Bachelorscriptie
Informatica

Radboud Universiteit

Een Deductieve Semantiek voor iTasks

Auteur:
Tom Nikken
s4229649
tom.nikken@student.ru.nl

Inhoudelijk begeleider:
Dr. P.M. Achten
P.Achten@cs.ru.nl

Tweede begeleider:
Prof. dr. J.H. Geuvers
H.Geuvers@cs.ru.nl

Tweede lezer:
Dr. P.W.M. Koopman
pieter@cs.ru.nl

18 augustus 2016

Samenvatting

iTasks is een web-gebaseerd framework voor de functionele taal Clean. iTasks is gebaseerd op het programmeerparadigma Task-Oriented Programming. Er bestaat een formele executeerbare semantiek voor iTasks, maar het is vrijwel ondoenlijk om met deze semantiek formele bewijzen over programma's in iTasks te maken. Ik presenteer een deductieve semantiek die geschikt is om formeel mee te redeneren. Ik maak het aannemelijk dat de deductieve semantiek correct is en geef een 'proof of concept' van de semantiek door de uitkomst van een illustratief programma te bewijzen.

Dankwoord

Ik wil graag mijn begeleiders Peter Achten en Herman Geuvers bedanken voor hun hulp. Hun adviezen en feedback waren zeer waardevol voor deze scriptie. Ook wil ik mijn tweede lezer Pieter Koopman bedanken voor zijn feedback.

Inhoudsopgave

Hoofdstuk 1

Inleiding

Het ontwikkelen van een softwareproduct is een kostbaar proces. Gelukkig zijn er frameworks beschikbaar om het ontwikkelproces te versnellen. iTasks[4][1] is een framework voor de functionele taal Clean waarmee gedistribueerde multi-user web-applicaties ontwikkeld kunnen worden. iTasks heeft, zoals veel softwaresystemen, een grote omvang en complexiteit. Dit maakt het lastig om te redeneren over programma's in iTasks. We hebben een semantische beschrijving nodig waarmee we eigenschappen van programma's in iTasks formeel kunnen bewijzen.

Plasmeijer et al. [5] hebben een semantiek ontwikkeld in de vorm van een vereenvoudigde implementatie van iTasks in Clean[2]. In het vervolg van deze scriptie zal deze semantiek de *implementatiesemantiek* worden genoemd. Een implementatie als semantiekvorm maakt de semantiek executeerbaar. Het voordeel hiervan is dat de Clean compiler kan bijdragen aan de correctheid van de semantiek door te type-checken. Ook maakt deze implementatie het eenvoudiger om de correctheid van de semantiek te testen.

De implementatiesemantiek is echter nog steeds te groot om in de praktijk gebruikt te worden om bewijzen te maken. Het doel van dit onderzoek is het vinden van een semantiek waarmee het makkelijker is om eigenschappen over programma's te bewijzen. De vraag die in deze scriptie beantwoord wordt, is:

> *Met welke semantiek kunnen we makkelijker redeneren over iTasks en over programma's in iTasks?*

De semantiek die uit dit onderzoek is gekomen, is een deductieve operationele semantiek. Ik heb gekozen voor een operationele semantiek, omdat dit erg dicht tegen de implementatiesemantiek aan ligt. Dit maakt het makkelijker om de correctheid van de semantiek te controleren. Het onderliggende formele systeem is een afleidingssysteem. Het voordeel hiervan is dat we

[1] `http://clean.cs.ru.nl`
[2] `http://clean.cs.ru.nl/ITasks`

hiermee per situatie kunnen aangeven wat het gedrag van een component is. In de implementatiesemantiek moet dit voor alle situaties in één keer worden gegeven.

Voor deze scriptie bleek het ondoenlijk om een semantische beschrijving van het hele systeem te maken. Ik heb daarom een versimpelde versie van iTasks ontworpen, genaamd iTasks'. Van iTasks' heb ik een semantiekbeschrijving gemaakt.

De opbouw van deze scriptie is als volgt. In hoofdstuk 2 worden de details van iTasks uitgelegd. Hoofdstuk 3 behandelt de implementatiesemantiek. In hoofdstuk 4 beschrijf ik de versimpelde versie van iTasks die ik heb gemaakt. De semantiek die uit mijn onderzoek gekomen is, leg ik uit in hoofdstuk 5. In hoofdstuk 6 beargumenteer ik de correctheid van de deductieve semantiek ten opzichte van de implementatiesemantiek. Om te laten zien hoe de semantiek in zijn werk gaat, zal ik in hoofdstuk 7 een paar eigenschappen van iTasks' bewijzen. In hoofdstuk 8 bespreek ik het gerelateerde werk. In hoofdstuk 9 bekijk ik welke conclusies kunnen worden getrokken op basis van dit onderzoek. We sluiten af met een vooruitzicht op toekomstig werk in hoofdstuk 10.

Hoofdstuk 2

iTasks

Dit hoofdstuk bevat een intuïtieve uitleg van de belangrijkste elementen van iTasks. In dit hoofdstuk is gebruik gemaakt van de developmentversie van Clean en iTasks voor Windows (29 januari 2016).

Taken zijn de fundamentele concepten van iTasks. Een taak is een opdracht die door een computer kan worden uitgevoerd. Een taak kan tijdens executie samenwerken met andere taken. Op elk tijdstip heeft een taak een taakwaarde. De taak gebruikt de taakwaarde om zijn voortgang te communiceren naar andere taken. De waarde kan door de context van de taak worden bekeken. De context kan reageren op de taakwaarde, bijvoorbeeld door het starten van een nieuwe taak. De opdracht die een taak uitvoert kan heel concreet zijn, zoals een berekening of I/O, maar kan ook abstract zijn, zoals interactie met de gebruiker, een webpagina weergeven et cetera.

Een taak wordt in iTasks gerepresenteerd door een expressie van het type `Task a`. Een expressie van het type `Task a` noemen we een taakexpressie of ook wel een taak. De waarde van een taak wordt gerepresenteerd door een expressie van het type `TaskValue a`. De waarde kan stabiel of instabiel zijn, of er kan zelfs helemaal geen waarde zijn:

```
:: TaskValue a = NoValue | Value !a !Stability
:: Stability :== Bool
```

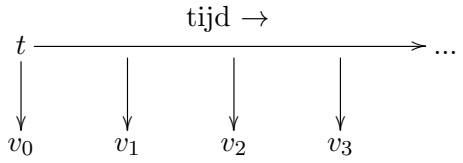
Een taak kan worden uitgevoerd door zijn representatie door iTasks te laten evalueren. Het evalueren bestaat uit meerdere evaluatiestappen. Tijdens een evaluatiestap kan de waarde van een taak veranderen. Tussen twee evaluatiestappen blijft de waarde van een taak gelijk. Als een taak een stabiele waarde heeft, betekent dit dat deze taak daarna continu dezelfde stabiele waarde zal hebben.

Een taak kan in plaats van een waarde ook een exceptie opleveren. Deze exceptie wordt door de meeste taken gepropageerd. Sommige taken kunnen de exceptie vangen.

Een taak reageert op events. Er wordt een iteratie gemaakt op het moment dat er een event het programma binnen komt. Op dat moment

produceert de taak een waarde. Het event kan onder andere gebruikersinvoer of een event van het operating system zijn.

In figuur 2.1 is het gedrag van een taak door de tijd heen weergeven. De taak heeft aan het begin van zijn executie de waarde v_0. Na de eerste iteratie heeft de taak de waarde v_1, na de tweede iteratie de waarde v_2 etc.



Figuur 2.1: Het gedrag van taak t door de tijd heen

iTasks is compositioneel opgebouwd. Er zijn functies om eenvoudige taken te maken. Daarnaast zijn er ook combinatoren waarmee een taak samengesteld kan worden.

In figuur 2.2 op bladzijde 8 staat een voorbeeld van een programma in iTasks. Met dit programma kunnen twee gasten van een restaurant een bestelling plaatsen. De twee gasten zullen we in het vervolg Romeo en Julia noemen, naar het bekende stel van William Shakespeare. Het programma zou gebruikt kunnen worden door de ober van het restaurant om de bestelling snel door te geven aan de keuken. We kunnen ons voorstellen dat dit programma draait op het bestelapparaat van de ober. De bestelling moet voor elke gast individueel bevestigd worden. Als alle bestellingen zijn bevestigd, worden ze op het scherm weergeven.

De `parallel` in `vraagBestellingen` (regel 10) zorgt ervoor dat Romeo en Julia hun keuze gelijktijdig kunnen opgeven. Het programma vraagt eerst de bestellingen op , waarna de bestelling wordt weergeven (regel 13). Het programma kijkt hiervoor of er evenveel gasten zijn die een keuze hebben gemaakt als het totaal aantal gasten (regel 18).

De beschrijving van een mogelijk scenario is als volgt:
Op een zaterdagavond gaan Romeo en Julia uit eten. Romeo weet vanaf het begin al dat hij gaat voor de biefstuk en geeft dit door aan de ober. De ober voert dit in en bevestigt de bestelling voor Romeo. Julia weet nog niet direct wat ze moet kiezen. Uiteindelijk besluit Julia dat ze de zalm neemt. Ze geeft dit door aan de ober, die het invoert in het programma en de bestelling voor Julia bevestigt. De bestelling van zowel Romeo als Julia zijn nu bevestigd. Het programma geeft de bestellingen door aan de kok, die kan beginnen met het klaarmaken van de gerechten.

Figuur 2.2: Bestelprogramma in iTasks

```
1  vraagBestelling :: Task String
2  vraagBestelling = enterInformation "Wat␣wilt␣u␣bestellen?" []
3                       >>* [OnAction ActionOk bevestigBestelling]
4  where
5      bevestigBestelling :: (TaskValue String) -> Maybe (Task String)
6      bevestigBestelling (Value order _) = Just (return order)
7      bevestigBestelling _ = Nothing
8
9  vraagBestellingen :: Task [(TaskTime,TaskValue String)]
10 vraagBestellingen = parallel (repeatn 2 (Embedded, const vraagBestelling)) []
11
12 programma :: Task ()
13 programma = vraagBestellingen >>* [OnValue alsIedereenBesteld]
14 where
15     alsIedereenBesteld :: (TaskValue [(TaskTime,TaskValue String)])
16                           -> Maybe (Task ())
17     alsIedereenBesteld (Value bestellingen _)
18     | length keuzes == length bestellingen
19                     = Just (toonBestellingen (zip2 [1..] keuzes))
20     = Nothing
21     where keuzes = [bestelling \\ (_, Value bestelling _) <- bestellingen]
22     alsIedereenBesteld NoValue = Nothing
23
24 toonBestellingen :: [(Int,String)] -> Task ()
25 toonBestellingen keuzes = viewInformation "Bestellingen" [] bestellingen
26                                                              @ (const ())
27 where
28     bestellingen = foldl (\txt (n,x) -> txt +++ toString n +++ ":␣"
29                           +++ x +++ "\n") "" keuzes
```

2.1 Basistaken

2.1.1 Return

Het meest simpele soort taak is een returntaak. Het type van return is als volgt:

```
return :: a -> Task a | iTask a
```

`return` a is een taak die continu de waarde `Value` a `True` oplevert.

Om sommige functies voor een bepaald type a te gebruiken, moeten de argumenten van de klasse `iTask` a zijn. Dit doe je door het statement **derive** `iTask` a in de code te plaatsen. Dit statement genereert boilerplate code die nodig is om de klasse `iTask` te implementeren. Door de automatische generatie van code, hoeft de programmeur zich niet bezig te houden met het schrijven van deze boilerplate code. Deze code wordt onder andere gebruikt

voor het genereren van gebruikersinterfaces (GUI's).

2.1.2 Editors

Een editor is een taak waarbij interactie met de gebruiker plaatsvindt. Hieronder staan een paar voorbeelden van functies waarmee editors in iTasks gemaakt kunnen worden:

```
enterInformation  :: !d ![EnterOption  m]     -> Task m | descr d & iTask m
updateInformation :: !d ![UpdateOption m m] m -> Task m | descr d & iTask m
viewInformation   :: !d ![ViewOption   m]  !m -> Task m | descr d & iTask m
```

Wanneer een editor van het type `Task` m wordt uitgevoerd, wordt er een GUI voor het type m gegenereerd. In het geval van `enterinformation` d *opt* is dit een lege GUI waar informatie van het type m ingevuld kan worden. `updateInformation` lijkt op `enterInformation`, maar bij `updateInformation` d *opt* m' wordt de informatie in de GUI geïnitialiseerd met m'. `viewInformation` d *opt* m' toont m' op het scherm. De waarde die deze taken opleveren, is de informatie die in de GUI aanwezig is. Deze waarde is nooit stabiel. Een editor heeft dus alleen een instabiele waarde of geen waarde.

Voorbeeld. Een voorbeeld van het gebruik van een editor in het bestelprogramma van figuur 2.2 is `enterInformation "Wat␣wilt␣u␣bestellen?" []`. Deze taak is van het type `Task String` en toont een GUI waar een bestelling als tekst ingevoerd kan worden.

2.1.3 Throw

De programmeur kan een exceptie genereren met een throwtaak:

```
throw :: e -> Task a | iTask e
```

`throw` e is een taak die de exceptie e gooit.

2.2 Combinatoren

Combinatoren zijn functies waarmee taken samengesteld kunnen worden. Als een taak t samengesteld wordt uit de taken $t_1, ..., t_n$, noemen we $t_1, ..., t_n$ de subtaken van t.

2.2.1 At

De waarde van een taak kan worden veranderd met de `@`. Het type van `@` is:

```
(@) infixl 1 :: !(Task a) !(a -> b) -> Task b | iTask a & iTask b
```

Elke keer dat de deeltaak een waarde produceert, wordt de waarde door de functie getransformeerd in een andere waarde. `return x @ f` is dus een taak die continu de waarde `Val (f x) True` produceert.

2.2.2 Bind

Je kunt taken sequentieel laten uitvoeren met de monadische bind combinator:

```
(>>=) infixl 1 :: (Task a) (a -> Task b) -> Task b | iTask a & iTask b
```

`task >>= aToTask` is een taak die eerst `task` uitvoert. Zolang `task` geen stabiele waarde oplevert, zal `task >>= aToTask` een `NoValue` opleveren. Indien `task` wel een stabiele waarde oplevert, zeg `Value a True`, zal `task >>= aToTask` de taak `aToTask a` gaan uitvoeren. We zeggen dan dat `task >>= aToTask` overstapt op de taak `aToTask a`. Op het moment dat de eerste taak een waarde heeft, kan de gebruiker ook zelf kiezen om door te gaan naar `aToTask a` door op een gegenereerde GUI knop te klikken.

Voorbeeld.

```
(return 42) >>= f
    where f n = viewInformation "Toon de waarde" [] n
```

`return 42` geeft direct een stabiele waarde. Hierdoor zal de taak `f 42` uitgevoerd worden. De waarde '42' wordt op het scherm getoond.

2.2.3 Step

De step is een algemenere versie van de bind. De programmeur kan bij de step zelf bepalen in welke gevallen de volgende taak begint.

```
(>>*) infixl 1 :: !(Task a) ![TaskCont a (Task b)] -> Task b | iTask a & iTask b
:: TaskCont a b
    =       OnValue             ((TaskValue a)  -> Maybe b)
    |       OnAction    Action  ((TaskValue a)  -> Maybe b)
    | E.e:  OnException         (e              -> b)           & iTask e
    |       OnAllExceptions     (String         -> b)
:: Action = ActionOK | ActionContinue | ActionCancel | ...
```

De step heeft twee argumenten: een `Task a` en een lijst van task continuations. De executie van *task* `>>*` *taksteps* begint met de executie van *task*. Iedere keer dat *task* een waarde v oplevert, zal er in de lijst met task continuations gezocht worden naar `OnValue` en `OnAction`. Als er een `OnValue` f wordt gevonden met $f\ v$ = `Just` $task_2$, dan gaat de taak verder met $task_2$. We zeggen dan dat *task* `>>*` *taksteps* overstapt op de taak $task_2$. Als *task* `>>*` *taksteps* niet overstapt, gaat de taak verder met *task* `>>*` *taksteps*.

`OnAction` lijkt sterk op `OnValue`. Het verschil is dat een `OnAction` *action* f alleen overstapt als $f\ v$ reduceert naar een `Just` $task_2$ èn de gebruiker een bepaalde invoer geeft. In de *action* is gespecificeerd welke gebruikersinvoer er precies nodig is. In het geval van `ActionOK` is dit bijvoorbeeld een GUI-knop die ingeklikt moet worden. Het type van `Action` in de implementatie is iets anders dan het type dat hier gegeven is. Om het niet te ingewikkeld te maken, heb ik hier voor een versimpeld type gekozen.

Als de deeltaak een exceptie e gooit, wordt er in de lijst met task continuations gezocht naar een `OnException`, waarvan het existentieel gekwantificeerde type overeenkomt met het type van de gegooide exceptie. Als die wordt gevonden, zeg `OnException` g, dan gaat de taak verder met $g\ e$. Indien er geen matchende `OnException` wordt gevonden, wordt er gekeken of er een `OnAllExceptions` f in de lijst met task continuations aanwezig is. Indien deze aanwezig is, zal de taak verder gaan met f `(toString` e `)`. Anders wordt de exceptie gepropageerd.

Voorbeeld. `vraagBestelling` uit het bestelprogramma zal eerst de editor `enterInformation "Wat␣wilt␣u␣bestellen?" []` uit gaan voeren. Tijdens elke iteratie wordt er gekeken naar de waarde van de editor. Als de editor een waarde heeft, zal er een OK-knop aanklikbaar worden. Wanneer de knop aangeklikt wordt, gaat `vraagBestellingen` verder met de taak `return` b, waarbij b de ingevulde bestelling is. Als de OK-knop niet aangeklikt wordt of als de editor geen waarde heeft, zal de taak verder gaan met `vraagBestelling`.

Voorbeeld. De taak `programma` uit het bestelprogramma begint met de executie van `vraagBestellingen`. Tijdens elke iteratie wordt `alsIedereenBesteld` toegepast op de waarde w van `vraagBestellingen`. `alsIedereenBesteld` kijkt of het aantal keuzes dat al gemaakt is even groot is als het aantal klanten. Als dat zo is, reduceert het naar `Just (toonBestellingen (zip2 [1..] keuzes)`. Hierdoor zal `programma` verder gaan met `toonBestellingen (zip2 [1..] keuzes` en de bestellingen weergeven. Als het aantal keuzes dat gemaakt is kleiner is dan het aantal klanten, reduceert `vraagBestellingen` w naar `Nothing`. Hierdoor zal de executie van `programma` verder gaan met `programma`.

Opmerking. De `>>*` kan worden gebruikt om de `>>=` te definiëren:

```
(>>=) t f = t >>* [ OnAction ActionContinue (hasValue f)
                  , OnValue (ifStable f)]
```

Dit is niet de precieze definitie van de `>>=`, maar heeft wel dezelfde werking.

2.2.4 Gedeelde data

Taken moeten soms onderling informatie uit kunnen wisselen. Binnen iTasks kan dit door het gebruik van Shared Data Sources (SDS). Een SDS is een globale variabele, waar op elk moment naar geschreven of van gelezen kan worden.

2.2.5 Or

Met de or kunnen twee taken tegelijkertijd uitgevoerd worden:

```
(-||-) infixr 3 :: !(Task a) !(Task a) -> Task a | iTask a
```

$task_1$ `-||-` $task_2$ is een taak die $task_1$ en $task_2$ tegelijkertijd uitvoert. $task_1$ noemen we de linkertaak en $task_2$ noemen we de rechtertaak. Als een van

de twee deeltaken een stabiele waarde produceert, zal $task_1$ `-||-` $task_2$ deze waarde in het vervolg altijd opleveren, ongeacht wat de waarde van de andere taak is.

2.2.6 Parallel

```
parallel :: ![(!ParallelTaskType,!ParallelTask a)]
    [TaskCont [(!TaskTime,!TaskValue a)] (!ParallelTaskType,!ParallelTask a)]
    -> Task [(!TaskTime,!TaskValue a)] | iTask a
:: ParallelTask a :== SharedTaskList a -> Task a
:: ParallelTaskType = Embedded | Detached ManagementMeta
```

`parallel [(`$type_1$`,`$task_1$`), ..., (`$type_n$`,`$task_n$`)] []` is een taak die de deeltaken $task_1$ tot en met $task_n$ parallel uitvoert. De deeltaken zijn van het type `ParallelTask a`. Deze taken kunnen de waardes die de andere deeltaken in de `parallel` produceren bekijken. Hiervoor is het gebruik van Shared Data Sources nodig. `ParallelTaskType` geeft aan of de taak door de huidige gebruiker (`Embedded`) of door een andere gebruiker (`Detached`) uitgevoerd moet worden. `ManagementMeta` bevat informatie die nodig is voor het uitvoeren van een taak door een andere gebruiker.

Als er een vriend van Romeo en Julia onverwacht aansluit bij het diner, is het handig als er dynamisch een taak bij `vraagBestellingen` kan komen om de bestelling van de nieuwe gast op te nemen. De lijst van `TaskCont` in het tweede argument biedt deze mogelijkheid. Een element uit de lijst specificeert welke condities er moeten gelden voordat de taak kan worden toegevoegd en welke taak moet worden toegevoegd aan de `parallel`.
Voorbeeld. `vraagBestellingen` uit het bestelprogramma is een voorbeeld van de toepassing van `parallel`. Deze taak voert de taak `vraagBestelling` twee maal gelijktijdig uit.

Hoofdstuk 3

Implementatiesemantiek

De implementatiesemantiek is een semantiek in de vorm van een implementatie. Zoals Plasmeijer et al. beschrijven heeft een implementatie als semantiekvorm meerdere voordelen:

- We kunnen de kracht van de onderliggende taal gebruiken om ingewikkelde taalconstructies makkelijker uit te leggen.
- De semantiek is executeerbaar. Hierdoor kunnen we testen of de semantiek correct is ten opzichte van de implementatie van iTasks.
- Het typesysteem kan helpen om de semantiek goed te definiëren. Dit maakt de kans dat er fouten in de semantiek zitten kleiner.
- De semantiek maakt het makkelijker om het framework in Clean of in een andere taal te implementeren.

De implementatiesemantiek is echter geen volledige implementatie en beperkt zich tot de kern van iTasks. De formele semantiek voor zaken zoals server-browser interactie, HTML en JavaScript, I/O, serialisatie, de creatie van GUI's en de creatie van events wordt niet beschreven. De focus ligt op de kern van de taal.

In de implementatiesemantiek worden de events één voor één aan het programma gegeven. Het programma houdt een gemeenschappelijke toestand bij. Een taak wordt gerepresenteerd als een functie die een event en een toestand als argumenten krijgt en die zich herschrijft naar o.a. een nieuwe toestand en een taakwaarde. Deze stap noem ik een iteratie. De semantiek legt uit hoe een event de control flow van een taak beïnvloedt, hoe het de toestand verandert en welke taakwaarde of exceptie de taak heeft. Als een taak een event verwerkt, geeft de taak het event en de toestand door aan zijn subtaken. In hoofdstuk 6 worden de details van de implementatiesemantiek behandeld.

De doelstelling van de implementatiesemantiek was om task-oriented programming te beschrijven aan de hand van het iTasks framework. Een

andere doel waar een formele semantiek voor gebruikt kan worden is het bewijzen van eigenschappen van programma's. Dit was echter niet het doel van de implementatiesemantiek.

Het is al veel eenvoudiger om met de implementatiesemantiek te redeneren over programma's in iTasks dan met de implementatie van iTasks. In de praktijk blijkt het nog steeds te lastig om eigenschappen over programma's te bewijzen. We zouden mogelijk formele bewijzen kunnen maken met gebruik van equationeel redeneren[1], maar de functietermen worden dan te groot om praktisch te gebruiken. Dit komt doordat de implementatiesemantiek nog te groot is. De definitie van de `>>*`(zie blz. 32) in de implementatiesemantiek bestaat bijvoorbeeld al uit 23 regels code en die van de `parallel`(blz. 34) bestaat uit 48 regels code.

Hoofdstuk 4

iTasks'

Het bleek te lastig om binnen het tijdbestek van een bachelorscriptie een semantiek te maken van alle componenten van de implementatiesemantiek. Daarom heb ik een selectie hiervan gemaakt. Deze selectie noem ik iTasks'. De volgende elementen zijn in iTasks' veranderd ten opzichte van iTasks:

- Alleen de taken die de kern van iTasks bepalen zijn opgenomen in de semantiek. Dit zijn de taken `return`, `edit`, `throw`, `@`, `>>=`, `>>*`, `-||-` en `parallel`.
- Shared data sources zijn verwijderd.
- Alle type-klasse constraints (`iTask` a, `descr` d) zijn verwijderd.
- Analoog aan de aanpak van de implementatiesemantiek is er geabstraheerd van de editors. In iTasks' bestaat de taak `edit` die een editor kan simuleren. Verschillende edits kunnen van elkaar worden onderscheiden op basis van het scenario dat ze simuleren.
- Excepties zijn niet meer universeel gekwantificeerd. In plaats daarvan is er een exceptiedomein `E`.
- De striktheids-annotaties in de types zijn weggelaten.
- De types van de `>>*` en de `parallel` zijn enigszins versimpeld.
- De naamgeving is op enkele plaatsen korter gemaakt.

iTasks'

Hieronder staan de types van de waardes, basistaken en combinatoren in iTasks':

```
:: Value a = Val a Stability | NoVal
:: Stability = Stable | Unstable

return :: a -> Task a
```

```
edit   :: [Maybe a] -> Task a
throw  :: E -> Task a

(@) infixl 1    :: (Task a) (a -> b) -> Task b
(>>=) infixl 1  :: (Task a) (a -> Task b) -> Task b
(>>*) infixl 1  :: (Task a) ((Value a) -> Maybe (Task b), E -> Maybe (Task b))
                           -> Task b
(-||-) infixr 3 :: (Task a) (Task a) -> Task a
parallel        :: [Task a] -> Task [Value a]
```

De types van de taakwaardes zijn conform de implementatiesemantiek.

De `edit` kan worden gebruikt om een editor te simuleren. Het idee van `edit` x is dat de gebruikersinvoer kan worden gesimuleerd met x. De gebruiker voert dus niet daadwerkelijk iets in, maar intern lijkt het alsof dat wel gebeurt. Als men wil simuleren dat er achtereenvolgens de waarden a_0, a_1, a_2 in de editor aanwezig zijn, dient de rij `[Just` a_1`, Just` a_2`, Just` a_3`]` aan `edit` gegeven te worden. De rij van achtereenvolgende waardes die worden gesimuleerd, wordt een scenario genoemd. De gebruiker kan zijn invoer ook weghalen. Dit kan worden gesimuleerd door een `Nothing` in de rij. `edit [Just 1, Nothing]` representeert bijvoorbeeld een editor waarin eerst de waarde 1 staat en vervolgens geen waarde.

iTasks' maakt gebruik van een type `E` voor excepties. Dit type kan door de programmeur worden gedefinieerd. Als in een programma in iTasks excepties van types E_1, E_2 etc. kunnen worden gegooid, kun je in iTasks' een equivalent programma schrijven door het exceptietype als volgt te definiëren:

```
:: E = E1 E1
     | E2 E2
     | ...
```

Op de plaatsen waar je een exceptie e van type E_1 gebruikt, moet je vervolgens E_1 e gebruiken. `throw` e wordt dan `throw` (E_1 e). Met deze aanpassing gaat dus geen theoretische expressiviteit verloren.

De `>>*` krijgt een paar van functies met de types `(Value a) -> Maybe (Task b)` en `E -> Maybe (Task b)` als argument in plaats van een lijst met continuations. t `>>*` (f, g) is een taak die begint met het uitvoeren van t. Elke keer wanneer t een waarde v produceert, wordt f v geëvalueerd. Wanneer dit reduceert naar een `Nothing`, zal er niets gebeuren. Als het reduceert naar `Just` t_2, zal de taak in zijn geheel verder gaan met het uitvoeren van t_2. Excepties kunnen worden afgevangen met g. Als t `>>*` (f, g) een exceptie e gooit, zal g e geëvalueerd worden. Als hier een `Just` t_3 uit komt, is t_3 de taak waarmee vervolgd wordt. Als g e evalueert naar `Nothing`, zal de exceptie gepropageerd worden.

De `>>*` in iTasks' heeft wel minder functionaliteit dan de `>>*` in iTasks, in tegenstelling tot de `throw`. Als er een `OnAction` in de lijst met continuations staat, is er geen equivalente taak in iTasks' te maken. De gebruiker heeft dus geen invloed op het moment van overstappen naar de volgende taak. Voor

het overige hebben de twee vormen van >>* wel dezelfde functionaliteit. De onderstaande twee programma's zijn bijvoorbeeld equivalent aan elkaar:

```
t >>* [ OnValue f1
      , OnValue f2
      , OnException g1
      , OnException g2
      , OnAllExceptions (const h) ]

g1 :: E1 -> Task a
g2 :: E2 -> Task a
```

iTasks

```
t >>* (f,g)
where
    f val
    | isJust (f1 val) = f1 val
    | isJust (f2 val) = f2 val
    = Nothing
    g (E1 e) = Just (g1 e)
    g (E2 e) = Just (g2 e)
    g _ = Just h
```

iTasks'

Indien er geen `OnAllExceptions` is, moet de regel `g _ = Just h` worden vervangen door `g _ = Nothing`, zodat de exceptie wordt gepropageerd.

De `parallel` is in iTasks' ook anders dan die in iTasks. De `parallel` in iTasks' krijgt een lijstje van taken mee die parallel uitgevoerd zullen worden. De waarde die de samengestelde taak oplevert, is een lijst van de waardes die alle subtaken hebben. De subtaken kunnen dus niet meer de waardes van de andere taken bekijken.

Het bestelprogramma in figuur 2.2 zou er in iTasks' uit kunnen zien als het programma in figuur 4.1. Het scenario zoals beschreven bij het bestelprogramma is in het programma in figuur 4.1 gecodeerd. `vraagBestellingen` is een parallelle compositie van de editors `romeo` en `julia` geworden. Romeo weet vanaf het begin al dat hij gaat voor de biefstuk. Daarom zijn de waardes in de rij van de `edit` voor Romeo gelijk aan `Just "Biefstuk"`. Julia weet nog niet direct wat ze moet kiezen. De eerste waarde in de rij van de `edit` is voor Julia dus `Nothing`. Daarna besluit Julia dat ze de zalm neemt. De tweede waarde in de rij is dus `Just "Zalm"`. Omdat dit programma in iTasks' is geschreven, is het voor de gasten niet mogelijk om hun keuze te bevestigen. Het programma is een step van `vraagBestellingen`, `f` en `g`. De `f` controleert of alle bestellingen gemaakt zijn en dus `Unstable` zijn. Als dit het geval is, worden de bestellingen gereturned. De `@` zorgt ervoor dat alleen een lijstje met de bestellingen wordt teruggegeven en niet de objecten van het type Value.

Figuur 4.1: Bestelprogramma in iTasks'

```
1  vraagBestellingen = parallel [romeo, julia]
2  where
3      romeo = edit [Just "Biefstuk", Just "Biefstuk"]
4      julia = edit [Nothing, Just "Zalm"]
5
6  programma = vraagBestellingen >>* (f,g)
7
8  f (Value bestellingen s)
9  | all isUnstable bestellingen = Just ( (return bestellingen) @ getVal)
10 = Nothing
11 where
12     isUnstable (Val v Unstable) = True
13     isUnstable _                = False
14     getVal values = map (\(Val v s) -> v) values
15
16 g exc = Nothing
```

Hoofdstuk 5

Deductieve semantiek van iTasks'

Dit hoofdstuk bevat de deductieve operationele semantiek van iTasks' die uit mijn onderzoek is gekomen. De semantiek bevat regels die aangeven welke waarde een taak op een zeker tijdstip heeft. Een overzicht van de regels is te vinden in bijlage A.

5.1 Uitspraken

We onderscheiden drie soorten uitspraken.

Definitie 1. $t \downarrow v$ betekent dat v de huidige waarde van taak t is. t is een term van type `Task` a en v een term van het type `Value` a. We zeggen dat taak t waarde v heeft.

Definitie 2. $t_1 \Rightarrow t_2$ betekent dat taak t_1 zich gedraagt als taak t_2 na één iteratie. t_2 staat voor het werk dat t_1 nog moet doen nadat t_1 één iteratie heeft gemaakt. t_1 en t_2 zijn termen van type `Task` a. We zeggen dat t_1 evolueert naar t_2. t_2 heet de *vervolgtaak* of de *evolutie* van t_1.

Definitie 3. $t \lightning e$ betekent dat taak t de exceptie e gooit tijdens de eerstvolgende iteratie. t is een term van type `Task` a en e is een term van type `E`.

Een voorbeeld van een uitspraak is:

`return` $a \downarrow$ `Val` a `Stable`

Deze uitspraak geeft aan dat `return` a de waarde `Val` a `Stable` heeft.

Figuur 2.1 geeft het gedrag van een taak weer als een zwarte doos. Een operationele semantiek breekt deze doos open en geeft een model van wat er binnenin de zwarte doos gebeurt. Figuur 5.1 lijkt op figuur 2.1, maar hier zijn de formele uitspraken van de denotationele semantiek gebruikt om het gedrag van de taak te beschrijven.

$$\begin{array}{ccccccc} t_0 & \Rightarrow & t_1 & \Rightarrow & t_2 & \Rightarrow & \ldots \\ \downarrow & & \downarrow & & \downarrow & & \\ v_1 & & v_2 & & v_3 & & \end{array}$$

Figuur 5.1: De beschrijving van het gedrag van taak t_0 door de denotationele semantiek

5.2 Regels

Hieronder volgen de deductieregels van de deductieve semantiek. Een overzicht van alle deductieregels is te vinden in de bijlage.

Notatie. De haakjes die gebruikt worden in de namen van de regels volgen de volgende conventie:

- Als de conclusie van de vorm $t \downarrow v$ is, zijn het ronde haakjes.
- Als de conclusie van de vorm $t \lightning e$ is, zijn het accolades.
- Als de conclusie van de vorm $t \Rightarrow t'$ is, zijn het vierkante haakjes.

5.2.1 Return

Het type van `return` is als volgt:

```
return :: a -> Task a
```

`return` a is een taak die altijd de waarde `Val` a `Stable` heeft:

$$\frac{}{\texttt{return}\ a \downarrow \texttt{Val}\ a\ \texttt{Stable}}\ \text{(return)}$$

De return-taak verandert nooit:

$$\frac{}{\texttt{return}\ a \Rightarrow \texttt{return}\ a}\ [\text{return}]$$

5.2.2 Edit

Met `edit` kan gebruikersinvoer gesimuleerd worden:

```
edit :: [Maybe a] -> Task a
```

Tijdens een iteratie bekijkt `edit` het eerste element uit de lijst. Als het eerste element uit de lijst `Just` a is, heeft de edit-taak de instabiele waarde a:

$$\frac{}{\texttt{edit}\ [\texttt{Just}\ a : xs] \downarrow \texttt{Val}\ a\ \texttt{Unstable}}\ \text{(editJust)}$$

Als het eerste element `Nothing` is of als de lijst leeg is, heeft de edit-taak geen waarde:

$$\frac{}{\texttt{edit}\ [\texttt{Nothing} : xs] \downarrow \texttt{NoVal}}\ (\text{editNothing}) \qquad \frac{}{\texttt{edit}\ [] \downarrow \texttt{NoVal}}\ (\text{edit}[])$$

Nadat de edit-taak het eerste element van de rij heeft gesimuleerd, moeten de overige elementen uit de rij gesimuleerd worden:

$$\frac{}{\texttt{edit}\ [x : xs] \Rightarrow \texttt{edit}\ xs}\ [\text{edit}]$$

Als de rij leeg is, evolueert `edit []` naar zichzelf:

$$\frac{}{\texttt{edit}\ [] \Rightarrow \texttt{edit}\ []}\ [\text{edit}[]]$$

5.2.3 Throw

De programmeur kan een exceptie genereren met een throw-taak:

```
throw :: E -> Task a
```

`throw` e is een taak die de exceptie e gooit:

$$\frac{}{\texttt{throw}\ e \lightning e}\ \{\text{throw}\}$$

5.2.4 At

Definitie 4. $\rightarrow_\lambda$ is de lazy reductie van Clean.

We zullen aannemen dat deze reductie deterministisch en terminerend is.

```
(@) infixl 1 :: (Task a) (a -> b) -> Task b
```

De `@` kan worden gebruikt om de waarde van een taak te transformeren. Als t de waarde `Val` a s produceert en $f\ a$ reduceert naar b, dan produceert t `@` f de waarde `Val` b s:

$$\frac{t \downarrow \texttt{Val}\ a\ s \qquad f\ a \rightarrow_\lambda b}{t\ \texttt{@}\ f \downarrow \texttt{Val}\ b\ s}\ (\text{@Value})$$

Als t geen waarde produceert, dan produceert t `@` f ook geen waarde:

$$\frac{t \downarrow \texttt{NoVal}}{t \texttt{ @ } f \downarrow \texttt{NoVal}} \text{ (@NoVal)}$$

Met de volgende regel kan de deeltaak binnen de `@` transformeren:

$$\frac{t \Rightarrow t'}{t \texttt{ @ } f \Rightarrow t' \texttt{ @ } f} \text{ [@]}$$

Excepties worden gepropageerd:

$$\frac{t \lightning e}{t \texttt{ @ } f \lightning e} \text{ \{@\}}$$

5.2.5 Bind

De programmeur kan taken sequentieel laten uitvoeren met de monadische bind operator:

```
(>>=) infixl 1 :: (Task a) (a -> Task b) -> Task b
```

t `>>=` f is een taak die eerst t uitvoert. Als t geen stabiele waarde heeft, heeft t `>>=` f een `NoVal` als waarde:

$$\frac{t \downarrow \texttt{Val } a \texttt{ Unstable}}{t \texttt{ >>= } f \downarrow \texttt{NoVal}} \text{ (>>=Unst)}$$

$$\frac{t \downarrow \texttt{NoVal}}{t \texttt{ >>= } f \downarrow \texttt{NoVal}} \text{ (>>=NoVal)}$$

Indien t wel een stabiele waarde heeft, zeg `Val` a `Stable`, zal t `>>=` f de taak f a gaan uitvoeren:

$$\frac{t_1 \downarrow \texttt{Val } a \texttt{ Stable} \qquad f\ a \rightarrow_\lambda t_2 \qquad t_2 \downarrow v}{t_1 \texttt{ >>= } f \downarrow v} \text{ (>>=Stable)}$$

Wanneer de geproduceerde waarde van t geen stabiele waarde is, verandert t `>>=` f in t' `>>=` f als t verandert in t':

$$\frac{t \downarrow \texttt{Val}\ a\ \texttt{Unstable} \qquad t \Rightarrow t'}{t \texttt{ >>= } f \Rightarrow t' \texttt{ >>= } f}\ [\texttt{>>=}\text{Unst}]$$

$$\frac{t \downarrow \texttt{NoVal} \qquad t \Rightarrow t'}{t \texttt{ >>= } f \Rightarrow t' \texttt{ >>= } f}\ [\texttt{>>=}\text{NoVal}]$$

Indien t wel een stabiele waarde heeft, zeg `Val` a `Stable`, zal t `>>=` f de taak $f\ a$ gaan uitvoeren. De regel die hierbij hoort, is:

$$\frac{t_1 \downarrow \texttt{Val}\ a\ \texttt{Stable} \qquad f\ a \rightarrow_\lambda t_2 \qquad t_2 \Rightarrow t_3}{t_1 \texttt{ >>= } f \Rightarrow t_3}\ [\texttt{>>=}\text{Stable}]$$

Excepties worden gepropageerd:

$$\frac{t \lightning e}{t \texttt{ >>= } f \lightning e}\ \{\texttt{>>=}\}$$

Het kan voorkomen dat er een exceptie optreedt op het moment dat er overgestapt wordt naar de volgende taak. In dat geval wordt de exceptie ook gepropageerd:

$$\frac{t_1 \downarrow \texttt{Val}\ a\ \texttt{Stable} \qquad f\ a \rightarrow_\lambda t_2 \qquad t_2 \lightning e}{t_1 \texttt{ >>= } f \lightning e}\ \{\texttt{>>=}\text{t}_2\}$$

5.2.6 Step

De `>>*` kunnen we gebruiken voor sequentiële compositie.

```
((>>*) infixl 1 :: (Task a) (Value a -> Maybe (Task b), E -> Maybe (Task b))
                   -> Task b
```

Als t_1 de waarde v heeft en $f\ v$ reduceert naar `Just` t_2, dan evolueert t_1 `>>*` (f, g) naar de vervolgtaak van t_2:

$$\frac{t_1 \downarrow v \qquad f\ v \rightarrow_\lambda \texttt{Just}\ t_2 \qquad t_2 \Rightarrow t_3}{t_1 \texttt{ >>* } (f, g) \Rightarrow t_3}\ [\texttt{>>*}\text{vJust}]$$

De waarde van t_1 `>>*` (f, g) is dan de waarde van t_2:

$$\frac{t_1 \downarrow v \quad f\ v \to_\lambda \texttt{Just}\ t_2 \quad t_2 \downarrow v'}{t_1 \texttt{>>*}\ (f, g) \downarrow v'}\ (\texttt{>>*vJust})$$

Als t_2 een exceptie gooit, wordt deze gepropageerd:

$$\frac{t_1 \downarrow v \quad f\ v \to_\lambda \texttt{Just}\ t_2 \quad t_2 \lightning e}{t_1 \texttt{>>*}\ (f, g) \lightning e}\ \{\texttt{>>*vJust}\}$$

Als t de waarde v heeft en $f\ v$ reduceert naar `Nothing`, dan stapt t `>>*` (f, g) niet over naar de volgende taak. De subtaak t evolueert binnen t `>>*` (f, g):

$$\frac{t \downarrow v \quad f\ v \to_\lambda \texttt{Nothing} \quad t \Rightarrow t'}{t \texttt{>>*}\ (f, g) \Rightarrow t' \texttt{>>*}\ (f, g)}\ [\texttt{>>*Nothing}]$$

In dit geval heeft t `>>*` (f, g) geen waarde:

$$\frac{t \downarrow v \quad f\ v \to_\lambda \texttt{Nothing}}{t \texttt{>>*}\ (f, g) \downarrow \texttt{NoVal}}\ (\texttt{>>*Nothing})$$

Als t_1 exceptie e gooit en $g\ e$ reduceert naar `Just` t_2, dan evolueert t_1 `>>*` (f, g) naar t_2:

$$\frac{t_1 \lightning e \quad g\ e \to_\lambda \texttt{Just}\ t_2 \quad t_2 \Rightarrow t_3}{t_1 \texttt{>>*}\ (f, g) \Rightarrow t_3}\ [\texttt{>>*eJust}]$$

t_1 `>>*` (f, g) heeft in dat geval de waarde van t_2:

$$\frac{t_1 \lightning e \quad g\ e \to_\lambda \texttt{Just}\ t_2 \quad t_2 \downarrow v}{t_1 \texttt{>>*}\ (f, g) \downarrow v}\ (\texttt{>>*eJust})$$

Als t_2 een nieuwe exceptie gooit, wordt deze gepropageerd:

$$\frac{t_1 \lightning e \quad g\ e \to_\lambda \texttt{Just}\ t_2 \quad t_2 \lightning e'}{t_1 \texttt{>>*}\ (f, g) \lightning e'}\ \{\texttt{>>*eJust}\}$$

Als $g\ e$ naar `Nothing` reduceert, wordt de exceptie gepropageerd:

$$\frac{t \lightning e \quad g\ e \to_\lambda \texttt{Nothing}}{t \texttt{>>*}\ (f, g) \lightning e}\ \{\texttt{>>*Nothing}\}$$

5.2.7 Or

`(-||-)` **infixr** `3 :: (Task a) (Task a) -> Task a`

De semantiek van de `-||-`, spreek uit 'or', is vrij ingewikkeld.
Als de linkertaak een exceptie geeft, wordt deze direct gepropageerd:

$$\frac{t_1 \lightning e}{t_1 \texttt{ -||- } t_2 \lightning e} \{\text{e-||-}\}$$

Als de linkertaak een waarde heeft en de rechtertaak een exceptie gooit, wordt die exceptie gepropageerd:

$$\frac{t_1 \downarrow v \quad t_2 \lightning e}{t_1 \texttt{ -||- } t_2 \lightning e} \{\text{-||-e}\}$$

Als de waarde van de linkertaak stabiel is, heeft de samengestelde taak deze waarde:

$$\frac{t_1 \downarrow \texttt{Val } a \texttt{ Stable} \quad t_2 \downarrow v}{t_1 \texttt{ -||- } t_2 \downarrow \texttt{Val } a \texttt{ Stable}} \text{ (S-||-)}$$

Als de waarde van de linkertaak een `NoVal` is, wordt de waarde van de rechtertaak doorgegeven:

$$\frac{t_1 \downarrow \texttt{NoVal} \quad t_2 \downarrow v}{t_1 \texttt{ -||- } t_2 \downarrow v} \text{ (N-||-)}$$

Als de waarde van de rechtertaak een `NoVal` is, wordt de waarde van de linkertaak doorgegeven:

$$\frac{t_1 \downarrow v \quad t_2 \downarrow \texttt{NoVal}}{t_1 \texttt{ -||- } t_2 \downarrow v} \text{ (-||-N)}$$

Als de linkertaak een instabiele waarde heeft en de rechtertaak een stabiele waarde, dan wordt de stabiele waarde doorgegeven:

$$\frac{t_1 \downarrow \texttt{Val } a_1 \texttt{ Unstable} \quad t_2 \downarrow \texttt{Val } a_2 \texttt{ Stable}}{t_1 \texttt{ -||- } t_2 \downarrow \texttt{Val } a_2 \texttt{ Stable}} \text{ (-||-*)}$$

We hebben nu één geval nog niet gehad: het geval waarbij t_1 en t_2 beide een instabiele waarde hebben. In dat geval hangt het ervan af welke taak als laatst is bewerkt door een gebruiker. Het is echter erg lastig om deze notie van 'laatst bewerkt zijn' te formaliseren. Daarom is deze notie als een meta-eigenschap geformuleerd:

$$\frac{t_1 \downarrow \texttt{Val}\ a_1\ \texttt{Unstable} \qquad t_2 \downarrow \texttt{Val}\ a_2\ \texttt{Unstable}}{t_1\ \texttt{-||-}\ t_2 \downarrow \texttt{Val}\ a_1\ \texttt{Unstable}}\ (\texttt{-||->})\text{als } t_1 \text{ het laatst bewerkt is}$$

$$\frac{t_1 \downarrow \texttt{Val}\ a_1\ \texttt{Unstable} \qquad t_2 \downarrow \texttt{Val}\ a_2\ \texttt{Unstable}}{t_1\ \texttt{-||-}\ t_2 \downarrow \texttt{Val}\ a_2\ \texttt{Unstable}}\ (\texttt{-||-<})\text{als } t_2 \text{ het laatst bewerkt is}$$

In tabel 5.1 staat een schematische weergave van welke waarde er wordt doorgegeven door t_1 `-||-` t_2. In de linkerkolom staat of de waarde van t_1 stabiel of instabiel is of dat t_1 geen waarde (NoVal) heeft. In de bovenste rij staat of de waarde van t_1 (links) of de waarde van t_2 (rechts) wordt doorgegeven.

t_1 \ t_2	NoVal	Instabiel	Stabiel
Noval	Noval	rechts	rechts
Instabiel	links	afhankelijk	rechts
Stabiel	links	links	links

Tabel 5.1: De waarde doorgegeven door t_1 `-||-` t_2

Zoals te zien is, heeft de linkertaak voorrang op het moment dat beide taken een stabiele waarde hebben.

Op het moment dat een van de twee taken stabiel is, wordt deze waarde ook in het vervolg doorgegeven:

$$\frac{t_1 \downarrow \texttt{Val}\ a\ \texttt{Stable} \qquad t_2 \downarrow v}{t_1\ \texttt{-||-}\ t_2 \Rightarrow \texttt{return}\ a}\ [\text{S-||-}]$$

$$\frac{t_1 \downarrow \texttt{NoVal} \qquad t_2 \downarrow \texttt{Val}\ a\ \texttt{Stable}}{t_1\ \texttt{-||-}\ t_2 \Rightarrow \texttt{return}\ a}\ [\text{N-||-S}]$$

$$\frac{t_1 \downarrow \texttt{Val}\ a_1\ \texttt{Unstable} \qquad t_2 \downarrow \texttt{Val}\ a_2\ \texttt{Stable}}{t_1\ \texttt{-||-}\ t_2 \Rightarrow \texttt{return}\ a_2}\ [\text{U-||-S}]$$

Je ziet dat de linkertaak weer voorrang heeft op het moment dat beide taken een stabiele waarde hebben.

In de andere gevallen evolueren de deeltaken binnen de samengestelde taak:

$$\frac{t_1 \downarrow \texttt{Val}\ a_1\ \texttt{Unstable} \quad t_2 \downarrow \texttt{Val}\ a_2\ \texttt{Unstable} \quad t_1 \Rightarrow t_1' \quad t_2 \Rightarrow t_2'}{t_1\ \texttt{-||-}\ t_2 \Rightarrow t_1'\ \texttt{-||-}\ t_2'}\ [\text{U-||-U}]$$

$$\frac{t_1 \downarrow \texttt{NoVal} \quad t_2 \downarrow \texttt{Val}\ a_2\ \texttt{Unstable} \quad t_1 \Rightarrow t_1' \quad t_2 \Rightarrow t_2'}{t_1\ \texttt{-||-}\ t_2 \Rightarrow t_1'\ \texttt{-||-}\ t_2'}\ [\text{N-||-U}]$$

$$\frac{t_1 \downarrow \texttt{Val}\ a_1\ \texttt{Unstable} \quad t_2 \downarrow \texttt{NoVal} \quad t_1 \Rightarrow t_1' \quad t_2 \Rightarrow t_2'}{t_1\ \texttt{-||-}\ t_2 \Rightarrow t_1'\ \texttt{-||-}\ t_2'}\ [\text{U-||-N}]$$

$$\frac{t_1 \downarrow \texttt{NoVal} \quad t_2 \downarrow \texttt{NoVal} \quad t_1 \Rightarrow t_1' \quad t_2 \Rightarrow t_2'}{t_1\ \texttt{-||-}\ t_2 \Rightarrow t_1'\ \texttt{-||-}\ t_2'}\ [\text{N-||-N}]$$

5.2.8 Parallel

`parallel :: [Task a] -> Task [Value a]`

`parallel` t is een taak die als waarde alle waardes van de subtaken heeft:

$$\frac{t_1 \downarrow v_1 \quad ... \quad t_n \downarrow v_n}{\texttt{parallel}\ [t_1, ..., t_n] \downarrow \texttt{Val}\ [v_1, ..., v_n]\ parval(v_1, ..., v_n)}\ (\text{par})$$

$$parval(V) = \begin{cases} \texttt{Stable} & \text{als } \forall v \in V \exists a : v = \texttt{Val}\ a\ \texttt{Stable} \\ \texttt{Unstable} & \text{anders} \end{cases}$$

Je ziet dat deze waarde alleen stabiel is als alle waardes van de subtaken ook stabiel zijn.

Met de volgende regel kunnen de deeltaken binnen de `parallel` evolueren:

$$\frac{t_1 \Rightarrow t_1' \quad ... \quad t_n \Rightarrow t_n'}{\texttt{parallel}\ [t_1, ..., t_n] \Rightarrow \texttt{parallel}\ [t_1', ..., t_n']}\ [\text{par}]$$

Wanneer één van de taken een exceptie genereert, wordt deze gepropageerd:

$$\frac{t_1 \downarrow v_1 \quad ... \quad t_{n-1} \downarrow v_{n-1} \quad t_n \lightning e}{\texttt{parallel}\ [t_1, ..., t_n, ..., t_{n+m}] \lightning e}\ \{\text{par}\}$$

Uit deze regel is op te maken: als er meerdere excepties optreden, wordt de exceptie van de eerste taak in de lijst die een exceptie gooit gepropageerd.

5.2.9 Big-step semantiek

De $\Rightarrow$relatie beschrijft hoe een taak verandert tijdens één iteratie. Om aan te geven hoe de executie van een programma verloopt over meerdere iteraties, definiëren we de relatie $\mapsto_i$. $t_1 \mapsto_i t_2$ geeft aan dat taak t_1 na i iteraties geëvolueerd is in taak t_2. $t \mapsto_i t'$ wil dus zeggen dat er $t_1, ..., t_{i-1}$ zijn zodat:

$$t \Rightarrow t_1 \Rightarrow ... \Rightarrow t_{i-1} \Rightarrow t'$$

Een taak evolueert altijd in 0 iteraties naar zichzelf:

$$\frac{}{t \mapsto_0 t} \text{ bigstep1}$$

Als een taak t_1 in i iteraties evolueert naar t_2 en t_2 evolueert naar t_3, kunnen we concluderen dat t_1 in $i + 1$ iteraties evolueert naar t_3:

$$\frac{t_1 \mapsto_i t_2 \qquad t_2 \Rightarrow t_3}{t_1 \mapsto_{i+1} t_3} \text{ bigstep2}$$

5.2.10 Equivalentie

Definitie 5. We noemen twee taken equivalent als ze altijd dezelfde waarden en excepties hebben. Notatie: $t_1 \simeq t_2$. $t_1 \simeq t_2$ geldt als voor alle t_1', t_2', i, v, e aan de volgende voorwaarden wordt voldaan:

- $t_1 \mapsto_i t_1'$ en $t_1' \downarrow v$ geldt precies als $t_2 \mapsto_i t_2'$ en $t_2' \downarrow v$ geldt
- $t_1 \mapsto_i t_1'$ en $t_1' \lightning e$ geldt precies als $t_2 \mapsto_i t_2'$ en $t_2' \lightning e$ geldt

Hoofdstuk 6

Informele uitleg correctheid

Hieronder geef ik een intuïtieve uitleg waarom de deductieve semantiek overeenkomt met de implementatiesemantiek.

De deductieve semantiek is een semantiek van iTasks', terwijl de implementatiesemantiek het systeem iTasks beschrijft. Hiertussen valt geen correctheid te bewijzen. Het is wel mogelijk om te bewijzen dat de semantiek van iTasks' overeenkomt met de semantiek van iTasks na de vertalingen beschreven in hoofdstuk 4.

Het type van `Task` in de implementatiesemantiek is:

```
:: Task a :== Event -> *State -> *(Reduct a, Responses, *State)
:: Reduct a = Reduct (TaskResult a) (Task a)
:: TaskResult a = ValRes TimeStamp (Value a)
                | ∃e: ExcRes e & iTasks e
:: TimeStamp :== Int
```

De taak in het reduct wordt de continuation genoemd. De `TimeStamp` van een waarde is het tijdstip waarop het event dat de waarde heeft gemaakt is binnengekomen.

De $\downarrow$ relatie geeft aan welke waarde een taak heeft. In de implementatiesemantiek zit de waarde in de taskresult van het reduct. Deze relatie is dus correct als voor alle taken t' geldt: als $t' \downarrow v$, dan evolueert t *event* s in een 3-tupel, waarvan het `TaskResult` er uit ziet als `ValRes` i v voor een zekere timestamp i. t is hier de vertaling van t' van iTasks' naar de implementatiesemantiek en *event* is de gebruikersinvoer die mogelijk gecodeerd is in t'. Als er geen gebruikersinvoer gedefinieerd is in t', dat wil zeggen als er geen editor aanwezig is in t', is *event* een willekeurige waarde.

De $\Rightarrow$ relatie geeft aan naar welke taak een gegeven taak evolueert. In de implementatiesemantiek zit de geëvolueerde taak in de continuation. Deze relatie is dus correct als voor alle taken t'_0 en t'_1 geldt: als $t'_0 \Rightarrow t'_1$, dan evolueert t_0 *events* s in een 3-tupel, waarvan de continuation gelijk is aan t_1. De taken zonder accent zijn weer de vertaling van de taken met accent van iTasks' naar de implementatiesemantiek.

De $\lightning$ relatie geeft aan welke exceptie een taak gooit. In de implementatiesemantiek zit de waarde in de taskresult van het reduct. Deze relatie is dus correct als voor alle taken t' geldt: als $t' \lightning e$, dan evolueert t *event* s in een 3-tupel, waarvan het `TaskResult` er uit ziet als `ExcRes (E` e`)`. t is de vertaling van t' van iTasks' naar de implementatiesemantiek.

Ik zal per taak beargumenteren dat de regels die horen bij de taak overeen komen met de implementatiesemantiek.

return De implementatie van return in de implementatiesemantiek is:

```
return :: a -> Task a
return va ev st=:{timestamp = t} = stable t va ev st

stable :: TimeStamp -> a -> Task a
stable t va _ st = (Reduct (ValRes t (Val va Stable)) (stable t va), [], st)
```

`return` pakt dus de huidige tijd uit de state en geeft die als parameter mee aan de `stable`, alsook de opgegeven `va`. `stable` heeft vervolgens `Val va Stable` als waarde en reduceert naar zichzelf. Met regel (return) kan worden afgeleid dat `return` de opgegeven a als waarde heeft. De regel [return] zegt dat `return` naar zichzelf evolueert. Deze regels komen dus overeen met de implementatiesemantiek.

edit In de implementatiesemantiek wordt ook een `edit` behandeld als generieke editor, maar deze is totaal anders dan de `edit` in iTasks':

```
edit :: String -> l -> RWShared r w -> (l -> r -> Maybe a) -> Task a
                                    | iTask l & iTask r
```

Wat wel overeenkomt, is dat beide taken een onstabiele waarde hebben als ze een waarde hebben. Op regel 13 van de definitie van de `edit` staat het reduct van `edit`. De `TaskValue` is gedefinieerd als `ValRes nt (toValue (cv nlv sr))`. `toValue` is een functie die van een `Just a` een `Val a Unstable` maakt en van een `Nothing` een `NoVal`. Dit komt overeen met (editJust) en (editNothing) in de deductieve semantiek.

throw De definitie van `throw` in de implementatiesemantiek is[1]

```
throw :: e -> Task a | iTask e
throw e _ st = (Reduct (ExcRes e) (throw e), [], st)
```

De regel {throw} komt hier mee overeen.

[1] `throw` is in de implementatiesemantiek onbedoeld te restrictief getypeerd als `throw :: e -> Task e | iTask e`.

at De definitie van `@` in de implementatiesemantiek is:

```
1  (@) infixl 1 :: Task a -> (a -> b) -> Task b
2  (@) ta f = t @? (\aval -> case aval of
3                            NoVal   = NoVal
4                            Val a s = Val (f a) s)
5
6  (@?) infixl 1 :: Task a -> (Value a -> Value b) -> Task b
7  (@?) ta f ev st
8  = case ta ev st of
9      (Reduct (ValRes t aval) nta, rsp, nst)
10         = case f aval of
11             Val b Stable = stable t b ev nst
12             bval         = (Reduct (ValRes t bval) (nta @? f), rsp, nst)
13     (Reduct (ExcRes e) _, _, nst)
14         = throw e ev nst
```

De `@` is gedefinieerd in termen van de `@?`. De `@?` transformeert de waarde van de subtaak (regels 11 en 12). De `stable` wordt gebruikt om er voor te zorgen dat stabiele waardes stabiel blijven[2]. Excepties worden gepropageerd (regels 13 en 14).

Uit regel 4 kunnen we halen dat (@NoVal) correct is: als de subtaak een `NoVal` heeft, heeft de gecompositioneerde taak dat ook. Als de subtaak een waarde heeft, past de `@` de `f` toe op de waarde. Dit komt overeen met (@Value). Excepties worden gepropageerd. Dit komt overeen met {@}. De regel [@] zegt dat de subtaak kan evolueren binnen de gecompositioneerde taak. Dit kan je zien aan `(nta @? f)` op regel 12. De regels van de `@` komen dus overeen met de implementatiesemantiek.

[2]De `@?` in de implementatie van iTasks die ik heb gebruikt, houd zich niet aan de definitie in de implementatiesemantiek. De `@?` in de implementatie dwingt niet af dat de waarde van t `@?` f hetzelfde blijft zodra t een waarde v heeft waarbij f v reduceert naar een stabiele waarde.

step De definitie van >>* in de implementatiesemantiek is:

```
(>>*) infixl 1 :: Task a -> [TaskStep a b] -> Task b | iTask a & iTask b
(>>*) ta steps = newTask (step1 ta)
where
    step1 ta tn t ev st
    # (Reduct tval nta, rsp, st) = ta ev st
    = hd( findTriggers tval
       ++ findActions  tval ev
       ++ [step1‘ tval nta rsp]
       ) ev st
    where
    findTriggers (ExcRes e)   = catchers e ++ [throw e]
    findTriggers (ValRes _ v) = values v

    findActions (ValRes _ v) (ActionEvent tid act)
    | tid == tn               = actions act v
    findActions _ _           = []

    step1‘ (ValRes _ v) nta rsp _ st
        = (Reduct no_tval (step1 nta tn t), nrsp ++ rsp, st)
    where
        no_tval = ValRes t NoVal
        as      = [(a,p v) \\ OnAction a p _<-steps]
        nrsp    = if (isEmpty as) [] [(tn, ActionResponse as)]

catchers    e = [etb e \\ OnException etb  <- steps               ]
values      v = [atb v \\ OnValue p atb    <- steps |           p v]
actions act v = [atb v \\ OnAction a p atb <- steps | act==a && p v]

:: TaskStep a b
   = OnAction Action (Predicate a) (NextTask a b)
   | OnValue         (Predicate a) (NextTask a b)
   | E.e: OnException (e -> Task b) & iTask e
:: Predicate a  :== Value a -> Bool
:: NextTask a b :== Value a -> Task b

newTask :: (TaskNo -> TimeStamp -> Task a) -> Task a
newTask ta ev st=:{taskNo = no, timeStamp = t}
  = ta no t ev {st & taskNo = no+1}
```

Het type van de >>* in de implementatiesemantiek verschilt van het type in iTasks. De lijst van `TaskCont a (Task b)` is in de implementatiesemantiek een lijst van `TaskStep a b`. In iTasks wordt het overstappen op de volgende taak bij de `OnValue` en de `OnAction` aangegeven door de functie te laten reduceren in een `Just` t, waarbij t de taak is waar naartoe geëvalueerd wordt. Het niet overstappen op de volgende taak wordt weergeven door de functie te laten reduceren naar `Nothing`. In de implementatiesemantiek geeft het `Predicate` aan of moet worden overgestapt op de volgende taak. De `NextTask`

geeft aan welke taak dit dan zou moeten zijn.

`newTask` voorziet een taak van een uniek taaknummer en het tijdstip van de eerste iteratie van de taak (regel 2). Bij regel 5 wordt de subtaak geëvalueerd. Eerst wordt gezocht naar 'triggers' (regel 6). Als de subtaak een exceptie e gooit, wordt gekeken of de exceptie door een van de `OnException`s wordt gevangen (regel 11 en 25).

Indien de exceptie wordt gevangen door een g `:: e -> Task b`, wordt g e geëvalueerd en het resultaat hiervan wordt doorgespeeld naar de gecompositioneerde taak (regels 6 t/m 9). Hieruit kunnen we concluderen dat [>>*eJust]correct is. Als g e een waarde v heeft (g e `ev st = (Reduct (ValRes` ... v ..., ..., ...)), is dit de waarde die de gehele taak heeft. Dit komt dus overeen met (>>*eJust). Als g e een exceptie gooit, wordt deze door de gecombineerde taak gepropageerd. Dit komt overeen met {>>*eJust}.

Indien de exceptie niet wordt gevangen door een van de `OnException`s, wordt de exceptie gepropageerd (regel 11). In iTasks' wordt het niet vangen van een exceptie aangegeven door de g te laten reduceren naar `Nothing`. Hieruit blijkt de correctheid van {>>*Nothing}.

Als de subtaak een waarde heeft, wordt gekeken of één van de predicaten van de `OnValue`s bij deze waarde een `True` oplevert (regel 12 en 26). Is dat het geval, dan wordt overgestapt op de taak gespecificeerd in de `NextTask`. Op eenzelfde manier als bij de excepties volgt hier de correctheid van (>>*vJust), [>>*vJust] en {>>*vJust} uit.

Als het lijstje van triggers op regel 6 leeg is, wordt gekeken of er een `Action` is uitgevoerd. Omdat de actions in iTasks' zijn weggelaten, hoeven we hier niet naar te kijken.

Als er ook geen actions worden gevonden, heeft de gecompositioneerde taak een `NoVal` als waarde (regels 8, 18 en 19). De subtaak evolueert binnen de gecompositioneerde taak. Dit komt dus overeen met (>>*Nothing) en [>>*Nothing].

bind De definitie van `>>=` in de implementatiesemantiek is gedefinieerd in termen van `>>*`:

```
(>>=) infixl 1 :: Task a -> (a -> Task b) -> Task a | iTask a & iTask b
(>>=) ta atb = ta >>* [OnValue isStable (atb o getValue)]
```

Als de subtaak een instabiele waarde of geen waarde heeft, zorgt de `>>*` er voor dat de gecompositioneerde taak geen waarde heeft. De subtaak zal binnen de gecompositioneerde taak evolueren. De regels (>>=Unst), (>>=NoVal), [>>=Unst] en [>>=NoVal]komen hiermee overeen.

Als de subtaak een stabiele waarde `Value` a `Stable` heeft, reduceert `isStable (Value` a `Stable)` naar `True`. De gecompositioneerde taak zal daardoor overstappen op de volgende taak. `getValue` haalt de a uit de waarde, waarna de gecompositioneerde taak evolueert naar `atb` a. Hieruit volgt de correctheid van (>>=Stable), [>>=Stable] en {>>=t_2}.

Excepties worden door de >>= niet gevangen in een OnException. Als de subtaak een exceptie gooit, zal deze door de >>* gepropageerd worden. Hieruit volgt de correctheid van {>>=}.

parallel De definitie van parallel in de implementatiesemantiek is:

```
1  parallel :: [ParallelTask a] -> Task [(TimeStamp,Value a)] | iTask a
2  parallel ptas ev st
3  # (stt,st) = createTaskList ptas st
4  = parallel‘ stt ev st
5
6  createTaskList :: [ParallelTask a] -> *State -> *(SharedTaskList a,*State)
7                 | iTask a
8  createTaskList ptas st=:{timeStamp = t}
9  # (stt,st) = createShared [] st
10 = (stt,stt.set [  (pid,Reduct (ValRes t NoVal) (pta stt))
11                \\ pta <- ptas & pid <- [0..]
12                ] st)
13
14 parallel‘ :: SharedTaskList a -> Task [(TimeStamp,Value a)] | iTask a
15 parallel‘ stt ev st
16 = case evalParTasks stt ev st of
17     (Left (ExcRes e),st) = throw e ev st
18     (Right rsp,st)
19     # (values,st) = get_task_values stt st
20     # maxt        = foldr max 0 (map fst values)
21     | all (isStable o snd) values = stable maxt values ev st
22     | otherwise   = (Reduct (ValRes maxt (Val values Unstable))
23                             (parallel‘ stt),rsp,st)
24
25 evalParTasks :: SharedTaskList a -> Event -> *State
26                  -> *(Either (TaskResult a) Responses,*State) | iTask a
27 evalParTasks stt ev st
28 # (tt,st) = stt.get st
29 = foldl (evalParTask stt ev) (Right [],st) tt
30
31 evalParTask :: SharedTaskList a -> Event
32             -> *(Either (TaskResult a) Responses,*State)
33             -> (Pid a,Reduct a)
34             -> *(Either (TaskResult a) Responses,*State)
35 evalParTask stt ev (Right rsp,st) (pid,Reduct _ ta)
36 # (newr,nrsp,st)     = ta ev st
37 # (Reduct ntval nta) = newr
38 | isExcRes ntval     = (Left ntval,st)
39 # (_,st)             = updateShared (updateFM (pid,newr)) stt st
40 = (Right (nrsp ++ rsp),st)
41 evalParTask _ _ (Left e,st) _
42 = (Left e,st)
43
```

```
44 get_task_values :: SharedTaskList a -> *State
45            -> *([(TimeStamp, Value a)],*State)
46 get_task_values stt st
47 # (tt,st) = stt.get st
48 = ([(t,val) \\ (_,Reduct (ValRes t val) _) <- tt],st)
```

In de implementatiesemantiek kunnen subtaken van de `parallel` de waardes zien die de andere taken tijdens dezelfde iteratie hebben. In iTasks' is dit weggelaten. We zullen daarom alleen naar de delen van de implementatiesemantiek kijken die niet gaan over het delen van waardes. Ook zullen we de `TimeStamp` in het resultaat van `parallel` in de implementatiesemantiek negeren, omdat deze niet voorkomt in iTasks'.

Een `SharedTaskList` is een lijst van reducten. Deze reducten worden met Shared Data Sources gedeeld onder de subtaken. Tijdens executie bevat de lijst het reduct dat is opgeleverd tijdens de laatste iteratie. Dit is dus de laatste taakwaarde, alsook de laatste continuation.

De `parallel` stopt de taken in de `SharedTaskList` en reduceert daarna in `parallel‘` (regel 3 en 4). De `parallel‘` evalueert alle subtaken met `evalParTasks` (regel 16). Deze laatste is een functie die naar een term van type `*(Either (TaskResult a) Responses, *State)` reduceert. Het idee van `Either (TaskResult a) Responses` is dat dit een `Right responses` is als geen van de subtaken een exceptie gooit. `evalParTasks` haalt de lijst van taken uit de toestand (regels 28). Daarna laat het `evalParTask` over alle taken itereren (regel 29). `evalPartask` krijgt vier argumenten: de lijst van taken, het `event` waarop wordt gereageerd, een tuple van de responses van de taken die al uitgevoerd zijn (of de exceptie die opgetreden is) en de toestand, en het reduct van de taak waar we nu aangekomen zijn in de lijst met taken.

Als er al eerder een exceptie is opgetreden, is het derde argument van `evalParTask` een `Left (ExcRes` e`)`. De exceptie wordt dan verder doorgegeven (regels 41 en 42). `evalParTasks` reduceert hierdoor in een `Left (ExcRes` e`)`. `parallel‘` zorgt ervoor dat de gecombineerde taak de exceptie gooit (regel 17). In het geval dat de vorige taken geen exceptie hebben gegooid (regel 35), wordt de taak uit de lijst waar we nu zijn aangekomen geëvalueerd (regel 36). Als dit een exceptie geeft, wordt deze verder doorgegeven middels een `Left` (regel 38). Deze constructie zorgt er dus voor dat de eerste exceptie die gegooid wordt, direct door de `parallel` gepropageerd wordt. {par} komt hiermee overeen.

Als de subtaak een waarde geeft in plaats van een exceptie, wordt het reduct geregistreerd in de lijst (regel 39). De responses worden toegevoegd aan de vorige responses en worden doorgegeven voor het evalueren van de volgende taak (regel 40).

Wanneer alle subtaken een waarde hebben, wordt dit teruggegeven aan `parallel‘` middels een `Right`. `parallel‘` haalt een lijst met alle taakwaardes uit de `SharedTaskList` (regel 19). Als alle waardes stabiel zijn, reduceert het geheel in een `stable` (regel 21). Zoals we eerder hebben gezien geeft de `stable`

altijd een stabiele waarde. Anders reduceert het geheel in een reduct met een instabiele waarde met de lijst met de taakwaardes van alle subtaken en zichzelf als continuation.

Als alle subtaken stabiel zijn, heeft `parallel` dus een stabiele waarde met het lijstje van alle waardes van alle subtaken. Anders heeft `parallel` een instabiele waarde met het lijstje van alle waardes van alle subtaken. (par) komt hiermee precies overeen.

De implementatiesemantiek komt ook overeen met [par]. Hiermee is de correctheid van `parallel` rond.

or De definitie van `-||-` in de implementatiesemantiek is:

```
1  (-||-) infixr 3 :: Task a -> Task a -> Task a |iTask a
2  (-||-) a b = parallel () [(Embedded, const a), (Embedded, const b)] @? first
3  where
4      first NoVal = NoVal
5      first (Value vs _)
6        = hd (  [v \\ (_, v:=(Val _ Stable)) <- vs]
7             ++ [v \\ (_, v:=(Val _ _)) <- sortBy newer vs]
8             ++ [NoVal]
9             )
10 newer (t1, _) (t2, _) = t1 > t2
```

De `-||-` voert de twee subtaken parallel uit. Zoals we net gezien hebben, gooit de parallel de eerste exceptie wanneer er een exceptie optreedt. Als de linkertaak een exceptie gooit, wordt deze dus naar boven gepropageerd, zelfs als de rechtertaak ook een exceptie gooit. Als linkertaak geen exceptie gooit en de rechtertaak wel, wordt de exceptie van de rechtertaak gepropageerd. De `@?` propageert de exceptie vervolgens verder naar boven. De implementatiesemantiek komt hier dus overeen met {e-||-} en {-||-e}. Bij {-||-e} zorgt $t_1 \downarrow v$ er voor dat de regel alleen geldt als de eerste taak geen exceptie gooit. In stelling 1 wordt bewezen dat de deductieve semantiek het gelijktijdig optreden van een waarde en een exceptie uitsluit.

`first` kiest de waarde uit de lijst van waardes die door de `parallel` wordt teruggegeven. Welke waarde dit is, wordt bepaald aan de hand van de volgende criteria:

1. Als een van de waardes van de subtaken stabiel is, is dit de waarde die de `-||-` heeft. Als beide taken een stabiele waarde hebben, zal de waarde van de linkertaak eerder in de list comprehension op regel 6 staan dan de waarde van de rechtertaak. De waarde van de linkertaak heeft daardoor 'voorrang'.

2. Als beide subtaken geen stabiele waarde hebben, maar een subtaak heeft een instabiele waarde, dan wordt deze instabiele waarde doorgegeven (regel 7). Als beide waardes instabiel zijn, worden de waardes

gesorteerd op hun timestamp door `sortBy newer`. De waarde van de `-||-` is dus de waarde van de taak die als laatst bewerkt is. Hieruit volgt direct de correctheid van (`-||->`) en (`-||-<`).

3. Als beide subtaken geen waarde hebben, heeft de `-||-` geen waarde.

We kijken nu voor de regels (S-||-), (N-||-), (-||-N) en (-||-*) per regel of deze overeenkomt met de implementatiesemantiek.

(S-||-)Als de waarde van de linkertaak stabiel is, zal deze als eerste voorkomen in de lijst op regel 6. We kunnen dus direct al concluderen dat dit de waarde is die zal worden doorgegeven aan de `-||-`.

(N-||-)Als de linkertaak geen waarde heeft en de rechtertaak wel, zal de waarde van de rechtertaak in één van de lijstjes op regel 6 en 7 zitten. Verder zullen er geen andere waardes in deze lijsten zitten. De waarde van de rechtertaak zal dus worden doorgegeven. Als beide subtaken geen waarde hebben, zijn de lijsten op regel 6 en 7 leeg. De `-||-` zal daardoor, net als de rechtertaak, een `NoVal` als waarde hebben. In beide gevallen geldt dus dat de waarde van de samengestelde taak gelijk is aan de waarde van de rechtertaak.

(-||-N)Hiervoor geldt eenzelfde redenering als bij (N-||-).

(-||-*)Als de waarde van de linkertaak instabiel en de waarde van de rechtertaak stabiel is, zal de lijst op regel 6 alleen de stabiele waarde van de rechtertaak bevatten. Deze waarde zal dus worden doorgegeven aan de `-||-`.

Als één van de twee taken een stabiele waarde heeft, zorgt de `@?` ervoor dat de gehele taak reduceert naar een `stable`. De taak blijft daardoor voor de volgende iteraties altijd deze stabiele waarde hebben, ongeacht of de andere taak in de toekomst een stabiele waarde heeft of een exceptie gooit. De linkertaak heeft hier weer voorrang over de rechtertaak. Dit komt overeen met [S-||-], [N-||-S] en [U-||-S].

Als beide taken geen stabiele waarde hebben, evolueren de subtaken binnen de `-||-`. Dit komt overeen met [N-||-N], [U-||-N], [N-||-U] en [U-||-U].

Hoofdstuk 7

Voorbeeldbewijzen en eigenschappen

In dit hoofdstuk bewijs ik met de deductieve semantiek verschillende eigenschappen van iTasks' om de werking van de semantiek te illustreren. Ik bewijs een paar algemene eigenschappen van iTasks in paragraaf 7.1. In paragraaf 7.2 gebruik ik de deductieve semantiek om te beredeneren wat het resultaat is van de executie van het bestelprogramma.

De stellingen die ik in 7.1 heb bewezen, zijn gestoeld op een hypothese en drie beweringen. Van de hypothese en beweringen verwacht ik dat ze gelden, maar ik heb er geen bewijs voor gegeven.

7.1 Algemene eigenschappen

Lemma 1. *Als een taak een waarde heeft, is deze waarde uniek voor de taak. Als een taak een exceptie gooit, is deze exceptie uniek voor de taak. Een taak kan niet zowel een waarde hebben als een exceptie gooien.*

Bewijs. We bewijzen het volgende:

- Als $t \downarrow v$, dan geldt:
  - Er is geen e met $t \lightning e$
  - Als $t \downarrow v'$, dan $v' = v$
- Als $t \lightning e$, dan geldt:
  - Er is geen v met $t \downarrow v$
  - Als $t \lightning e'$, dan $e' = e$

We bewijzen dit met inductie naar de afleiding van $t \downarrow v$ en $t \lightning e$. We geven een schets van het bewijs.

Geval (return): dan `return` $a \downarrow$ `Val` a `Stable`. Er is geen regel met `return` $a \not\downarrow e$ als conclusie, dus `return` a gooit geen exceptie.

Stel `return` $a \downarrow v'$. De enige regel die gebruikt kan zijn om `return` $a \downarrow v'$ te verkrijgen, is (return). Hiermee volgt dat $v' =$ `Val` a `Stable`.

Geval (editJust): Dan `edit` [`Just` $a : xs$] $\downarrow$ `Val` a `Unstable`. Er is geen regel met `edit` [`Just` a : xs] $\not\downarrow e$ als conclusie, dus `edit` [`Just` a : xs] gooit geen exceptie.

Stel `edit` [`Just` a : xs] $\downarrow v'$. De enige regel die gebruikt kan zijn om `edit` [`Just` $a : xs$] $\downarrow v'$ te krijgen, is (editJust), dus $v' =$ `Val` a `Unstable`

Gevallen (editNothing) en (edit[]): op eenzelfde manier als (editJust).

Geval {throw}: Dan `throw` $e \not\downarrow e$. Er is geen regel van de vorm `throw` $e \downarrow v$, dus `throw` e heeft geen waarde.

Stel `throw` $e \not\downarrow e'$. De enige regel die gebruikt kan zijn om dit te verkrijgen, is {throw}, dus $e = e'$.

Geval (@Value): Dan geldt:

$$t \texttt{ @ } f \downarrow \texttt{Val } b\ s \tag{7.1}$$

$$t \downarrow \texttt{Val } a\ s \tag{7.2}$$

$$f\ a \rightarrow_\lambda b \tag{7.3}$$

Als je zou kunnen afleiden dat t `@` f een exceptie gooit, zou je dit moeten afleiden met {@}. De inductiehypothese toegepast op $t \downarrow$ `Val` $a\ s$ zegt dat t geen excepties gooit. Één van de premissen van {@} is $t \not\downarrow e$; dit geeft een tegenspraak. Dus t `@` f gooit geen excepties.

Stel t `@` $f \downarrow v'$. (@NoVal) kan niet toegepast zijn om dit te verkrijgen, want één van de premissen is $t \downarrow$ `Nothing` en de inductiehypothese zegt dat t alleen de waarde `Val` a `Stable` kan hebben. (@Value) is dus de enige regel die toegepast kan zijn om t `@` $f \downarrow v'$ te krijgen. Dus:

$$v' = \texttt{Val } b'\ s' \tag{7.4}$$

$$f\ a' \rightarrow_\lambda b' \tag{7.5}$$

$$t \downarrow \texttt{Val } a'\ s' \tag{7.6}$$

Met de inductiehypothese volgt dat `Val` $a\ s =$ `Val` $a'\ s'$, dus $a = a'$ en $s = s'$. $\rightarrow_\lambda$ is deterministisch, dus met 7.3 en 7.5 volgt $b = b'$. Nu kunnen we concluderen dat $v' =$ `Val` $a'\ s' =$ `Val` $a\ s$.

Gevallen (@NoVal) en {@}: gaan op eenzelfde manier als (@Value).

De gevallen voor de `>>=` worden op een soortgelijke manier bewezen als de gevallen voor de `>>*` en laat ik hier weg.

Geval (>>*vJust): Dan $t \downarrow v$, $f\ v \rightarrow_\lambda$ `Just` t', $t' \downarrow v_2$ en t `>>*` $(f, g) \downarrow v_2$. Als we de inductiehypothese op $t \downarrow v$ en $t' \downarrow v_2$ toepassen, volgt dat {>>*eJust}, {>>*Nothing} en {>>*vJust} niet toepasbaar zijn. Hiermee kunnen we concluderen dat t `>>*` (f, g) geen exceptie gooit.

Stel t `>>*` $(f, g) \downarrow v'$. De inductiehypothese toegepast op $t \downarrow v$ geeft dat (`>>*`eJust) niet toegepast kan zijn om t `>>*` $(f, g) \downarrow v'$ te krijgen. (`>>*`Nothing) kan ook niet toegepast zijn, want $f\ v \rightarrow_\lambda$ `Just` t'. De (`>>*`vJust) moet dus toegepast zijn om t `>>*` $(f, g) \downarrow v'$ te krijgen. Dan $t \downarrow v''$, $f\ v'' \rightarrow_\lambda$ `Just` t'' en $t'' \downarrow v'$. Met de inductiehypothese volgt dat $v = v''$. De reductierelatie $\rightarrow_\lambda$ is deterministisch, dus $t' = t''$. Met de inductiehypothese toegepast op $t' \downarrow v_2$ kunnen we concluderen dat $v_2 = v'$, hetgeen we voor dit geval moesten bewijzen.

De andere gevallen van `>>*` gaan op eenzelfde manier.

Gevallen {e-||-} en {-||-e}: triviaal.

Geval (S-||-): dan $t_1 \downarrow$ `Val` a `Stable`, $t_2 \downarrow v$ en t_1 `-||-` $t_2 \downarrow$ `Val` a `Stable`. Met de inductiehypothese toegepast op $t_1 \downarrow$ `Val` a `Stable` en $t_2 \downarrow v$ volgt vrij snel dat t_1 `-||-` t_2 geen exceptie gooit.

Stel t_1 `-||-` $t_2 \downarrow v'$. Met de inductiehypothese toegepast op $t_1 \downarrow$ `Val` a `Stable` volgt dat (N-||-) en (-||-*) niet toegepast kunnen zijn om t_1 `-||-` $t_2 \downarrow v'$ te krijgen. Er blijven twee regels over:
(S-||-): dan $v' =$ `Val` a' `Stable`, t_1 `-||-` $t_2 \downarrow$ `Val` a' `Stable` en $t_1 \downarrow$ `Val` a' `Stable`. Met de inductiehypothese volgt dat `Val` a `Stable` = `Val` a' `Stable`, dus $v' =$ `Val` a `Stable`.
(-||-N): dan $t_1 \downarrow v'$. Met de inductiehypothese volgt dat $v' =$ `Val` a `Stable`.

Geval (N-||-): dan $t_1 \downarrow$ `NoVal`, $t_2 \downarrow v$ en t_1 `-||-` $t_2 \downarrow v$. Met toepassing van de inductiehypothese kan weer vrij eenvoudig worden afgeleid dat t_1 `-||-` t_2 geen exceptie gooit.

Stel t_1 `-||-` $t_2 \downarrow v'$. Met de inductiehypothese toegepast op $t_1 \downarrow$ `NoVal` volgt dat (S-||-) en (-||-*) niet toegepast kunnen zijn om t_1 `-||-` $t_2 \downarrow v'$ te krijgen. Er blijven twee regels over:
(N-||-): dan $t_2 \downarrow v'$. Met de inductiehypothese volgt dat $v = v'$.
(-||-N): dan $t_1 \downarrow v'$ en $t_2 \downarrow$ `NoVal`. Met de inductiehypothese toegepast op $t_1 \downarrow$ `NoVal` volgt dat $v' =$ `NoVal`. Met de inductiehypothese toegepast op $t_2 \downarrow v$ volgt dat $v =$ `NoVal`. Dus $v = v' =$ `NoVal`.

Geval (-||-N): idem. Geval (-||-*) is vrij makkelijk met inductie te bewijzen.

Geval (-||->): dan $t_1 \downarrow$ `Val` a_1 `Unstable`, $t_2 \downarrow$ `Val` a_2 `Unstable` en t_1 `-||-` $t_2 \downarrow$ `Val` a_1 `Unstable`. Bovendien geldt de meta-eigenschap dat t_1 later voor het laatst bewerkt is dan t_2. Met toepassing van de inductiehypothese kan weer vrij eenvoudig worden afgeleid dat t_1 `-||-` t_2 geen exceptie gooit.

Stel t_1 `-||-` $t_2 \downarrow v'$. Met inductie kan worden bewezen dat t_1 `-||-` $t_2 \downarrow v'$ alleen kan zijn afgeleid door toepassing van (-||->) of (-||-<). De meta-eigenschap van (-||-<) is spreekt de meta-eigenschap van (-||->)echter tegen, waardoor (-||->)als enige regel over blijft. Dan $v' =$ `Val` a_1' `Unstable` en $t_1 \downarrow$ `Val` a_1' `Unstable`. Met de inductiehypothese toegepast op $t_1 \downarrow$ `Val` a_1 `Unstable` volgt dat `Val` a_1 `Unstable` = `Val` a_1' `Unstable`, dus $v' =$ `Val` a_1 `Unstable`.

Geval (-||-<): Eenzelfde argumentatie als bij (-||->).

Geval (par): dan $t_1 \downarrow v_1$, ..., $t_n \downarrow v_n$ en $\texttt{parallel}[t_1, ..., t_n] \downarrow \texttt{Val}\ [v_1, ..., v_n]$ $parval(\{v_1, ..., v_n\})$. Met de inductiehypothese volgt dat geen van de subtaken t_i een exceptie gooit, dus {par} is niet toepasbaar. $\texttt{parallel}[t_1, ..., t_n]$ kan dus geen exceptie gooien.

Stel $\texttt{parallel}[t_1, ..., t_n] \downarrow v'$. De enige regel waarmee dit verkregen kan worden is (par), dus $v' = \texttt{Val}[v'_1, ..., v'_n]parval(\{v'_1, ..., v'_n\})$

Geval {par}: triviaal.

De stelling is hiermee bewezen.

□

Hypothese 1. *Voor alle taken geldt: als alle functietoepassingen $\rightarrow_\lambda$ binnen de taak termineren, heeft de taak een waarde of gooit de taak een exceptie.*

Stelling 1. *Voor elke taak t is precies één van de volgende uitspraken waar:*

- *t heeft precies één waarde en gooit geen exceptie*
- *t gooit precies één exceptie en heeft geen waarde*

Bewijs. Volgens hypothese 1 heeft elke taak t een waarde of gooit het een exceptie. Als t een waarde heeft, zegt lemma 1 dat dit de enige waarde van t is en dat t geen exceptie gooit. Als t een exceptie gooit, zegt lemma 1 dat dit de enige exceptie van t is en dat t geen waarde heeft. □

Bewering. *$\Rightarrow$ is deterministisch. Dat wil zeggen: als $t_1 \Rightarrow t_2$ en $t_1 \Rightarrow t_3$, dan $t_2 = t_3$.*

Bewering. *Elke taak die een waarde heeft, heeft een vervolgtaak. Dat wil zeggen: als $t \downarrow v$, dan $\exists t' : t \Rightarrow\ t'$*

Stelling 2. *Elke taak die een waarde heeft, heeft precies één vervolgtaak.*

Bewijs. Dit is een direct gevolg van hypothese 7.1 en hypothese 7.1. □

Bewering. *$t_1 \simeq t_2$ geldt precies als aan de volgende voorwaarde wordt voldaan:*

- *$t_1 \downarrow v \iff t_2 \downarrow v$*
- *$t_1 \lightning e \iff t_2 \lightning e$*
- *Als $t_1 \Rightarrow t_3$ en $t_2 \Rightarrow t_4$, dan $t_3 \simeq t_4$*

Stelling 3. *Als $f\ a \rightarrow_\lambda t$, dan $\texttt{return}\ a \texttt{ >>= } f \simeq t$*

Bewijs. Stel $f\ a \rightarrow_\lambda t$. Met stelling 1 volgt dat ofwel $t \downarrow v$ ofwel $t \lightning e$ voor unieke v en e.

Geval 1: $t \downarrow v$. Dan kunnen we de volgende afleiding maken:

$$\dfrac{\dfrac{}{\texttt{return}\ a \downarrow \texttt{Val}\ a\ \texttt{Stable}}\ (\text{return}) \qquad f\ a \to_\lambda t \qquad t \downarrow v}{\texttt{return}\ a\ \texttt{>>=}\ f \downarrow v}\ (\texttt{>>=}\text{Stable})$$

Dus $\texttt{return}\ a\ \texttt{>>=}\ f \downarrow v$. Met stelling 1 volgt dat v de enige waarde is die t heeft en dat t geen exceptie gooit. Hiermee kunnen we concluderen dat $\forall v' : \texttt{return}\ a\ \texttt{>>=}\ f \downarrow v' \iff t \downarrow v'$.

Geval 2: $t \lightning e$. Dan kunnen we een soortgelijke afleiding maken:

$$\dfrac{\dfrac{}{\texttt{return}\ a \downarrow \texttt{Val}\ a\ \texttt{Stable}}\ (\text{return}) \qquad f\ a \to_\lambda t \qquad t \lightning e}{\texttt{return}\ a\ \texttt{>>=}\ f \lightning e}\ \{\texttt{>>=}\}$$

Dus $\texttt{return}\ a\ \texttt{>>=}\ f \lightning e$. Op eenzelfde manier als bij geval 1 kunnen we concluderen dat $\forall e' : \texttt{return}\ a\ \texttt{>>=}\ f \lightning e' \iff t \lightning e'$.

In beide gevallen geldt dus dat $\forall v' : \texttt{return}\ a\ \texttt{>>=}\ f \downarrow v' \iff t \downarrow v'$ en $\forall e' : \texttt{return}\ a\ \texttt{>>=}\ f \lightning e' \iff t \lightning e'$. We hoeven dus alleen nog te bewijzen dat $\texttt{return}\ a\ \texttt{>>=}\ f \Rightarrow t' \iff t \Rightarrow t'$. Hierna volgt de conclusie met behulp van hypothese 7.1.

Stel $t \Rightarrow t'$. Dan kunnen we de volgende afleiding maken:

$$\dfrac{\dfrac{}{\texttt{return}\ a \downarrow \texttt{Val}\ a\ \texttt{Stable}}\ (\text{return}) \qquad f\ a \to_\lambda t \qquad t \Rightarrow t'}{\texttt{return}\ a\ \texttt{>>=}\ f \Rightarrow t'}\ [\texttt{>>=}\text{Stable}]$$

Dus $\texttt{return}\ a\ \texttt{>>=}\ f \Rightarrow t'$.

Met stelling 2 volgt dat t en $\texttt{return}\ a\ \texttt{>>=}\ f$ alleen t'als evolutie hebben. Dus $\texttt{return}\ a\ \texttt{>>=}\ f \Rightarrow t' \iff t \Rightarrow t'$. Met hypothese 7.1 volgt nu dat $\texttt{return}\ a\ \texttt{>>=}\ f \simeq t$. □

Stelling 4. *Er is een t zodat t $\texttt{>>=}$ $\texttt{return} \not\simeq t$*

Bewijs. Voor $t = \texttt{edit [Just 1]}$ kunnen we de volgende afleiding maken:

$$\dfrac{\dfrac{}{\texttt{edit [Just 1]} \downarrow \texttt{Val 1 Unstable}}\ (\text{editJust})}{\texttt{edit [Just 1] >>= return} \downarrow \texttt{NoVal}}\ (\texttt{>>=}\text{Unst})$$

Dus $\texttt{edit [Just 1]} \downarrow \texttt{Val 1 Unstable}$ en $\texttt{edit [Just 1] >>= return} \downarrow \texttt{NoVal}$. Hiermee kunnen we concluderen dat `edit [Just 1]` niet equivalent is aan `edit [Just 1] >>= return`.

□

Opmerking. De bind bezit hiermee niet alle eigenschappen van een monad.

7.2 Concrete afleiding

Op de volgende bladzijde staat een bewijs van het resultaat van de executie van het bestelprogramma uit figuur 4.1. Het bewijs laat zien dat:

- `vraagBestellingen >>* (f,g)` $\Rightarrow t'$ voor een zekere t'
- $t' \downarrow$ `Val ["Biefstuk", "Zalm"] Stable`

Hiermee kunnen we concluderen dat Romeo en Julia krijgen wat ze besteld hebben: biefstuk en zalm. We zien dat de semantiek compact genoeg is om het bewijs op één pagina te laten passen.

Figuur 7.1: Bewijs van het bestelprogramma

$$A: \dfrac{\dfrac{}{\texttt{edit [Just "Biefstuk", Just "Biefstuk"] } \downarrow \texttt{ Val "Biefstuk" Unstable}}\ (\text{editJust}) \qquad \dfrac{}{\texttt{edit [Nothing, Just "Zalm"] } \downarrow \texttt{ NoVal}}\ (\text{editNothing})}{\texttt{vraagBestellingen } \downarrow \texttt{ Val [Val "Biefstuk" Unstable, NoVal] Unstable}}\ (\text{par})$$

$$B: \dfrac{\dfrac{}{\texttt{edit [Just "Biefstuk", Just "Biefstuk"] } \Rightarrow \texttt{ edit [Just "Biefstuk"]}}\ [\text{edit}] \qquad \dfrac{}{\texttt{edit [Nothing, Just "Zalm"] } \Rightarrow \texttt{ edit [Just "Zalm"]}}\ [\text{edit}]}{\texttt{vraagBestellingen } \Rightarrow \texttt{ parallel [edit [Just "Biefstuk"], edit [Just "Zalm"]]}}\ [\text{par}]$$

$$C: \dfrac{A \qquad \texttt{f (Val [Val "Biefstuk" Unstable, NoVal] Unstable)} \rightarrow_\lambda \texttt{ Nothing} \qquad B}{\texttt{vraagBestellingen >>* (f,g)} \Rightarrow \texttt{(parallel [edit [Just "Biefstuk"], edit [Just "Zalm"]]) >>* (f,g)}}\ [>>^*\text{Nothing}]$$

$$X: \dfrac{\dfrac{}{\texttt{edit [Just "Biefstuk"] } \downarrow \texttt{ Val "Biefstuk" Unstable}}\ (\text{editJust}) \qquad \dfrac{}{\texttt{edit [Just "Zalm"] } \downarrow \texttt{ Val "Zalm" Unstable}}\ (\text{editJust})}{\texttt{parallel [edit [Just "Biefstuk"], edit [Just "Zalm"]] } \downarrow \texttt{ Val [Val "Biefstuk" Unstable, Val "Zalm" Unstable] Unstable}}\ (\text{par})$$

$$Y: \dfrac{\dfrac{}{\texttt{return } b \Rightarrow \texttt{return } b}\ [\text{return}]}{\texttt{(return } b\texttt{) @ getVal} \Rightarrow \texttt{(return } b\texttt{) @ getVal}}\ [@] \qquad \begin{array}{c} b = \texttt{[Val "Biefstuk" Unstable, Val "Zalm" Unstable]} \\ b' = \texttt{["Biefstuk", "Zalm"]} \end{array}$$

$$\dfrac{X \qquad \texttt{f (Val } b \texttt{ Unstable)} \rightarrow_\lambda \texttt{ Just ((return } b\texttt{) @ getVal)} \qquad Y}{\texttt{(parallel [edit [Just "Biefstuk"], edit [Just "Zalm"]]) >>* (f,g)} \Rightarrow \texttt{(return } b\texttt{) @ getVal}}\ [>>^*\text{vJust}]$$

$$\dfrac{\dfrac{}{\texttt{return } b \downarrow \texttt{ Val } b \texttt{ Stable}}\ (\text{return}) \qquad \texttt{getVal } b \rightarrow_\lambda b'}{\texttt{(return } b\texttt{) @ getVal} \downarrow \texttt{ Val } b' \texttt{ Stable}}\ (@\text{Value})$$

Hoofdstuk 8

Gerelateerd werk

Een formele semantiek is een nauwkeurige specificatie van de betekenis van een programma of een taal. Volgens Nielson en Nielson[3] zijn er drie hoofdstromingen van semantiek:

- **Operationele semantiek** Deze vorm beschrijft hoe de executie van een programma in de taal verloopt.
- **Denotationele semantiek** Deze vorm koppelt elk taalconstruct aan een wiskundig object dat het resultaat van de executie van het taalconstruct modelleert.
- **Axiomatische semantiek** Deze vorm beschrijft aan welke logische axioma's een taalconstruct voldoet.

Binnen de operationele semantiek kunnen twee vormen onderscheiden worden: structurele operationele semantiek en natuurlijke semantiek. Structurele operationele semantiek beschrijft de werking van de individuele stappen van de executie van een programma. Natuurlijke semantiek beschrijft wat het resultaat is van de executie van een programma.

Ik heb gekozen voor een structurele operationele semantiek, omdat de implementatiesemantiek ook deze vorm heeft. Dit maakt het controleren van de correctheid van de deductieve semantiek ten opzichte van de implementatiesemantiek zoals gedaan in hoofdstuk 6 makkelijker en minder foutgevoelig.

Hoofdstuk 9

Conclusies

In dit onderzoek heb ik een deductieve semantiek voor iTasks', een dialect van iTasks, ontwikkeld met als doel om formeel te kunnen redeneren over iTasks. De onderzoeksvraag luidde: Met welke semantiek kunnen we makkelijker redeneren over iTasks en over programma's in iTasks?

Waar het vrijwel ondoenlijk zoniet onmogelijk was om met de implementatiesemantiek een formeel bewijs te maken, is dat met de deductieve semantiek gelukt voor het bestelprogramma (figuur 2.2), na een vertaling naar iTasks' (figuur 4.1), resulterend in het bewijs in figuur 7.1. Dit komt doordat de regels van de deductieve semantiek uit hoofdstuk 5 een uitspraak doen over slechts één aspect: waarde, exceptie of evolutie. De definities van implementatiesemantiek behandelen deze aspecten tegelijkertijd. Ook hebben we aannemelijk gemaakt dat de deductieve semantiek correct is ten opzichte van de implementatiesemantiek. We kunnen dus concluderen dat het is gelukt een semantiek te maken waarmee formeel geredeneerd kan worden over een deelverzameling van programma's in iTasks.

Hoofdstuk 10

Toekomstig werk

Er zijn nog vele mogelijkheden om de deductieve semantiek te verbeteren. Ik heb een informele bewijsschets gegeven van de correctheid van de deductieve semantiek ten opzichte van de al bestaande implementatiesemantiek. Er zou onderzocht kunnen worden of dit ook op een formele manier bewezen kan worden, eventueel op basis van de bewijsschets. We moeten dan wel een manier vinden om om te gaan met de brug tussen iTasks en iTasks'.

Ik heb in dit onderzoek niet gekeken hoe praktisch het voor een programmeur is bewijzen met deze semantiek kan maken. Dit zou verder onderzocht kunnen worden in een case study.

We hebben nu een semantiek die iTasks' beschrijft. Deze semantiek moet echter nog compleet worden gemaakt ten opzichte van de implementatiesemantiek. We zouden de elementen die in hoofdstuk 4 uit iTasks' zijn weggelaten, iteratief toe kunnen voegen. Hiervoor moet er een oplossing worden gevonden om onder andere gebruikersinvoer en existentieel getypeerde excepties te modelleren in de semantiek. Ook moeten Shared Data Sources worden toegevoegd. Deze constructen zijn in iTasks' volledig weggelaten, terwijl ze vaak worden gebruikt bij het programmeren. Dit moet echter zo gebeuren, dat de semantiek beknopt genoeg blijft om in de praktijk mee te kunnen werken. Ik verwacht dat de uitdaging in deze laatste eis zit.

De conventie dat de waarde van een taak niet verandert zodra deze stabiel is, is een belangrijke eigenschap van Task-Oriented Programming. Elke taak dient zich aan de conventie te houden en verwacht dat andere taken dit ook doen. Met de deductieve semantiek kan nu bewezen worden dat alle taken zich ook daadwerkelijk aan deze conventie houden. Ook kunnen we verschillende algebraïsche eigenschappen over iTasks' bewijzen, zoals symmetrie, transitiviteit en distributie. Voorbeelden van een paar eigenschappen die wellicht te bewijzen zijn, zijn:

- $(t_1 \texttt{ -||- } t_2) \texttt{ @ } f \simeq t_1 \texttt{ @ } f \texttt{ -||- } t_2 \texttt{ @ } f$
- $(\texttt{return } a) \texttt{ @ } f \simeq \texttt{ return } (f\ a)$

De deductieve semantiek zegt ons wat de waarde of de exceptie van een taak is tijdens de eerste iteratie en wat de taak na de eerste iteratie nog moet doen. We zouden hier drie uitspraken aan toe kunnen voegen:

- Een uitspraak die zegt welke waarde een taak heeft na i iteraties.
- Een uitspraak die zegt welke exceptie een taak gooit na i iteraties.
- Een uitspraak die zegt wat een taak nog moet doen na i iteraties.

Dit ligt dichter bij het concept van Task-Oriented Programming dat een taak een waarde heeft die door de tijd heen kan veranderen, maar verder van de implementatiesemantiek. Verder onderzoek moet uitwijzen of dit inderdaad handiger is.

We zouden kunnen bekijken of het mogelijk is om een axiomatische semantiek te maken. Taken zijn instructies die serieel samengesteld kunnen worden en die een gemeenschappelijke toestand bewerken. Een semantiek gebaseerd op Hoare-logica[2] zou daarom een goede kandidaat zijn. Verder onderzoek zou echter moeten uitwijzen of deze vorm ook daadwerkelijk geschikt is om bewijzen mee te maken.

Bibliografie

[1] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice-Hall International, March 1988.

[2] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[3] Hanne Riis Nielson and Flemming Nielson. Semantics with applications: a formal introduction. 1992.

[4] Rinus Plasmeijer, Peter Achten, and Pieter Koopman. itasks: executable specifications of interactive work flow systems for the web. *ACM SIGPLAN Notices*, 42(9):141–152, 2007.

[5] Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter Koopman. Task-oriented programming in a pure functional language. In *Proceedings of the 14th symposium on Principles and Practice of Declarative Programming*. ACM, 2012.

Bijlage A

Overzicht deductieregels

A.1 Return

$$\frac{}{\texttt{return}\ a \downarrow \texttt{Val}\ a\ \texttt{Stable}}\ (\text{return}) \qquad \frac{}{\texttt{return}\ a \Rightarrow \texttt{return}\ a}\ [\text{return}]$$

A.2 Edit

$$\frac{}{\texttt{edit}\ [\texttt{Just}\ a : xs] \downarrow \texttt{Val}\ a\ \texttt{Unstable}}\ (\text{editJust}) \qquad \frac{}{\texttt{edit}\ [\texttt{Nothing} : xs] \downarrow \texttt{NoVal}}\ (\text{editNothing})$$

$$\frac{}{\texttt{edit}\ [] \downarrow \texttt{NoVal}}\ (\text{edit}[]) \qquad \frac{}{\texttt{edit}\ [x : xs] \Rightarrow \texttt{edit}\ xs}\ [\text{edit}] \qquad \frac{}{\texttt{edit}\ [] \Rightarrow \texttt{edit}\ []}\ [\text{edit}[]]$$

A.3 Throw

$$\frac{}{\texttt{throw}\ e \lightning e}\ \{\text{throw}\}$$

A.4 At

$$\frac{t \downarrow \texttt{Val}\ a\ s \qquad f\ a \rightarrow_\lambda b}{t\ \texttt{@}\ f \downarrow \texttt{Val}\ b\ s}\ (\text{@Value}) \qquad \frac{t \downarrow \texttt{NoVal}}{t\ \texttt{@}\ f \downarrow \texttt{NoVal}}\ (\text{@NoVal})$$

$$\frac{t \Rightarrow t'}{t\ \texttt{@}\ f \Rightarrow t'\ \texttt{@}\ f}\ [@] \qquad \frac{t \lightning e}{t\ \texttt{@}\ f \lightning e}\ \{@\}$$

A.5 Bind

$$\frac{t \downarrow \texttt{Val}\ a\ \texttt{Unstable}}{t\ \texttt{>>=}\ f \downarrow \texttt{NoVal}}\ (\texttt{>>=}\text{Unst}) \qquad \frac{t \downarrow \texttt{NoVal}}{t\ \texttt{>>=}\ f \downarrow \texttt{NoVal}}\ (\texttt{>>=}\text{NoVal}) \qquad \frac{t_1 \downarrow \texttt{Val}\ a\ \texttt{Stable} \qquad f\ a \rightarrow_\lambda t_2 \qquad t_2 \downarrow v}{t_1\ \texttt{>>=}\ f \downarrow v}\ (\texttt{>>=}\text{Stable})$$

$$\frac{t \downarrow \texttt{Val}\ a\ \texttt{Unstable} \qquad t \Rightarrow t'}{t\ \texttt{>>=}\ f \Rightarrow t'\ \texttt{>>=}\ f}\ [\texttt{>>=}\text{Unst}] \qquad \frac{t \downarrow \texttt{NoVal} \qquad t \Rightarrow t'}{t\ \texttt{>>=}\ f \Rightarrow t'\ \texttt{>>=}\ f}\ [\texttt{>>=}\text{NoVal}] \qquad \frac{t_1 \downarrow \texttt{Val}\ a\ \texttt{Stable} \qquad f\ a \rightarrow_\lambda t_2 \qquad t_2 \Rightarrow t_3}{t_1\ \texttt{>>=}\ f \Rightarrow t_3}\ [\texttt{>>=}\text{Stable}]$$

$$\frac{t\ \lightning\ e}{t\ \texttt{>>=}\ f\ \lightning\ e}\ \{\texttt{>>=}\} \qquad \frac{t_1 \downarrow \texttt{Val}\ a\ \texttt{Stable} \qquad f\ a \rightarrow_\lambda t_2 \qquad t_2\ \lightning\ e}{t_1\ \texttt{>>=}\ f\ \lightning\ e}\ \{\texttt{>>=}\text{t}_2\}$$

A.6 Step

In de tabel op pagina 52 staan de regels van de step. De kolom geeft aan of de deeltaak een waarde heeft of een exceptie gooit. De rij geeft aan of er wordt overgestapt op de volgende taak(rij Just) of niet(rij Nothing).

	Waarde	Exceptie
Just	$\dfrac{t_1 \downarrow v \quad f\ v \to_\lambda \texttt{Just}\ t_2 \quad t_2 \downarrow v'}{t_1 \gg\!* (f,g) \downarrow v'}$ (>>*vJust)	$\dfrac{t_1 \lightning e \quad g\ e \to_\lambda \texttt{Just}\ t_2 \quad t_2 \downarrow v}{t_1 \gg\!* (f,g) \downarrow v}$ (>>*eJust)
	$\dfrac{t_1 \downarrow v \quad f\ v \to_\lambda \texttt{Just}\ t_2 \quad t_2 \Rightarrow t_3}{t_1 \gg\!* (f,g) \Rightarrow t_3}$ [>>*vJust]	$\dfrac{t_1 \lightning e \quad g\ e \to_\lambda \texttt{Just}\ t_2 \quad t_2 \Rightarrow t_3}{t_1 \gg\!* (f,g) \Rightarrow t_3}$ [>>*eJust]
	$\dfrac{t_1 \downarrow v \quad f\ v \to_\lambda \texttt{Just}\ t_2 \quad t_2 \lightning e}{t_1 \gg\!* (f,g) \lightning e}$ {>>*vJust}	$\dfrac{t_1 \lightning e \quad g\ e \to_\lambda \texttt{Just}\ t_2 \quad t_2 \lightning e'}{t_1 \gg\!* (f,g) \lightning e'}$ {>>*eJust}
Nothing	$\dfrac{t \downarrow v \quad f\ v \to_\lambda \texttt{Nothing}}{t \gg\!* (f,g) \downarrow \texttt{NoVal}}$ (>>*Nothing)	n.v.t.
	$\dfrac{t \downarrow v \quad f\ v \to_\lambda \texttt{Nothing} \quad t \Rightarrow t'}{t \gg\!* (f,g) \Rightarrow t' \gg\!* (f,g)}$ [>>*Nothing]	n.v.t.
	n.v.t.	$\dfrac{t \lightning e \quad g\ e \to_\lambda \texttt{Nothing}}{t \gg\!* (f,g) \lightning e}$ {>>*Nothing}

A.7 Or

$$\frac{t_1 \lightning e}{t_1 \texttt{ -||- } t_2 \lightning e}\ \{\text{e-||-}\} \qquad \frac{t_1 \downarrow v \quad t_2 \lightning e}{t_1 \texttt{ -||- } t_2 \lightning e}\ \{\text{-||-e}\} \qquad \frac{t_1 \downarrow \texttt{NoVal} \quad t_2 \downarrow v}{t_1 \texttt{ -||- } t_2 \downarrow v}\ (\text{N-||-}) \qquad \frac{t_1 \downarrow v \quad t_2 \downarrow \texttt{NoVal}}{t_1 \texttt{ -||- } t_2 \downarrow v}\ (\text{-||-N})$$

$$\frac{t_1 \downarrow \texttt{Val } a \texttt{ Stable} \quad t_2 \downarrow v}{t_1 \texttt{ -||- } t_2 \downarrow \texttt{Val } a \texttt{ Stable}}\ (\text{S-||-}) \qquad \frac{t_1 \downarrow \texttt{Val } a_1 \texttt{ Unstable} \quad t_2 \downarrow \texttt{Val } a_2 \texttt{ Stable}}{t_1 \texttt{ -||- } t_2 \downarrow \texttt{Val } a_2 \texttt{ Stable}}\ (\text{-||-*})$$

t_1 \ t_2	NoVal	Instabiel	Stabiel
Noval	Noval	rechts	rechts
Instabiel	links	nondet	rechts
Stabiel	links	links	links

$$\frac{t_1 \downarrow \texttt{Val } a \texttt{ Stable} \quad t_2 \downarrow v}{t_1 \texttt{ -||- } t_2 \Rightarrow \texttt{return } a}\ [\text{S-||-}] \qquad \frac{t_1 \downarrow \texttt{NoVal} \quad t_2 \downarrow \texttt{Val } a \texttt{ Stable}}{t_1 \texttt{ -||- } t_2 \Rightarrow \texttt{return } a}\ [\text{N-||-S}] \qquad \frac{t_1 \downarrow \texttt{Val } a_1 \texttt{ Unstable} \quad t_2 \downarrow \texttt{Val } a_2 \texttt{ Stable}}{t_1 \texttt{ -||- } t_2 \Rightarrow \texttt{return } a_2}\ [\text{U-||-S}]$$

$$\frac{t_1 \downarrow \texttt{Val } a_1 \texttt{ Unstable} \quad t_2 \downarrow \texttt{Val } a_2 \texttt{ Unstable} \quad t_1 \Rightarrow t_1' \quad t_2 \Rightarrow t_2'}{t_1 \texttt{ -||- } t_2 \Rightarrow t_1' \texttt{ -||- } t_2'}\ [\text{U-||-U}] \qquad \frac{t_1 \downarrow \texttt{NoVal} \quad t_2 \downarrow \texttt{Val } a_2 \texttt{ Unstable} \quad t_1 \Rightarrow t_1' \quad t_2 \Rightarrow t_2'}{t_1 \texttt{ -||- } t_2 \Rightarrow t_1' \texttt{ -||- } t_2'}\ [\text{N-||-U}]$$

$$\frac{t_1 \downarrow \texttt{Val } a_1 \texttt{ Unstable} \quad t_2 \downarrow \texttt{NoVal} \quad t_1 \Rightarrow t_1' \quad t_2 \Rightarrow t_2'}{t_1 \texttt{ -||- } t_2 \Rightarrow t_1' \texttt{ -||- } t_2'}\ [\text{U-||-N}] \qquad \frac{t_1 \downarrow \texttt{NoVal} \quad t_2 \downarrow \texttt{NoVal} \quad t_1 \Rightarrow t_1' \quad t_2 \Rightarrow t_2'}{t_1 \texttt{ -||- } t_2 \Rightarrow t_1' \texttt{ -||- } t_2'}\ [\text{N-||-N}]$$

$$\frac{t_1 \downarrow \texttt{Val}\ a_1\ \texttt{Unstable} \qquad t_2 \downarrow \texttt{Val}\ a_2\ \texttt{Unstable}}{t_1\ \texttt{-||-}\ t_2 \downarrow \texttt{Val}\ a_1\ \texttt{Unstable}}\ (\texttt{-||->})\text{als } t_1 \text{ het laatst bewerkt is}$$

$$\frac{t_1 \downarrow \texttt{Val}\ a_1\ \texttt{Unstable} \qquad t_2 \downarrow \texttt{Val}\ a_2\ \texttt{Unstable}}{t_1\ \texttt{-||-}\ t_2 \downarrow \texttt{Val}\ a_2\ \texttt{Unstable}}\ (\texttt{-||-<})\text{als } t_2 \text{ het laatst bewerkt is}$$

A.8 Parallel

$$\frac{t_1 \downarrow v_1 \quad \ldots \quad t_n \downarrow v_n}{\texttt{parallel}\ [t_1, ..., t_n] \downarrow \texttt{Val}\ [v_1, ..., v_n]\ parval(v_1, ..., v_n)}\ (\text{par})$$

$$parval(V) = \begin{cases} \texttt{Stable} & \text{als } \forall v \in V \exists a : v = \texttt{Val}\ a\ \texttt{Stable} \\ \texttt{Unstable} & \text{anders} \end{cases}$$

$$\frac{t_1 \Rightarrow t_1' \quad \ldots \quad t_n \Rightarrow t_n'}{\texttt{parallel}\ [t_1, ..., t_n] \Rightarrow \texttt{parallel}\ [t_1', ..., t_n']}\ [\text{par}]$$

$$\frac{t_1 \downarrow v_1 \quad \ldots \quad t_{n-1} \downarrow v_{n-1} \quad t_n \lightning e}{\texttt{parallel}\ [t_1, ..., t_n, ..., t_{n+m}] \lightning e}\ \{\text{par}\}$$

A.9 Big-step semantiek

$$\frac{}{t \mapsto_0 t}\ \text{bigstep1}$$

$$\frac{t_1 \mapsto_i t_2 \qquad t_2 \Rightarrow t_3}{t_1 \mapsto_{i+1} t_3}\ \text{bigstep2}$$