berekend is, door te onderzoeken of in zijn type ketting ‘psp:Query’ voorkomt. Maar dat betekent niet, dat het type van de query-expressie zelf een query is.

# Prototype inheritance en partial instantiation

Er kunnen zich situaties voordoen waarbij telkens opnieuw een instantie van een context wordt aangemaakt waarbij enkele rollen op dezelfde manier worden gevuld. Denk bijvoorbeeld aan een Zaak Vervoer met twee rollen, Vervoerder en Passagier. Professionele Vervoerders zullen heel veel van dat soort zaken hebben waarbij ze telkens die ene rol vervullen!

In zo’n geval kan een *prototype* uitkomst bieden. Een prototype zou in dit geval niets anders zijn dan een Vervoer-Zaak waarvan alleen de Vervoerder-rol ingevuld is. Voor het vervoer van een passagier maak je dan een andere, nieuwe instantie van Vervoer aan, waarbij je aangeeft dat het prototype gebruikt wordt. Je hoeft dan alleen nog maar de Passagier-rol in te vullen.

In het algemeen: een prototype is een gedeeltelijk ingevulde Context. We representeren het prototype bij een context door de buitenrol van die context met dat prototype te vullen. Zo kunnen we gebruik maken van het bestaande roltelescoop mechanisme om (externe) properties van het prototype te gebruiken.

Functionele rollen en properties van het type die in het prototype een binding of waarde hebben in het prototype, zijn *gesloten* voor instanties die dat prototype refereren. De waarde van een gesloten rol of property wordt niet bij de instantie gevonden, maar bij zijn prototype.

Een context kan dus een type en een prototype hebben. Dan geldt dat het type van het prototype gelijk is aan het type van de context!

In CLR leggen we het prototype van een Context vast met de rol *prototype*.

psp:Context psp:Context

psp:rolInContext => psp:externalProperty

psp:rolInContext => psp:internalProperty

psp:rolInContext => psp:rolInContext

psp:rolInContext => psp:interneView

psp:rolInContext => psp:externeView

**psp:rolInContext => psp:prototype**

psp:Context psp:prototype

# Acties en ActorRol

Een Actie is een Context met standaard rollen. Deze dragen de namen van grammaticale begrippen: onderwerp, gezegde, lijdend voorwerp, meewerkend voorwerp. Het onderwerp van een Actie moet een ActorRol zijn. Een ActorRol wordt uiteindelijk gevuld door een Gebruiker van het Systeem.

Een Actie kan voor meerdere ActorRollen gelden. Daarom representeren we bij een ActorRol welke Acties hij kan gebruiken, met een speciale rol:

psp:Context psp:ActorRol

…

psp:rolInContext => psp:actie

De verzameling van Acties bij een ActorRol noemen we een *Perspectief*.

# Andere rollen en acties

Bij elk ander type rol kan de modelleur aangeven welke Acties erop aangrijpen (een Actie grijpt aan op een Rol als het lijdend voorwerp van die Rol eraan gebonden is). Dit kan op dezelfde manier gerepresenteerd worden.

De verzameling van Acties bij een gewone rol noemen we de *Bewerkingsmogelijkheden*.

# Aspecten

Perspectives ondersteunt geen type-overerving, maar heeft in plaats daarvan een compositioneel veel sterker middel: het *Aspect*. Elke typebeschrijving van een Context (in deze paragraaf kortweg: Context) kan een Aspect zijn.

Een type is een aspect voor een ànder type. De modelleur verbindt een actor-rol in een type met een ActorRol in een Aspect en voegt er zo de properties en Acties van de Aspect ActorRol aan toe. Verbindt hij een gewone rol-in-context met die van een Aspect, dan voegt hij alleen de properties van de Aspect-rol toe aan de rol van het context-type.

Een context waar we een Aspect aan toevoegen, is in feite (mede) samengesteld uit dat Aspect. Bovendien kun je ad libidum Aspecten aan een Context toevoegen. We spreken dan ook in dit verband van een *samengestelde* Context.

Een Actie van een Aspect stelt eisen aan de samengestelde Context. De rol die het lijdend voorwerp vult, moet de view die voor dat lijdend voorwerp geldt, ondersteunen. D.w.z. dat die rol de properties moet dragen die in de view genoemd worden. Bovendien kunnen Constraints gelden voor het lijdend voorwerp. Een Constraint is opgesteld in termen (Rollen, Properties) waarin de samengestelde Context óók moet voorzien.

Dit geldt ook voor de ActorRol van de samengestelde Context.

Merk op dat het Aspect ook properties kan tóevoegen aan Rollen. In dat geval wordt automatisch aan de betreffende eisen voldaan.

Merk op dat, omdat een Aspect gewoon een Context is (althans, een beschrijving van het type van een Context), een Aspect zelf ook een Samengestelde Context mag zijn!

Een Rol van een als Aspect gebruikte Context kan natuurlijk ook mogelijkeBindingen hebben. Als een AspectRol wordt toegevoegd aan een samengestelde Context, moeten we verifiëren dat in elk pad van de RolDAG minstens één van die als mogelijkeBindingen genoemde rollen voorkomt.

Een Aspect representeren we met twee extra rollen, één voor Context en één voor Rol.

psp:Context psp:Context

…

psp:rolInContext => psp:aspect

psp:Context psp:Rol

…

psp:rolInContext => psp:aspectRol

# Constraints

Een Constraint is een query waarvan het type psp:Boolean is. Een Constraint is dus heel vergelijkbaar met een Property. Het runtime systeem signaleert een fout als de Constraint van een Rol de waarde false heeft.

Bijvoorbeeld: de rol chauffeur van een Zaak Vervoer zou als constraint kunnen hebben dat de leeftijd van de vuller van die rol hoger moet zijn dan 18. Binden we de rol aan een jongere chauffeur, dan krijgen we een melding (of het systeem verhindert de binding).

In type-time zal de type-checker controleren of de Constraint in runtime wel tot een resultaat kan leiden.

Constraints kunnen willekeurige expressies zijn, zolang ze maar een boolean waarde hebben. Net zoals bij queries is de variabele #uitgangspunt standaard gebonden (aan de rol waar de Constraint voor geldt).

Een Constraint op een Context is natuurlijk een Constraint op de binnen- of buitenrol van die Context.

# Appendix 1. Integraal voorbeeld

:Context :Aangifte

:externalProperty =>

:Property :urgentie

extern :isFunctioneel = true

extern :isVerplicht = false

:range => :Number

:internalProperty =>

:Property : aantekening

extern :isFunctioneel = true

extern :isVerplicht = false

:range => :String

:rolInContext =>

:Rol :aangever

extern :isFunctioneel = false

:rolProperty =>

:Property : betrouwbaarheid

extern :isFunctioneel = true

extern :isVerplicht = false

:range => :Number

:mogelijkeBinding => :Persoon

:view =>

:view :aangeverBetrouw…Perspectief

:propertyReferentie => :betrouwbaarheid

:volgNummer = 1

:rolInContext =>

:compose :mijnBerekendeRol

:operand => :aangever

:operand => binding

# Appendix 2. Alle types

-- CRL DEFINITIE

Context model:Perspectives als psp:

Section psp:rolInContext

---------------------------------------------------------------------------

-- SYSTEM BASE

---------------------------------------------------------------------------

psp:Context psp:internalProperty

psp:Context psp:externalProperty

psp:Context psp:rolProperty

psp:Context psp:query

psp:Context psp:rolInContext

psp:Context psp:viewInRol

psp:Context psp:interneView

psp:Context psp:externeView

psp:Context psp:prototype

psp:Context psp:Boolean

psp:Context psp:String

psp:Context psp:Number

psp:Context psp:Date

psp:Property psp:isFunctioneel

extern psp:isFunctioneel = true

extern psp:isVerplicht = true

psp:range => psp:Boolean

psp:Property psp:isVerplicht

extern psp:isFunctioneel = true

extern psp:isVerplicht = true

psp:range => psp:Boolean

psp:Rol psp:range

extern psp:isFunctioneel = true

psp:mogelijkeBinding => psp:Context

psp:Rol psp:mogelijkeBinding

extern psp:isFunctioneel = false

psp:Context psp:Property

psp:externalProperty => psp:isFunctioneel

psp:externalProperty => psp:isVerplicht

psp:rolInContext => psp:range

psp:Rol psp:propertyReferentie

psp:rolProperty => psp:volgNummer

psp:mogelijkeBinding => psp:Property

psp:Property psp:volgNummer

extern psp:isFunctioneel = true

extern psp:isVerplicht = true

psp:range => psp:Number

psp:Context psp:View

psp:rolInContext => psp:propertyReferentie

psp:Context psp:Context

psp:rolInContext => psp:externalProperty

psp:rolInContext => psp:internalProperty

psp:rolInContext => psp:rolInContext

psp:rolInContext => psp:interneView

psp:rolInContext => psp:externeView

psp:rolInContext => psp:prototype

psp:rolInContext => psp:aspect

psp:Context psp:Rol

psp:rolInContext => psp:rolProperty

psp:rolInContext => psp:mogelijkeBinding

psp:rolInContext => psp:viewInRol

psp:externalProperty => psp:isVerplicht

psp:externalProperty => psp:isFunctioneel

psp:rolInContext => psp:query

psp:rolInContext => psp:aspectRol

psp:rolInContext => psp:actie

1. Views horen strikt genomen niet tot CRL, maar tot ARC. Een View speelt namelijk alleen een rol in een Actie en Acties horen per definitie bij ARC, niet bij CRL. [↑](#footnote-ref-1)
2. NB. De purescript implementatie constateert een lus in de definities van rolBinding, expression en context. Daarom heb ik expression niet geïmplementeerd. [↑](#footnote-ref-2)