

Een gecombineerde calculus voor algebraïsche, scoped en parallelle effecten

Lander Debreyne

Thesis voorgedragen tot het behalen
van de graad van Master of Science
in de ingenieurswetenschappen:
computerwetenschappen, hoofdoptie
Artificiële intelligentie

Promotor:

Prof. dr. ir. T. Schrijvers

Begeleider:

Ir. B. van den Berg

© Copyright KU Leuven

Zonder voorafgaande schriftelijke toestemming van zowel de promotor als de auteur is overnemen, kopiëren, gebruiken of realiseren van deze uitgave of gedeelten ervan verboden. Voor aanvragen tot of informatie i.v.m. het overnemen en/of gebruik en/of realisatie van gedeelten uit deze publicatie, wend u tot het Departement Computerwetenschappen, Celestijnenlaan 200A bus 2402, B-3001 Heverlee, +32-16-327700 of via e-mail info@cs.kuleuven.be.

Voorafgaande schriftelijke toestemming van de promotor is eveneens vereist voor het aanwenden van de in deze masterproef beschreven (originele) methoden, producten, schakelingen en programma's voor industrieel of commercieel nut en voor de inzending van deze publicatie ter deelname aan wetenschappelijke prijzen of wedstrijden.

Voorwoord

...

Lander Debreyne

Inhoudsopgave

Voorwoord	i
Samenvatting	iv
Lijst van figuren en tabellen	v
1 Inleiding	1
1.1 Effecten	1
1.2 Monads	2
1.3 Effect handlers	2
1.4 Programmeertaal-onderzoek en industrie	2
1.5 Gecombineerde calculus	2
1.6 Doel	3
1.7 Overzicht	4
2 Achtergrond	5
2.1 Effecten en handlers	5
2.2 Algebraïsche operaties	5
2.3 Algebraïsche effect handlers	6
2.4 Beperkingen algebraïsche effecten en handlers	7
2.5 Effecten met scope	7
2.6 Parallele algebraïsche effecten	7
2.7 Calculi	8
3 Startpunt	11
3.1 λ_{sc} : A Calculus for Scoped Effects & Handlers	11
3.2 λ^p : Parallel Algebraic Effect Handlers	16
3.3 Gecombineerde calculus	20
4 Motivatie en Uitdagingen	21
4.1 Sequentiële vs parallele uitvoering	21
5 Syntaxis	23
5.1 Lijst-structuur	23
5.2 Let rec	25
5.3 Handler clauses	25
6 Operationele Semantiek	27
6.1 Letrec: Recursieve functies	27
6.2 For clause	27

6.3	Lijst- en hulp-functies	30
7	Type- en Effect-Systeem	33
7.1	Lijsten	33
8	Uitgewerkte Voorbeelden	35
8.1	Voorbeelden met Scoped Effecten	35
8.2	Voorbeelden met Scoped Effecten	35
8.3	Voorbeelden met Beide Effecten	35
9	Metatheorie	37
9.1	Lemma's	37
9.2	Behoud	37
9.3	Vooruitgang	37
10	Evaluatie	39
10.1	Bijdrage	39
10.2	Bruikbaarheid	39
10.3	Correctheid	39
10.4	Implementatie	39
10.5	Backwards compatibility	39
11	Gerelateerd Werk	41
12	Besluit	43
12.1	Resultaten	43
12.2	Beperkingen	43
12.3	Toekomstig werk	43
	Bibliografie	47

Samenvatting

...

Lijst van figuren en tabellen

Lijst van figuren

4.1 Scope van de verschillende calculi vermeld in deze thesis	21
---	----

Lijst van tabellen

2.1 Pure, simpel getypeerde λ -calculus syntaxis	8
2.2 Operationele semantiek van pure, simpel getypeerde λ -calculus	9
2.3 Typesysteem van pure, simpel getypeerde λ -calculus	9
3.1 Syntaxis voor λ_{sc}	12
3.2 Operationele semantiek van λ_{sc}	14
3.3 Waarde typering van λ_{sc}	16
3.4 Computatie typering van λ_{sc}	17
3.5 Handler typering van λ_{sc}	18
3.6 Syntaxis voor λ^p	18
3.7 Operationele semantiek van λ^p	19
5.1 λ_{sc}^p Syntaxis	24
6.1 Operationele semantiek van λ_{sc}^p	28
6.2 Operationele semantiek van λ_{sc}^p , lijst- en hulp-functies	29

Hoofdstuk 1

Inleiding

Programmeren omvat het schrijven van instructies die een computer uitvoert om een specifieke taak te volbrengen. Programmeurs schrijven programma's in een programmeertaal, een speciale taal die bestaat uit een set woorden, symbolen en regels. Een programmeertaal heeft als doel instructies te geven aan een computer. Calculi zijn nuttig om de eigenschappen en het gedrag van een programmeertaal formeel te bestuderen. Programmeertaal-calculi bieden een wiskundig kader dat formele analyses mogelijk maakt om de fundamentele eigenschappen van de taal te begrijpen. Met behulp van abstracte algebra en logische notatie worden de syntaxis, de operationele semantiek en het type-en-effectsysteem van een vereenvoudigde versie van de programmeertaal voorgesteld.

1.1 Effecten

Effecten zijn het resultaat van een bewerking die door een computerprogramma wordt uitgevoerd. Mogelijke effecten zijn het weergeven van informatie op het scherm, het opslaan van gegevens in een database of het uitvoeren van een berekening. In programma's zijn effecten noodzakelijk om het gewenste gedrag te bereiken. Anderzijds kunnen effecten onbedoelde en ongewenste gevolgen hebben. In pure functionele programmeertalen kunnen in principe geen effecten plaatsvinden buiten het uitvoeren van een berekening. Omdat de programmeur geen rekening moet houden met andere effecten kan code geschreven in pure functionele programmeertalen makkelijker te begrijpen en over te redeneren zijn. Deze code verhoogt de productiviteit en verlaagt de kans op fouten. Het puur of vrij van neveneffecten zijn van deze programma's kan limiterend zijn voor de expressiviteit en het bereiken van het gewenste gedrag. Daarom is het elegant introduceren en correct afhandelen van effecten een belangrijke open vraag in het programmeertaal-onderzoek, in het bijzonder voor functionele programmeertalen. Een goede functionele programmeertaal isoleert, controleert en beheert effecten op een voorspelbare manier. Voor de programmeur maakt een goede functionele programmeertaal het redeneren, uitbreiden, testen en onderhouden van programma's met effecten duidelijk en efficiënt. Expliciete constructies voor het redeneren over effecten zijn essentieel om dit doel te bereiken.

1.2 Monads

De meest industrie-relevante aanpak om effecten te modelleren in pure, functionele programmeertalen is de monad [5]. De bekendste voorbeelden zijn `Optional` in Java, `Result` in Rust, en de IO monad en de Maybe monad in Haskell. In Haskell zijn monad transformers [4] populair om modulaire compositie van monads, en bijgevolg effecten, te realiseren. Dit gebeurt door verschillende types monads te combineren tot een enkele, gecombineerde monad.

1.3 Effect handlers

Een tweede constructie om effecten te modelleren is het gebruik van algebraïsche effecten [9] en effect handlers. Het concept is om de aanroep van het effect, of de syntaxis, te scheiden van de afhandeling van het effect, of de semantiek. Een effect handler kan worden beschouwd als een functie die verantwoordelijk is voor de afhandeling van een effect in een andere functie zodat dit op een voorspelbare en modulaire manier kan gebeuren.

1.4 Programmeertaal-onderzoek en industrie

WebAssembly [3] is een goed voorbeeld van hoe principes en vooruitgang uit programmeertaal-onderzoek vertaald kunnen worden naar een industrie-relevante programmeertaal. WebAssembly is van het begin af aan ontworpen met een formele semantiek hetgeen bewijst dat dit een waardevolle aanpak kan zijn. De doelen die de auteurs vooropstellen voor een veilige, snelle, draagbare en compacte taal zijn toepasbaar op het ontwerp van bijna alle talen. Met deze principes in gedachte kan toegewerkt worden naar een calculus voor effect handlers die, mits verder ontwikkeling, kan evolueren naar een industrie-relevante aanpak om in een pure functionele taal met effecten om te gaan. Deze taal zou vervolgens het monopolie van monads in pure, functionele talen kunnen beëindigen en zo programmeurs een keuze geven.

1.5 Gecombineerde calculus

Algebraïsche effect handlers behandelen algebraïsche effecten. Algebraïsche effecten zijn effecten die generisch geschreven kunnen worden als $\mathbf{op}v(y.c)$ met v een parameter voor het effect en $(y.c)$ voor de continuatie of de rest van het programma na het effect.

Calculi van effect handlers zouden ideaal gezien drie eigenschappen moeten bezitten.

- Overloading van de effecten: De effecten krijgen een verschillende semantiek door het schrijven en gebruiken van verschillende handlers voor hetzelfde programma.
- Modulaire compositie: Het modulaair combineren van bewerkingen van verschillende effecten is mogelijk zonder strikte beperkingen.
- Effectinteracties: Effecten interageren met elkaar, waarbij de volgorde van de verschillende handlers een rol speelt en nieuwe semantische mogelijkheden biedt.

Het scheiden van operaties en hun behandeling, respectievelijk in de effecten en de handlers, zorgt voor overloading en modulaire compositie.

Algebraïsche effect handlers zijn in de huidige vorm niet klaar voor industrie aangezien deze handlers niet alle computationele effecten kunnen modelleren. Hierdoor is de expressiviteit van programmeren met algebraïsche effect handlers beperkt.

Het toevoegen van scoped effecten verhoogt de expressiviteit van een effect handler-gebaseerde programmeertaal. Scoped effect zijn effecten waarbij het gedrag in scope beperkt is tot een deel van de totale berekening, wat niet mogelijk is bij algebraïsche bewerkingen. Dit type effect verdeelt het programma in een computatie die binnen het bereik van het effect valt en een deel dat buiten het bereik valt. Generisch is dit te schrijven als $\mathbf{sc} \ v \ (y. \ c_1) \ (z. \ c_2)$ met v de parameter, $(y. \ c_1)$ de berekening in scope en $(z. \ c_2)$ de continuatie. De λ_{sc} calculus [2] beschrijft een calculus voor algebraïsche en scoped effecten. Het resultaat is een calculus die expressiviteit biedt die dichterbij de expressiviteit van monads komt. Om deze calculus industrie-relevant te noemen moet de performantie echter nog beter. Handlers in de λ_{sc} calculus handelen effecten immers sequentieel af.

Om de performantie te verbeteren kan methodiek toegevoegd worden om effecten in parallel te behandelen. Parallele algebraïsche effecten [11] bieden deze mogelijkheid. De λ^p calculus maakt parallelle afhandeling van effecten mogelijk en levert performantieverbeteringen voor parallelliseerbare algebraïsche effecten.

1.6 Doel

Het voorgaande beschrijft het doel van deze masterproef, met name een calculus die effect handlers modelleert met ondersteuning voor algebraïsche en scoped effecten en parallelle afhandeling van effecten. In de literatuur bestaat een leemte voor calculi die deze combinatie als **first class citizens** behandelt. Deze masterproef wil een ontwerp aanreiken voor een dergelijke calculus. Het resulterende ontwerp steunt op een sterke theoretische basis door een formele syntax, semantiek en type-en effectsysteem.

De beoogde calculus heeft minimaal volgende eigenschappen:

- Veilig voor het programmeren met effecten door het schrijven van overzichtelijke en duidelijke programma's mogelijk te maken door scheiding van syntax en semantiek van effecten door gebruik van de effect handler aanpak.
- Snel door parallelle afhandeling van effecten mogelijk te maken.
- Compact door een minimale calculus voor te stellen die aan deze eigenschappen voldoet.
- Draagbaar door een calculus voor te stellen die geen exotische constructies bevat en bijgevolg wijd implementeerbaar is.

De onderzoeksvraag voor deze masterproef luidt:

Hoe kan een compacte calculus worden gedefinieerd die sequentiële en parallelle afhandeling van algebraïsche en scoped effecten modelleert met behoud van handler overloading, modulaire compositie en effect interacties?

1.7 Overzicht

Hoofdstuk 2 geeft meer achtergrondinformatie over de onderwerpen die in deze masterproef worden behandeld. Vervolgens wordt in 3 de literatuur besproken waarop deze thesis voortbouwt. De motivatie voor dit onderwerp en de uitdagingen zijn het onderwerp van 4. De hoofdstukken die daarop volgen vormen de bijdrage van deze masterproef en stellen de λ_{sc}^p -calculus voor met de syntaxis in 5, de operationele semantiek in 6, het type-en effect-systeem in 7, voorbeelden in 8 en de metatheorie in 9. Vervolgens geeft hoofdstuk 10 een evaluatie van de masterproef, hoofdstuk 11 geeft een overzicht van gerelateerd werk. Tot slot bevat hoofdstuk 12 de conclusie.

Hoofdstuk 2

Achtergrond

2.1 Effecten en handlers

Effecten en handlers [1] zijn een mechanisme om effecten gestructureerd te behandelen in een programmeertaal. Ze stellen de programmeur in staat om de pure, functionele code in het programma te scheiden van de impure, effectvolle code. Dit gebeurt door effect-operaties te introduceren die effectvolle primitieven zijn. Vervolgens definieert de aanpak specifieke functies, effect handlers, voor het afhandelen van de effect-operaties. Effecten en handlers maken het makkelijker over de logica van de code te redeneren en effecten te coderen en te manipuleren op een modulaire componeerbare manier, waardoor meer modulaire en herbruikbare programma's kunnen worden gemaakt. Dit is bijzonder interessant wanneer complexe effecten moeten worden gecontroleerd, zoals in parallele of gedistribueerde systemen.

Handlers worden gebruikt om de instructies te beschrijven die moeten uitgevoerd worden wanneer een effect optreedt in de controlestroom van het programma. Doordat de semantiek van het effect in de handler zit, maakt dit de scheiding mogelijk tussen het voorkomen van het effect en de afhandeling, waardoor het gemakkelijker wordt te redeneren over de effecten in een programma en deze te beheren.

De belangrijkste eigenschap van programmeren met effect handlers is de hoge mate van modulariteit en diversiteit van semantiek die de programmeur kan bereiken met dezelfde bouwstenen. Dit vloeit voort uit overloading van effecten via verschillende handlers, modulaire compositie van effecten en scheiding van syntax en semantiek bij de afhandeling van effecten. Deze eigenschappen maken dit een veelbelovende aanpak voor het programmeren met effecten.

2.2 Algebraïsche operaties

Zoals vermeld in Sectie 2.1 zijn de operaties effectvolle primitieven. Deze primitieven hebben een type-signatuur $A \rightarrow B$, wat aanduidt dat het effect een term van type A omvormt naar type B. Het *choose* : $() \rightarrow Bool$ effect is een voorbeeld van een

algebraïsch effect dat een eenheids-waarde neemt als input en op willekeurige of niet-deterministische manier een boolean waarde teruggeeft. De syntaxis hiervoor in de λ_{sc}^p -calculus die deze masterproof uitwerkt is als volgt:

$$\mathbf{op\ choose\ } () (y. c) \quad (2.1)$$

Hier geeft het sleutelwoord **op** aan dat het om een algebraïsch effect gaat, gevolgd door het label *choose*, gevolgd door de input parameter $()$, gevolgd door de resumptie $(y. c)$. De resumptie is de computatie die de rest van het programma bevat gegeven de resulterende waarde van het effect. Een voorbeeld programma met het *choose* effect is een programma dat "pasta" of "pizza" teruggeeft afhankelijk van het resultaat:

$$c_{ND} = \mathbf{op\ choose\ } () (x. \text{then return "pasta" else return "pizza"}) \quad (2.2)$$

Algebraïsche effecten worden gekenmerkt doordat ze aan elkaar gerijgd kunnen worden door gebruik te maken van de algebraïciteits-eigenschap als volgt:

$$\begin{aligned} \mathbf{do\ } food \leftarrow \mathbf{op\ choose\ } () (x. \text{if } x \text{ then return "pasta" else return "pizza"}) ; \\ \mathbf{return\ } "We are eating " ++ food ++ " tonight!" \sim \\ \mathbf{op\ choose\ } () (x. \mathbf{do\ } food \leftarrow \text{if } x \text{ then return "pasta" else return "pizza"} ; \\ \mathbf{return\ } "We are eating " ++ food ++ " tonight!") \end{aligned} \quad (2.3)$$

Algemeen opgesteld, ziet deze eigenschap er zo uit:

$$\mathbf{do\ } x \leftarrow \mathbf{op\ } l\ v (y. c_1) ; c_2 \rightsquigarrow \mathbf{op\ } l\ v (y. \mathbf{do\ } x \leftarrow c_1 ; c_2) \quad (2.4)$$

Het effect van deze eigenschap is enerzijds dat de **do** clause wordt doorgeschoven naar de continuatie en anderzijds dat de computatie na de het **do** statement (de c_2 computatie) in de continuatie wordt geduwd.

2.3 Algebraïsche effect handlers

De effect handlers geven semantiek aan de algebraïsche operaties door te definiëren hoe de operaties te interpreteren. De handler h_{True} selecteert de *true* tak van de computatie voor het *choose* effect:

$$h_{True} = \mathbf{handler\ } \{ \mathbf{return\ } x \mapsto \mathbf{return\ } x, \mathbf{op\ choose\ } _ k \mapsto k\ true \} \quad (2.5)$$

Deze handler toegepast op het programma uit Eq. 2.3 zal als resultaat

$$"We are eating pasta tonight!" \quad (2.6)$$

hebben. De handler h_{Max} selecteert beide takken en combineert het resultaat:

$$h_{True} = \mathbf{handler\ } \{ \mathbf{return\ } x \mapsto \mathbf{return\ } [x], \mathbf{op\ choose\ } _ k \mapsto \mathbf{do\ } t \leftarrow k\ true ; \mathbf{do\ } f \leftarrow k\ false\ t ++ f \} \quad (2.7)$$

Deze handler toegepast op het programma uit Eq. 2.3 zal als resultaat

$$["We are eating pasta tonight!", "We are eating pizza tonight!"] \quad (2.8)$$

hebben.

2.4 Beperkingen algebraïsche effecten en handlers

Algebraïsche effecten en handlers zijn beperkt in hun vermogen om bepaalde soorten effecten te modelleren. Ze kunnen effecten modelleren zoals onder andere I/O, het veranderen van de staat van variabelen en niet-determinisme. Algebraïsche effecten kunnen echter geen effecten voorstellen die de controlestroom van een programma veranderen, zoals uitzonderingen (geen catch functionaliteit mogelijk). Deze beperking vloeit voort uit de aard van algebraïsche effecten, die gericht zijn op het modelleren van effecten die abstract kunnen worden voorgesteld als algebraïsche bewerkingen en compositioneel kunnen worden gecombineerd met andere effecten. Controlestroomveranderende effecten daarentegen vereisen meer verfijnde mechanismen voor hun behandeling, die buiten de mogelijkheden van algebraïsche effecten vallen.

2.5 Effecten met scope

Scoped effecten [2], [10], [12], [8] zijn effecten waarbij het gedrag in scope beperkt is tot een deel van de totale berekening. Dit type effect verdeelt het programma in een computatie die binnen het bereik van het effect valt en een deel dat buiten het bereik valt. De λ_{sc} calculus [2] beschrijft een calculus voor algebraïsche en scoped effecten. De syntaxis en semantiek van deze calculus wordt beschreven in Sectie 3.1. De syntaxis van scoped operaties ziet er als volgt uit:

$$\mathbf{sc} \ l \ v \ (y. c_1) \ (z. c_2) \quad (2.9)$$

Hier geeft het sleutelwoord **sc** aan dat het om een scoped effect gaat, gevolgd door een label l , een input parameter v , een berekening in scope $(y. c_1)$ en een resumptie $(z. c_2)$. Het verschil met de algebraïsche effecten is de toevoeging van een berekening in scope. Deze toevoeging laat de handler toe om de berekening in scope anders te interpreteren dan de resumptie. Door deze toevoeging kan ook geen generieke forwarding van effecten door verschillende handlers meer gebeuren en moet de handler een expliciete forwarding clause hebben van de volgende vorm:

$$\mathbf{fwd} \ f \ p \ k \mapsto c_f \quad (2.10)$$

Deze clause wordt meer in detail besproken in Sectie 3.1 en [2].

2.6 Parallele algebraïsche effecten

Effect handlers behandelen effecten standaard sequentieel. De λ^p calculus [11] maakt parallelle afhandeling van algebraïsche effecten mogelijk. Deze calculus wordt

termen t	$::=$	x	variabele
		$ \quad \lambda x : T. t$	abstractie
		$ \quad t t$	applicatie
waarden v	$::=$	$\lambda x : T. t$	abstractie waarde
types T	$::=$	$T \rightarrow T$	type van functies
contexten Γ	$::=$	\emptyset	Lege context
		$ \quad \Gamma, x : T$	term variabele binding

TABEL 2.1: Pure, simpel getypeerde λ -calculus syntaxis

behandeld in Sectie 3.2. De λ^p -calculus maakt gebruik van een constructie

$$\mathbf{for} \ x : n. e \tag{2.11}$$

Waarbij **for** het sleutelwoord is, en de expressie e in parallel wordt uitgevoerd voor elke x in n . Gelijkaardig aan de expliciete forwarding clause voor scoped effecten zoals besproken in Sectie 2.5, heeft deze calculus een **traverse** clause in de vorm

$$\mathbf{traverse} \ n \ l \ k \tag{2.12}$$

Deze **traverse** clause is handler specifiek en verandert de resumptie.

2.7 Calculi

Om een ontwerp voor een programmeertaal met bepaalde eigenschappen of mogelijkheden formeel voor te stellen, kan gebruikt worden gemaakt van een wiskundig kader zoals een calculus. Een calculus stelt een vereenvoudigde versie van een taal voor in abstracte wiskundige notatie en is een elegante manier om programmeertalen te modelleren omdat ook metatheoretische bewijzen van gewenste eigenschappen geleverd kunnen worden. Deze sectie zal een overzicht geven van een simpele calculus, de pure simpel getypeerde lambda-calculus met als doel de verschillende onderdelen van een calculus uiteen te zetten. De calculus is een licht aangepaste versie van de calculus in Hoofdstuk 9 van TAPL [7].

2.7.1 Syntaxis

De syntaxis geeft een abstract overzicht van de verschillende bouwstenen waaruit een correct programma kan bestaan. Tabel 2.1 geeft de syntaxis voor de pure, simpel getypeerde λ -calculus. De syntaxis maakt een onderscheid tussen termen, waarden, types en contexten en definieert voor elk van deze welke vormen ze kunnen aannemen.

$c \rightsquigarrow c'$	Computatie reductie
$\frac{c_1 \rightsquigarrow c'_1}{c_1 c_2 \rightsquigarrow c'_1 c_2}$	E-App1
$\frac{c_2 \rightsquigarrow c'_2}{v_1 c_2 \rightsquigarrow v_1 c'_2}$	E-App2
$\frac{}{(\lambda x : T_{11}. t_{12}) v_2 \rightsquigarrow [x \mapsto v_2] t_{12}}$	E-AppAbs

TABEL 2.2: Operationele semantiek van pure, simpel getypeerde λ -calculus

$\Gamma \vdash t : T$	typering
$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$	T-Var
$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1. t_2 : T_1 \rightarrow T_2}$	T-Abs
$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}}$	T-App

TABEL 2.3: Typesysteem van pure, simpel getypeerde λ -calculus

2.7.2 Semantiek

Tabel 2.2 toont de operationele semantiek voor de simpel, getypeerde λ -calculus. Dit zijn semantische regels die gevolgd kunnen worden om berekeningen te reduceren tot normaalvormen. De relatie $c \rightsquigarrow c'$ betekent dat de computatie kan reduceren.

2.7.3 Type-systeem

Tabel 2.3 toont het typesysteem voor de pure, simpel getypeerde λ -calculus. Het typesysteem bevat typeregels voor de verschillende mogelijke termen. Dit typesysteem bevat regels voor variabelen (**T-Var**), abstractie (**T-Abs**) en applicatie (**T-App**).

2.7.4 Metatheorie

Voor deze calculus kan een bewijs van vooruitgang gemaakt worden wat inhoudt dat bewezen kan worden dat elke gesloten, getypeerde term t ofwel een waarde is, of er

2. ACHTERGROND

een term t' bestaat zodat $t \rightsquigarrow t'$ en voor t' hetzelfde geldt.

Voor deze calculus kan eveneens een bewijs van preservatie geleverd worden wat inhoudt dat voor als voor termen t en t' geldt: $\Gamma \vdash t : T$ en $t \rightsquigarrow t'$ dan geldt $\Gamma \vdash t' : T$. Dit betekent dat als een term goed getypeerd is, elke geldige reductie van de term ook goed getypeerd is.

Hoofdstuk 3

Startpunt

Deze masterproef heeft als doel een calculus voor te stellen die algebraïsche en scoped effecten en effect handlers als **first-class citizens** beschouwt met sequentiële en parallelle behandeling voor deze effecten. Het startpunt voor deze calculus is $\lambda_{sc}[2]$, een calculus die sequentiële behandeling van scoped en algebraïsche effecten ondersteunt. Het startpunt voor de parallelle behandeling van effecten is $\lambda^p[11]$, een calculus die parallelle algebraïsche effecten ondersteunt.

3.1 λ_{sc} : A Calculus for Scoped Effects & Handlers

3.1.1 Syntaxis

Tabel 3.1 beeldt de syntaxis voor de λ_{sc} -calculus af.

De syntaxis onderscheidt termen en types. De termen zijn gesplitst in waarden v en computaties c waarbij waarden verschillen van computaties in dat enkel op de computaties reducties toegepast kunnen worden.

Waarden

Een waarde is ofwel de eenheids-waarde $()$, een paar van twee waarden (v_1, v_2) , een variabele x , een functie $\lambda x. c$ of een handler h . Een handler bestaat uit een return clause, geen of meerdere effect clauses en een forwarding clause. De return clause **return** $x \mapsto c_r$ duidt erop dat het resultaat x door de handler wordt behandeld door de computatie c_r . Een algebraïsche effect clause **op** $l x k \mapsto c$ wordt gekenmerkt door het sleutelwoord **op** gevolgd door een label l , een input parameter x en een resumptie $k. \mapsto c$ duidt aan dat een handler deze clause verwerkt door de computatie c toe te passen. Een scoped effect clause **sc** $l x p k \mapsto c$ is gelijkaardig aan een algebraïsche effect clause met een verschillend sleutelwoord **sc** en bijkomende een bewerking in scope p . De forwarding clause **fwd** $f p k \mapsto c_f$, die elke handler moet implementeren, zorgt voor expliciete forwarding wanneer de te behandelen scoped effect clause niet geïmplementeerd wordt door de handler. De computatie c_f specificeert hoe de onbekende scoped effect clause moet aangepast worden en kan

waarden v	$::=$	$() \mid (v_1, v_2) \mid x \mid \lambda x. c \mid h$	
handlers h	$::=$	handler { return $x \mapsto c_r$, $opr s$, fwd $f p k \mapsto c_f$ }	return clause effect clauses forwarding clause
effect clauses $opr s$	$::=$	\cdot op $l x k \mapsto c, opr s$ sc $l x p k \mapsto c, opr s$	algebraïsche effect clauses scoped effect clauses
computaties c	$::=$	return v op $l v (y. c)$ sc $l v (y. c_1) (z. c_2)$ $v \star c$ do $x \leftarrow c_1 ; c_2$ $v_1 v_2$ let $x = v$ in c	return waarde algebraïsch effect scoped effect behandeling do clause applicatie let
waarde types A, B, M	$::=$	$() \mid (A, B) \mid A \rightarrow \underline{C} \mid \underline{C} \Rightarrow \underline{D}$ α $\lambda \alpha. A$ $M A$	type variabele type operator abstractie type applicatie
type schemas σ	$::=$	$A \mid \forall \mu. \sigma \mid \forall \alpha. \sigma$	
computatie types $\underline{C}, \underline{D}$	$::=$	$A! \langle E \rangle$	
effect type rows E, F	$::=$	$\cdot \mid \mu \mid l ; E$	
kinds K	$::=$	$* \mid K \rightarrow K$	
signatuur contexten Σ	$::=$	$\cdot \mid \Sigma, l : A \rightarrow B$	
type contexten Γ	$::=$	$\cdot \mid \Gamma, x : A \mid \Gamma, \mu \mid \Gamma, \alpha$	

TABEL 3.1: Syntaxis voor λ_{sc}

hiervoor gebruik maken van f , dat het onbekende scoped effect bevat, de computatie in scope p en de resumptie k .

Computaties

Computaties kunnen een waarde uitkomen door middel van de return clause **return** v . Een algebraïsch effect **op** l v (y . c) bestaat uit het sleutelwoord **op**, een label l , een input parameter waarde v en een resumptie in de vorm (y . c). Een scoped effect **sc** l v (y . c_1) (z . c_2) bestaat uit het sleutelwoord **sc**, een label l , een input parameter waarde v , een scoped berekening (y . c_1) en een resumptie (z . c_2). In beide soorten effecten is de resumptie de computatie die gegeven de resulterende waarde van het effect, de rest van het programma bevat. De behandeling $v \star c$ drukt de behandeling van een computatie c door een waarde v , die een handler is, door middel van de \star -operator. Computaties worden aan elkaar gerijgd doormiddel van do clauses in de vorm **do** $x \leftarrow c_1$; c_2 waarbij het resultaat van c_1 in c_2 beschikbaar is als x . De syntax van applicatie in de calculus is door v_1 v_2 . Let-polymorfisme is beschikbaar in de calculus als **let** $x = v$ **in** c .

Types

Zoals de termen zijn ook de types opgesplitst in waarde types A , B , M en computatie types \underline{C} , \underline{D} . Waarde types zijn ofwel een eenheids-type $()$, een paar-type (A, B) , een functie type $A \rightarrow \underline{C}$, een handler type $\underline{C} \Rightarrow \underline{D}$, een type variabele α , type operator abstractie $\lambda \alpha. A$ of type applicatie $M A$. Een computatie type $A! \langle E \rangle$ bestaat uit een waarde type A , het type van de resulterende waarde van de computatie, en een effect type E , het effect type van de effecten die tijdens de computatie kunnen opgeroepen worden. Een Effect type bestaat uit een mogelijke lege collectie atomische labels l , mogelijk met een row-variabele μ als laatste element. Deze calculus ondersteunt verder type schemas σ en kinds K . De calculus houdt een gespecialiseerde signatuur context Σ bij om via hun labels de signatuur van gebruikte effecten te typeren. De calculus heeft ook een type context die de variabelen bijhoudt.

3.1.2 Operationele semantiek

Tabel 3.2 toont de kleine-staps operationele semantiek van de λ_{sc} -calculus. De relatie $c \rightsquigarrow c'$ duidt aan dat c naar c' stapt of reduceert.

De regels **E-AppAbs** en **E-Let** behandelen functie applicatie en let-binding. De andere regels zijn op te delen in twee domeinen: computaties aan elkaar rijgen en computaties behandelen met handlers.

Computaties aan elkaar rijgen

Om computaties aan elkaar te rijgen van de vorm **do** $x \leftarrow c_1$; c_2 is een onderscheid te maken tussen het geval waarbij c_1 een stap kan zetten naar c'_1 (**E-Do**) of waarbij c_1 in normale vorm is (**return**, **op** ..., of **sc** ...). In het laatste geval, hangt

$c \rightsquigarrow c'$	Reductie van computaties	
$\frac{}{(\lambda x. c) v \rightsquigarrow c[v/x]}$	E-AppAbs	$\frac{}{\mathbf{let} x = v \mathbf{in} c \rightsquigarrow c[v/x]}$ E-Let
$\frac{c_1 \rightsquigarrow c'_1}{\mathbf{do} x \leftarrow c_1 ; c_2 \rightsquigarrow \mathbf{do} x \rightsquigarrow c'_1 ; c_2}$	E-Do	$\frac{}{\mathbf{do} x \leftarrow \mathbf{return} v ; c_2 \rightsquigarrow c_2[v/x]}$ E-DoRet
$\frac{}{\mathbf{do} x \leftarrow \mathbf{op} l v (y. c_1) ; c_2 \rightsquigarrow \mathbf{op} l v (y. \mathbf{do} x \leftarrow c_1 ; c_2)}$	E-DoOp	
$\frac{}{\mathbf{do} x \leftarrow \mathbf{sc} l v (y. c_1) (z. c_2) ; c_3 \rightsquigarrow \mathbf{sc} l v (y. c_1) (z. \mathbf{do} x \leftarrow c_2 ; c_3)}$	E-DoSc	
$\frac{c \rightsquigarrow c'}{h \star c \rightsquigarrow h \star c'}$	E-Hand	$\frac{(\mathbf{return} x \mapsto c_r) \in h}{h \star \mathbf{return} v \rightsquigarrow c_r[v/x]}$ E-HandRet
$\frac{(\mathbf{op} l x k \mapsto c) \in h}{h \star \mathbf{op} l v (y. c_1) \rightsquigarrow c[v/x, (\lambda y. h \star c_1) / k]}$	E-HandOp	
$\frac{(\mathbf{op} l _ _) \notin h}{h \star \mathbf{op} l v (y. c_1) \rightsquigarrow \mathbf{op} l v (y. h \star c_1)}$	E-FwdOp	
$\frac{(\mathbf{sc} l x p k \mapsto c) \in h}{h \star \mathbf{sc} l v (y. c_1) (z. c_2) \rightsquigarrow c[v/x, (\lambda y. h \star c_1) / p, (\lambda z. h \star c_2) / k]}$	E-HandSc	
$\frac{(\mathbf{sc} l _ _) \notin h \quad (\mathbf{fwd} f p k \mapsto c_f) \in h \quad g = \lambda(p', k'). \mathbf{sc} l v (y. p' y) (z. k' z)}{h \star \mathbf{sc} l v (y. c_1) (z. c_2) \rightsquigarrow c_f[(\lambda y. h \star c_1) / p, (\lambda z. h \star c_2) / k, g / f]}$	E-FwdSc	

TABEL 3.2: Operationele semantiek van λ_{sc}

de regel af van de normale vorm die zich presenteert in c_1 . In het geval dat de computatie een resultaat is met een waarde, $\mathbf{do} x \leftarrow \mathbf{return} v ; c_2$, wordt x vervangen door v in c_2 of $\rightsquigarrow c_2[v/x]$ (**E-DoRet**). Is de computatie een algebraïsche effect, $\mathbf{do} x \leftarrow \mathbf{op} l v (y. c_1) ; c_2$, dan kan de computatie herschreven worden met behulp van de algebraïciteits-eigenschap $\rightsquigarrow \mathbf{op} l v (y. \mathbf{do} x \leftarrow c_1 ; c_2)$ (**E-DoOp**). Bij een scoped effect, $\mathbf{do} x \leftarrow \mathbf{sc} l v (y. c_1) (z. c_2) ; c_3$, wordt een generalisatie van de algebraïciteits-eigenschap gebruikt $\rightsquigarrow \mathbf{sc} l v (y. c_1) (z. \mathbf{do} x \leftarrow c_2 ; c_3)$ (**E-DoSc**).

Computaties behandelen met handlers

Om computaties te behandelen met handlers, $h \star c$, is er opnieuw onderscheid te maken tussen het geval waar c kan reduceren $c \rightsquigarrow c'$ en waar c in normale vorm is ($\mathbf{return} v$, $\mathbf{op} \dots$, $\mathbf{sc} \dots$). In het eerste geval wordt de mogelijke stap genomen,

$h \star c \rightsquigarrow h \star c'$ (**E-Hand**). Geeft c een waarde terug, $h \star \mathbf{return} v$, dan wordt de return clause van de handler opgeroepen, $\rightsquigarrow c_r [v / x]$ (**E-HandRet**). Komt een handler een algebraïsch effect tegen, $h \star \mathbf{opl} v(y.c_1)$, dan wordt onderscheid gemaakt tussen het geval waar de handler een clause heeft voor de algebraïsche operatie of niet. Is dat zo, $(\mathbf{opl} x k \mapsto c) \in h$, dan is de volgende stap geldig $\rightsquigarrow c[v/x, (\lambda y. h \star c_1)/k]$ (**E-HandOp**). Is dat niet zo, $(\mathbf{opl} _ _) \notin h$, dan wordt de volgende stap gemaakt $\rightsquigarrow \mathbf{opl} v(y. h \star c_1)$ (**E-FwdOp**). Deze laatste stap is een generische forwarding voor algebraïsche effecten. Is de te behandelen computatie een scoped effect, $h \star \mathbf{sc} l v(y.c_1)(z.c_2)$, dan wordt hetzelfde onderscheid gemaakt. Geldt $(\mathbf{sc} l x p k \mapsto c) \in h$ dan is het resultaat $\rightsquigarrow c[v/x, (\lambda y. h \star c_1)/p, (\lambda z. h \star c_2)/k]$ (**E-HandSc**). Anders, wanneer $(\mathbf{sc} l _ _) \notin h$ $(\mathbf{fwd} f p k \mapsto c_f) \in h$ $g = \lambda(p', k'). \mathbf{sc} l v(y.p'y)(z.k'z)$ geldt, is het resultaat $\rightsquigarrow c_f[(\lambda y. h \star c_1)/p, (\lambda z. h \star c_2)/k, g/f]$ (**E-FwdSc**). Dit is een expliciete forwarding regel die nodig is om de correcte werking van scoped effecten te bekomen.

3.1.3 Type- en effect-systeem

De λ_{sc} -calculus heeft een typesysteem voor de termen en een type-systeem voor de effecten. Het effect-systeem maakt gebruik van effect rows. De volgende subsectie behandelen dit systeem, met in Tabel 3.3 de waarde typering, in Tabel 3.4 de computatie typering en in Tabel 3.5 de handler typering.

Waarde typering

Tabel 3.3 toont de waarde typeringsregels voor λ_{sc} . **T-Var**, **T-Unit** en **T-Pair** behandelen de typering van respectievelijk variabelen, de eenheidswaarde en paren. De volledige type equivalentie is hier achterwege gelaten maar te vinden in de appendix van de originele paper [2]. **T-Inst** en **T-InstEff** behandelen de instantie van respectievelijk type- en row-variabelen. **T-Gen** en **T-GenEff** behandelen de generalisatie van respectievelijk type- en row-variabelen.

Computatie typering

Tabel 3.4 toont de computatie typeringsregels voor λ_{sc} . Dezelfde opmerking rond type equivalentie als gemaakt voor typering van waarden geldt hier.

Handler typering

Tabel 3.5 toont de typeringsregels voor handlers in λ_{sc} . De typering bevat een regel voor elk type clause dat in een handler kan voorkomen, inclusief de lege handler, namelijk **T-Empty** voor de lege handler, **T-OprOp** voor de algebraïsche effecten, **T-OprSc** voor de scoped effecten, **T-Fwd** voor de forwarding clauses en **T-Handler** voor de handler als verzameling van deze clauses.

$\boxed{\Gamma \vdash v : \sigma}$	Waarde typing
$\frac{(x : \sigma) \in \Gamma}{\Gamma \vdash x : \sigma} \text{ T-Var}$	$\frac{}{\Gamma \vdash () : ()} \text{ T-Unit}$
$\frac{\Gamma \vdash v_1 : A \quad \Gamma \vdash v_2 : B}{\Gamma \vdash (v_1, v_2) : (A, B)} \text{ T-Pair}$	
$\frac{\Gamma, x : A \vdash c : \underline{C}}{\Gamma \vdash \lambda x. c : A \rightarrow \underline{C}} \text{ T-Abs}$	$\frac{\Gamma \vdash v : A \quad A \equiv B}{\Gamma \vdash v : B} \text{ T-Eqv}$
$\frac{\Gamma \vdash A : * \quad \Gamma \vdash v : \forall \alpha. \sigma}{\Gamma \vdash v : \sigma [A / \alpha]} \text{ T-Inst}$	$\frac{\Gamma, \alpha \vdash v : \sigma \quad \alpha \notin \Gamma}{\Gamma \vdash v : \forall \mu. \sigma} \text{ T-Gen}$
$\frac{\Gamma \vdash v : \forall \mu. \sigma}{\Gamma \vdash b : \sigma [E / \mu]} \text{ T-InstEff}$	$\frac{\Gamma, \mu \vdash v : \sigma \quad \mu \notin \Gamma}{\Gamma \vdash v : \forall \mu. \sigma} \text{ T-GenEff}$

TABEL 3.3: Waarde typing van λ_{sc}

3.1.4 Metatheorie

De λ_{sc} -calculus heeft een type veilig type-en-effect-systeem.

3.1.5 Veranderingen tegenover een calculus voor algebraïsche effecten

De grootste veranderingen van de λ_{sc} -calculus tegenover een calculus voor algebraïsche effecten zijn enerzijds de introductie van de **sc** clause voor scoped effecten en de syntaxis en semantiek hierrond, inclusief de nood voor niet-generieke forwarding doormiddel van de **fwd** clause. Anderzijds is er de introductie van let-polymorfisme door het **let** sleutelwoord.

3.2 λ^p : Parallel Algebraic Effect Handlers

De λ^p -calculus voor de parallele behandeling van algebraïsche effecten zoals beschreven in "Parallel Algebraic Effect Handlers"[11] modelleert een grote-staps operationele semantiek.

3.2.1 Syntaxis

Tabel 3.6 geeft de syntaxis voor de λ^p -calculus weer. De waarden zijn ofwel een literal i , een variabele x , een lambda-functie $\lambda x. e$, een lijst van waarden $\langle v_0, \dots, v_n \rangle$ of **perform** op die een algebraïsche effect uitvoert. Voor waarden wordt f gebruikt

$\boxed{\Gamma \vdash c : \underline{C}}$	Computatie typering
$\frac{\Gamma \vdash v_1 : A \rightarrow \underline{C} \quad \Gamma \vdash v_2 : A}{\Gamma \vdash v_1 v_2 : \underline{C}} \text{ T-App}$	
$\frac{\Gamma \vdash c_1 : A! \langle E \rangle \quad \Gamma, x : A \vdash c_2 : B! \langle E \rangle}{\Gamma \vdash \mathbf{do} \ x \leftarrow c_1; c_2 : B! \langle E \rangle} \text{ T-Do}$	
$\frac{\Gamma \vdash c : \underline{C} \quad \underline{C} \equiv \underline{D}}{\Gamma \vdash c : \underline{D}} \text{ T-EqC}$	
$\frac{\Gamma \vdash v : \sigma \quad \Gamma, x : \sigma \vdash c : \underline{C}}{\Gamma \vdash \mathbf{let} \ x = v \ \mathbf{in} \ c : \underline{C}} \text{ T-Let}$	
$\frac{\Gamma \vdash v : A}{\Gamma \vdash \mathbf{return} \ v : A! \langle E \rangle} \text{ T-Ret}$	
$\frac{\Gamma \vdash v : \forall \alpha. \alpha! \langle E \rangle \Rightarrow M \ \alpha! \langle F \rangle \quad \Gamma \vdash c : A! \langle E \rangle}{\Gamma \vdash v \star c : M \ A! \langle F \rangle} \text{ T-Hand}$	
$\frac{(l : A_l \rightarrow B_l) \in \Sigma \quad \Gamma \vdash v : A_l \quad \Gamma, y : B_l \vdash c : A! \langle l; E \rangle}{\Gamma \vdash \mathbf{op} \ l \ v \ (y. c) : A! \langle l; E \rangle} \text{ T-Op}$	
$\frac{(l : A_l \rightarrow B_l) \in \Sigma \quad \Gamma \vdash v : A_l \quad \Gamma, y : B_l \vdash c_1 : B! \langle l; E \rangle \quad \Gamma, z : B \vdash c_2 : A! \langle l; E \rangle}{\Gamma \vdash \mathbf{sc} \ l \ v \ (y. c_1) \ (z. c_2) : A! \langle l; E \rangle} \text{ T-Sc}$	

TABEL 3.4: Computatie typering van λ_{sc}

voor lambda-functies, n voor literals en s voor state. Handlers hebben steeds 3 clausules, namelijk een return clausule **return** $\mapsto f_r$, een algebraïsche effect clausule **op** $\mapsto f_p$ en een traverse clause **traverse** $\mapsto f_t$. De **traverse** clausule is essentieel in de parallele behandeling. Computaties zijn ofwel een waarde v , een applicatie $e \ e$, een for constructie **for** $x \ n. e$ om een lijst te maken van lengte n of een behandel frame **handle** $h \ s \ e$.

Een opvallende verschil tussen deze syntaxis en de syntaxis voor de λ_{sc} -calculus in Tabel 3.1 is dat in tegenstelling tot λ_{sc} in de λ^p -calculus waarden computaties zijn, er is dus geen strikt onderscheid. Verder kan een handler slechts 1 effect behandelen en hebben effecten geen label.

$\boxed{\Gamma \vdash \text{opr}s : \underline{C}}$	$\boxed{\Gamma \vdash \mathbf{fwd} \ f \ p \ k \mapsto c : \underline{C}}$	$\boxed{\Gamma \vdash h : \underline{C} \Rightarrow \underline{D}}$	Handler typing
$\frac{}{\Gamma \vdash . : \underline{C}} \text{ T-Empty}$			
$\frac{\Gamma \vdash \text{opr}s : \underline{C} \quad (l : A_l \rightarrow B_l) \in \Sigma \quad \Gamma, x : A_l, k : B_l \rightarrow \underline{C} \vdash c : \underline{C}}{\Gamma \vdash \mathbf{op} \ l \ x \ k \mapsto c, \text{opr}s : \underline{C}} \text{ T-OprOp}$			
$\frac{\Gamma \vdash \text{opr}s : M \ A! \langle E \rangle \quad (l : A_l \rightarrow b_l) \in \Sigma \quad \Gamma, \beta, x : A_l, p : B_l \rightarrow M \ \beta! \langle E \rangle, k : \beta \rightarrow M \ A! \langle E \rangle \vdash c : M \ A! \langle E \rangle}{\Gamma \vdash \mathbf{sc} \ l \ v \ p \ k \mapsto c, \text{opr}s : M \ A! \langle E \rangle} \text{ T-OprSc}$			
$\frac{A_p = \alpha \rightarrow M \ \beta! \langle E \rangle \quad A_k = \beta \rightarrow M \ A! \langle E \rangle \quad A'_k = M \ \beta \rightarrow M \ A! \langle E \rangle \quad \Gamma, \alpha, \beta, p : A_p, k : A_k, f : (A_p, A'_k) \rightarrow M \ A! \langle E \rangle \vdash c_f : M \ A! \langle E \rangle}{\Gamma \vdash \mathbf{fwd} \ f \ p \ k \mapsto c_f : M \ A! \langle E \rangle} \text{ T-Fwd}$			
$\frac{\langle E \rangle = \langle \text{labels}(\text{opr}s); F \rangle \quad \Gamma, \alpha \vdash \mathbf{return} \ x \mapsto c_r : M \ \alpha! \langle F \rangle \quad \Gamma, \alpha \vdash \text{opr}s : M \ \alpha! \langle F \rangle \quad \Gamma, \alpha \vdash \mathbf{fwd} \ f \ p \ k \mapsto c_f : M \ \alpha! \langle F \rangle}{\Gamma \vdash \mathbf{handler} \ \{ \mathbf{return}, \text{opr}s, \mathbf{fwd} \} : \forall \alpha. \alpha! \langle E \rangle \Rightarrow M \ \alpha! \langle F \rangle} \text{ T-Handler}$			

TABEL 3.5: Handler typing van λ_{sc}

waarden v, f, n, s	$::=$	$i \mid x \mid \lambda x. e$ $\mid \langle v_0, \dots, v_n \rangle \mid \mathbf{perform} \ op$	
handlers h	$::=$	$\{ \mathbf{return} \mapsto f_r$ $, \mathbf{op} \mapsto f_p$ $, \mathbf{traverse} \mapsto f_t \}$	return clause algebraïsche effect clause traverse clause
computaties e	$::=$	v $\mid e \ e$ $\mid \mathbf{for} \ x : n. e$ $\mid \mathbf{handle} \ h \ s \ e$	waarde applicatie for constructie behandel frame
evaluatie context F	$::=$	$. \mid F \ e \mid v \ F$	parallele context
E	$::=$	$. \mid E \ e \mid v \ E \mid \mathbf{handle} \ h \ s \ e$	sequentiële context

TABEL 3.6: Syntaxis voor λ^p

$c \rightsquigarrow c'$	Reductie van computaties
$\frac{}{(\lambda x. e) v \rightsquigarrow e[v/x]} \text{ E-App}$	
$\frac{}{\langle v_0, \dots, v_n \rangle i \rightsquigarrow v_i} \text{ E-Index}$	
$\frac{(\mathbf{return} \mapsto f_r) \in h}{\mathbf{handle} \ h \ s \ v \rightsquigarrow f_r \ s \ v} \text{ E-Return}$	
$\frac{\begin{array}{l} \mathbf{op} \notin \text{bop}(E) \wedge (\mathbf{op} \mapsto f_p) \in h \\ \text{where } k = \lambda s. \lambda x. \mathbf{handle} \ h \ s \ E[x] \end{array}}{\mathbf{handle} \ h \ s \ E[\mathbf{perform} \ op \ v] \rightsquigarrow f_p \ s \ v \ k} \text{ E-Perform}$	
$\frac{\begin{array}{l} (\mathbf{traverse} \mapsto f_t) \in h \\ \text{where } l = \lambda ss. \mathbf{for} \ x : n. \mathbf{handle} \ h \ (ss \ x) \ e \\ k = \lambda s. \lambda xs. \mathbf{handle} \ h \ s \ F[xs] \end{array}}{\mathbf{handle} \ h \ s \ F[\mathbf{for} \ x : n. e] \rightsquigarrow f_t \ n \ s \ l \ k} \text{ E-Traverse}$	
$\frac{e \rightsquigarrow e'}{E[e] \rightsquigarrow E[e']} \text{ E-Step}$	
$\frac{\forall 0 \leq i < n. e[x ::= i] \mapsto v_i}{F[\mathbf{for} \ x : n. e] \mapsto F[\langle v_0, \dots, v_{n-1} \rangle]} \text{ E-Parallel}$	

 TABLE 3.7: Operationele semantiek van λ^p

3.2.2 Operationele semantiek

De operationele semantiek in deze calculus is een grote-stap semantiek en is afgebeeld in Tabel 3.7. De regels **E-App**, **E-Index** zijn standaard. De **E-Step** is een standaard evaluatie regel voor evaluatie binnen de sequentiële evaluatie context E . **E-Return** past de return clause $(\mathbf{return} \mapsto f_r) \in h$ in de handler toe wanneer de handler een waarde tegenkomt $\mathbf{handle} \ h \ s \ v \rightsquigarrow f_r \ s \ v$. **E-Perform** past een algebraïsche operatie clause toe wanneer deze in de handler zit $(\mathbf{op} \mapsto f_p) \in h$ en de handler een effect tegenkomt: $\mathbf{handle} \ h \ s \ E[\mathbf{perform} \ op \ v] \rightsquigarrow f_p \ s \ v \ k$. **E-Parallel** is de regel waarin een **for**-constructie in parallel naar een lijst van waarden wordt geëvalueerd, de functie e wordt hierbij in parallel toegepast voor elke x in n . De meest essentiële regel voor de λ^p -calculus is **E-Traverse**. Zonder deze regels kan **E-Parallel** vast komen te zitten doordat er eventueel operaties moeten worden uitgevoerd zonder handler waardoor het programma foutief halt houdt op een normaalvorm. **E-Traverse** verhelpt dit fenomeen door de handler in de expressie e te duwen in l , de staat van het programma s en de resumptie aan te passen door middel van k en f_t .

Deze calculus definieert enkele regels niet expliciet zoals een forwarding regel voor operaties en aangezien de semantiek grote stappen neemt, evalueert een **for** constructie in een enkele stap naar een lijst van waarden. Het resultaat hiervan is dat bijkomende syntaxis en semantiek nodig zijn om deze calculus te vertalen naar een kleine-staps semantiek variant.

3.3 Gecombineerde calculus

De λ_{sc} -calculus ondersteunt de sequentiële behandeling van algebraïsche en scoped effecten. De λ^p -calculus ondersteunt de parallelle afhandeling van algebraïsche effecten. Hoofdstukken 5 en 6 behandelen de syntaxis en semantiek voor de gecombineerde calculus λ_{sc}^p die sequentiële en parallelle behandeling van algebraïsche en scoped effecten ondersteunt.

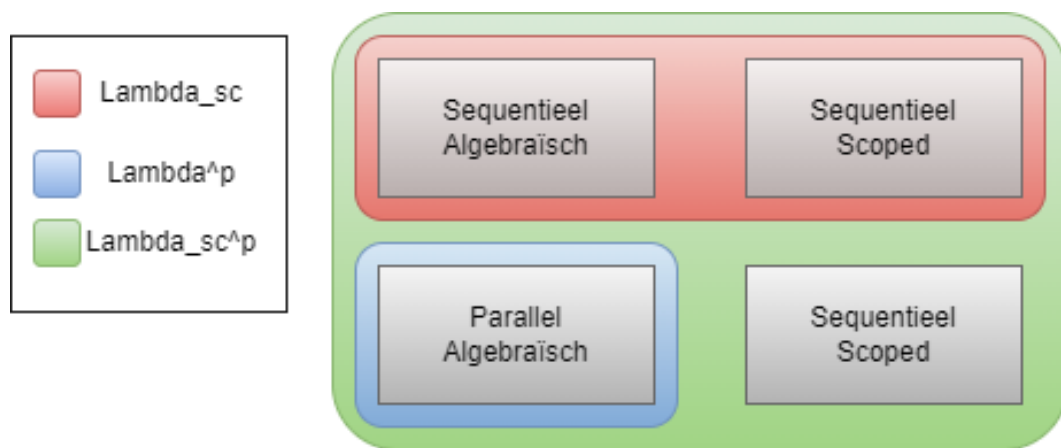
Hoofdstuk 4

Motivatie en Uitdagingen

Dit hoofdstuk behandelt de motivatie voor dit onderwerp en de uitdagingen die vermeldingswaardig zijn bij de zoektocht naar een gepaste oplossing.

4.1 Sequentiële vs parallelle uitvoering

De primaire motivatie voor de ontwikkeling van de calculus is het maken van een calculus die zowel parallelle als sequentiële uitvoering van zowel algebraïsche als scoped effecten ondersteunt. De λ_{sc} -calculus [2] ondersteunt sequentiële uitvoering van algebraïsche en scoped effecten. De λ^p -calculus ondersteunt parallelle uitvoering van algebraïsche effecten. Het doel van de λ_{sc}^p -calculus is om vrije keuze te geven over parallelle of sequentiële uitvoering van algebraïsche of scoped effecten zoals afgebeeld in Figuur 4.1.



FIGUUR 4.1: Scope van de verschillende calculi vermeld in deze thesis

4.1.1 Aanpak

Omdat parallelle of sequentiële uitvoering een verschil in semantiek geeft en in de effect handler aanpak de semantiek doorgeschoven is naar de handler, gebeurt dit ook in de λ_{sc}^p -calculus voor de semantiek van uitvoeringswijze. Concreet zal elke handler een clause uitwerken die bepaalt hoe een lijst van computatie met de effecten die de handler behandelt, behandeld worden.

4.1.2 Voorbeeld

```

do _ ← op accum "start";
for (1 : 2 : 3 : 4 : 5 : [ ]) (x. if x ≡ 2
                                then do _ ← op accum "!" (x. return x);
                                do _ ← op throw "error" (x. return x); (4.1)
                                op accum "unreachable" (x. return x)
                                else op accum x (x. return x));)
do _ ← op accum "end";

```

Het voorbeeld programma in Eq. 4.1 is een programma dat een accumulatie doet in een string waarbij een functie mapt over een met lijst met getallen van 1 tot 5. De functie gooit een fout als de input 2 is, anders voegt de functie de input bij het resultaat. Bij sequentiële uitvoering zouden we verwachten dat het programma de executie stopt vanaf het de functie met input 2 behandelt. Het resultaat van de accumulatie gaat dan "start 1!" zijn. Bij parallelle afhandeling gaat de exception bij input 2 geen effect kunnen hebben op de andere accumulaties die parallel gebeuren en zal het resultaat "start 1!345" zijn. Deze verschillende semantiek zou kunnen bekomen worden door 2 verschillende handlers te maken die het exception effect afhandelen waarbij de ene werkt op de sequentiële wijze en de andere op de parallelle wijze.

Hoofdstuk 5

Syntaxis

Tabel 5.1 toont de syntaxis voor de termen en types in de λ_{sc}^p calculus. De uitbreidingen op de λ_{sc} calculus zijn gemarkeerd. De syntaxis is uitgebreid met een lijst-structuur voor waarden, een lijst-structuur voor computaties en handler clauses om de parallelle afhandeling van effecten mogelijk te maken. Verder is ook een **let rec** constructie toegevoegd om recursieve functies mogelijk te maken, een **if** constructie en enkele hulp-functies voor paar- en lijst-manipulatie.

5.1 Lijst-structuur

Een mogelijke manier om parallelle afhandeling van effecten mogelijk te maken is functionaliteit toe te voegen waardoor een lijst van computaties die mogelijk effecten bevatten parallel behandeld kunnen worden. Daarvoor is het essentieel om datastructuren te hebben die meerdere computaties kunnen opslaan om zo de verschillende computaties die parallel moeten reduceren te groeperen. Een lijst-structuur is een datastructuur die relatief eenvoudig te implementeren is, weinig eigen complexe functies met zich meebrengt en bijgevolg relatief eenvoudig te vervangen door een andere datastructuur indien nodig. Een voorbeeld van een lijst van computaties is:

$$\begin{aligned} (39 + 49) : (29 + 74) : (100 + 61) : (41 + 56) : (97 + 67) : [] \\ \rightsquigarrow 88 : 103 : 161 : 97 : 164 : [] \end{aligned} \quad (5.1)$$

De reductie in dit voorbeeld kan parallel gebeuren voor elk element in de lijst.

Aangezien de λ_{sc} -calculus syntaxis een strikte scheiding heeft tussen waarden en termen ligt het voor de hand dat dit onderscheid er ook zou zijn voor lijsten in de λ_{sc}^p -calculus.

5.1.1 Lijst van waarden

Een lijst van waarden is zelf een waarde_p. De syntaxis is intuïtief, de lijst eindigt op een lege lijst $[]$ en wordt voorgegaan door één of meerdere waarden.

waarden v	$::=$	$() \mid (v_1, v_2) \mid x \mid \lambda x. c \mid h \mid lstv$	
handlers h	$::=$	handler { return $x \mapsto c_r$, $opr s$, fwd $f p k \mapsto c_f$; reduce $lstv k \mapsto c_{red}$ }	return clause effect clauses forwarding clause reducerende clause
lijst waarden $lstv$	$::=$	$[] \mid v : lstv$	lijst van waarden
effect clauses $opr s$	$::=$	\cdot op $l x k \mapsto c, opr s$ sc $l x p k \mapsto c, opr s$ for $c p k \mapsto c$	algebraïsch effect scoped effect for effect clause
computaties c	$::=$	return v op $l v (y. c)$ sc $l v (y. c_1) (z. c_2)$ for $v (y. c_1) (z. c_2)$ $v \star c$ do $x \leftarrow c_1 ; c_2$ $v_1 v_2$ let $x = v$ in c let rec $f = c_1$ in c_2 head v tail v empty v head c tail c empty c fst v snd v map $f v$ $lstc$	return waarde algebraïsch effect scoped effect for effect behandeling do clause applicatie let let rec head lstv tail lstv test lege lstv head lstc tail lstc test lege lstc first paar second paar map over lstv lijst computaties
lijst computaties $lstc$	$::=$	$[] \mid c : lstc$	lijst computaties
waarde types A, B, M	$::=$	$() \mid (A, B) \mid A \rightarrow \underline{C} \mid \underline{C} \Rightarrow \underline{D}$ α $\lambda \alpha. A$ $M A$	type variabele type operator abstractie type applicatie
type schemas σ	$::=$	$A \mid \forall \mu. \sigma \mid \forall \alpha. \sigma$	
computatie types $\underline{C}, \underline{D}$	$::=$	$A! \langle E \rangle$	
effect type rows E, F	$::=$	$\cdot \mid \mu \mid l; E$	
kinds K	$::=$	$* \mid K \rightarrow K$	
signatuur contexten Σ	$::=$	$\cdot \mid \Sigma, l : A \rightarrow B$	
type contexten Γ	$::=$	$\cdot \mid \Gamma, x : A \mid \Gamma, \mu \mid \Gamma, \alpha$	

TABEL 5.1: λ_{sc}^p Syntaxis

5.1.2 Lijst van computaties

Een lijst van computaties is een computatie, dit omdat de elementen in de lijst kunnen reduceren en semantische regels geïntroduceerd worden in Sectie 6 om een lijst te reduceren naar een lijst van normaalwaarden. Hiervoor moet de lijst zelf een computatie zijn omdat een reductie in de λ_{sc}^p -calculus een computatie naar een andere computatie mapt. De syntaxis voor de lijst is intuïtief en analoog aan de lijst van waarden, de lijst eindigt op een lege lijst `[]` en wordt voorgegaan door één of meerdere computaties.

5.1.3 Lijst-manipulatie functies

De lijst heeft minimaal een **head**, **tail** en **empty** functie nodig om elementen uit de lijst te halen en na te kijken of de lijst leeg is. Deze minimale implementatie is aanwezig in de syntaxis.

5.1.4 Map functie

Een map functie die een lijst van waarden omvormt naar een lijst van computaties met als argument een functie om toe te passen op de lijst van waarden.

5.2 Let rec

let rec helpt om recursieve functies te implementeren in de calculus. Recursieve functies zijn nodig om complexere functies te definiëren zoals een functie die alle elementen in een lijst opsomt:

```

let rec sumList = l.
    do n  $\leftarrow$  empty l;
    if n then return []
    else
        do h  $\leftarrow$  head l;
        do t  $\leftarrow$  tail l;
        do y  $\leftarrow$  reduce t;
        return (h + y) in
            do m  $\leftarrow$  reduce first;
            return (m + m', b)

```

(5.2)

5.3 Handler clauses

Deze sectie behandelt de clauses die moeten toegevoegd worden aan de handlers om parallelle behandeling van effecten mogelijk te maken.

5.3.1 Reduce clause

Het doel is om effecten parallel te behandelen in een lijst, maar niet altijd om als resultaat van de behandeling een lijst van waarden te hebben. Als voorbeeld hebben we Eq. 5.1 die we kunnen reduceren door de elementen op te sommen via Eq. 5.2 na de parallelle behandeling.

$$\begin{aligned} \text{sumList } ((39 + 49) : (29 + 74) : (100 + 61) : (41 + 56) : (97 + 67) : []) \\ \rightsquigarrow \text{sumList } (88 : 103 : 161 : 97 : 164 : []) \quad (5.3) \\ \rightsquigarrow 613 \end{aligned}$$

Aangezien lijsten niet generiek gereduceerd kunnen worden en compositie van handlers mogelijk moet zijn, heeft elke handler een **reduce** clause met als argumenten de resumptie k en de lijst van waarden $lstv$ die het resultaat is van de parallelle behandeling. De computatie in de **reduce** clause vervangt de originele resumptie na behandeling. Op deze manier kan elke handler een eigen manier definiëren om resultaten te reduceren. De operationele semantiek hiervan wordt gepresenteerd in Hoofdstuk 6.

5.3.2 For clause

De **for** clause bestaat uit het sleutelwoord **for**, gevolgd door een lijst van waarden, gevolgd door een mapping-functie $(y. c_1)$ die op alle waarden in de gegeven lijst wordt uitgevoerd, gevolgd door de resumptie $(z. c_2)$ die de continuatie van het programma bevat gegeven de resulterende lijst van waarden na toepassing van de mapping-functie. Deze clause wordt slechts geïmplementeerd door een enkele handler die enkel dit effect behandelt en vooraan het programma staat.

Hoofdstuk 6

Operationele Semantiek

Tabellen 6.1 en 6.2 toont de operationele semantiek voor de λ_{sc}^p -calculus. De toegevoegde regels zijn gemarkeerd. De meest cruciale toevoegingen zijn de regels rond de toevoeging van het nieuwe sleutelwoord **for**, namelijk **E-DoFor** voor het aaneenrijgen van computaties, **E-HandFor** voor het behandelen van de **for**-constructie door de handler die dit behandelt en **E-FwdFor** voor het forwarden van **for**. Het inzicht dat dit niet generiek kan gebeuren, is hierbij belangrijk. Gelijkaardige regels waren nodig voor de **sc** en **op** sleutelwoorden met dezelfde functies. Verder zijn enkele toevoegingen gedaan om recursieve functie-behandeling mogelijk te maken (**E-Letrec**) en lijst- en andere hulp-functies (Tabel 6.2) expliciet toe te voegen.

6.1 Letrec: Recursieve functies

De **let rec** constructie biedt de mogelijkheid om recursieve functies te definiëren in de calculus. De clause bestaat uit het **let rec** sleutelwoord gevolgd door een variabele-naam f gevolgd door een computatie c_1 waar de variabele f in kan voorkomen gevolgd door het sleutelwoord **in** gevolgd door de computatie waarin de variabele f kan voorkomen. Deze clause wordt gereduceerd naar c_2 met f in c_2 vervangen door c_1 met f vervangen door **let rec** = c_1 **in** f . Deze inferentie regel is een aangepaste versie van de regel beschreven in hoofdstuk twee van het boek "Principles of Programming Languages" [6].

6.2 For clause

Het **for** sleutelwoord laat toe om een computatie te mappen over een lijst van waarden. De gewenste semantiek om computaties met eventueel effecten te mappen over een lijst van waarden is niet mogelijk via een klassieke map functie omdat de computatie in dit geval niet puur is en alle relevante handlers niet noodzakelijk binnen de computatie zitten. Het gevolg is dat een klassieke map aanpak vastloopt op een normaalvorm (**return**, **op** of **sc**). Om een correcte semantiek voor het **for** sleutelwoord te bekomen zijn drie semantische regels nodig.

$\boxed{c \rightsquigarrow c'}$	Computatie reductie	
$\frac{}{(\lambda x. c) v \rightsquigarrow c [v / x]}$	E-AppAbs	$\frac{}{\mathbf{let} \ x = v \ \mathbf{in} \ c \rightsquigarrow c [v / x]}$ E-Let
$\frac{}{\mathbf{let} \ \mathit{rec} \ f = c_1 \ \mathbf{in} \ c_2 \rightsquigarrow c_2 [c_1 [(\mathbf{let} \ \mathit{rec} \ f = c_1 \ \mathbf{in} \ f) / f] / f]}$	E-Letrec	
$\frac{c_1 \rightsquigarrow c'_1}{\mathbf{do} \ x \leftarrow c_1 ; c_2 \rightsquigarrow \mathbf{do} \ x \rightsquigarrow c'_1 ; c_2}$	E-Do	$\frac{}{\mathbf{do} \ x \leftarrow \mathbf{return} \ v ; c_2 \rightsquigarrow c_2 [v / x]}$ E-DoRet
$\frac{}{\mathbf{do} \ x \leftarrow \mathbf{op} \ l \ v \ (y. c_1) ; c_2 \rightsquigarrow \mathbf{op} \ l \ v \ (y. \mathbf{do} \ x \leftarrow c_1 ; c_2)}$	E-DoOp	
$\frac{}{\mathbf{do} \ x \leftarrow \mathbf{sc} \ l \ v \ (y. c_1) \ (z. c_2) ; c_3 \rightsquigarrow \mathbf{sc} \ l \ v \ (y. c_1) \ (z. \mathbf{do} \ x \leftarrow c_2 ; c_3)}$	E-DoSc	
$\frac{}{\mathbf{do} \ x \leftarrow \mathbf{for} \ \mathit{lstv} \ (y. c_1) \ (z. c_2) ; c_3 \rightsquigarrow \mathbf{for} \ \mathit{lstv} \ (y. c_1) \ (z. \mathbf{do} \ x \leftarrow c_2 ; c_3)}$	E-DoFor	
$\frac{c \rightsquigarrow c'}{h \star c \rightsquigarrow h \star c'}$	E-Hand	$\frac{(\mathbf{return} \ x \mapsto c_r) \in h}{h \star \mathbf{return} \ v \rightsquigarrow c_r [v / x]}$ E-HandRet
$\frac{(\mathbf{op} \ l \ x \ k \mapsto c) \in h}{h \star \mathbf{op} \ l \ v \ (y. c_1) \rightsquigarrow c [v / x, (\lambda y. h \star c_1) / k]}$	E-HandOp	
$\frac{(\mathbf{op} \ l \ _ _) \notin h}{h \star \mathbf{op} \ l \ v \ (y. c_1) \rightsquigarrow \mathbf{op} \ l \ v \ (y. h \star c_1)}$	E-FwdOp	
$\frac{(\mathbf{sc} \ l \ x \ p \ k \mapsto c) \in h}{h \star \mathbf{sc} \ l \ v \ (y. c_1) \ (z. c_2) \rightsquigarrow c [v / x, (\lambda y. h \star c_1) / p, (\lambda z. h \star c_2) / k]}$	E-HandSc	
$\frac{(\mathbf{sc} \ l \ _ _ _) \notin h \quad (\mathbf{fwd} \ f \ p \ k \mapsto c_f) \in h \quad g = \lambda(p', k'). \mathbf{sc} \ l \ v \ (y. p' y) \ (z. k' z)}{h \star \mathbf{sc} \ l \ v \ (y. c_1) \ (z. c_2) \rightsquigarrow c_f [(\lambda y. h \star c_1) / p, (\lambda z. h \star c_2) / k, g / f]}$	E-FwdSc	
$\frac{(\mathbf{for} \ \mathit{lstv}_1 \ p \ k \mapsto c_{for}) \in h}{h \star \mathbf{for} \ \mathit{lstv}_2 \ (y. c_1) \ (z. c_2) \rightsquigarrow c_{for} [\mathit{lstv}_2 / \mathit{lstv}_1, (y. h \star c_1) / p, (z. h \star c_2) / k]}$	E-HandFor	
$\frac{(\mathbf{for} \ _ _ _) \notin h \quad (\mathbf{red} \ \mathit{lstv} \ k \mapsto c_{red}) \in h}{h \star \mathbf{for} \ \mathit{lstv}_1 \ (y. c_1) \ (z. c_2) \rightsquigarrow \mathbf{for} \ (y. h \star c_1) \ (w. c_{red} [w / \mathit{lstv}, (z. h \ \mathit{starc}_2) / k])}$	E-FwdFor	

 TABEL 6.1: Operationele semantiek van λ_{sc}^p

$c \rightsquigarrow c'$	Computatie reductie
$\frac{}{\mathbf{map} \ f \ (v_1 : v_2 : \dots : v_n) \rightsquigarrow ((f \ v_1) : (f \ v_2) : \dots : (f \ v_n))} \text{E-Map}$	
$\frac{\begin{array}{c} normal(c) : c \in \mathbf{return} _ \mid \mathbf{op} _ _ _ \mid \mathbf{sc} _ _ _ \mid \mathbf{for} _ _ _ _ \\ \exists j !normal(c_j) \quad \forall i : c_i \rightsquigarrow c'_i \leftarrow !normal(c_i) \end{array}}{(c_1 : \dots : c_j : \dots : c_{i1} : \dots : c_{i2} : \dots : c_n : [\]) \rightsquigarrow (c_1 : \dots : c'_j : \dots : c'_{i1} : \dots : c'_{i2} : \dots : c'_n : [\])} \text{E-ParList}$	
$\frac{}{((\mathbf{return} \ v_1) : (\mathbf{return} \ v_2) : \dots : (\mathbf{return} \ v_n) : [\]) \rightsquigarrow \mathbf{return} \ (v_1 : v_2 : \dots : v_n : [\])} \text{E-ListRet}$	
$\frac{}{\mathbf{head} \ (v_1 : lstv) \rightsquigarrow \mathbf{return} \ v_1} \text{E-HeadLstv}$	$\frac{}{\mathbf{head} \ (c_1 : lstc) \rightsquigarrow \mathbf{return} \ c_1} \text{E-HeadLstc}$
$\frac{}{\mathbf{tail} \ (v_1 : lstv) \rightsquigarrow \mathbf{return} \ lstv} \text{E-TailLstv}$	$\frac{}{\mathbf{tail} \ (c_1 : lstc) \rightsquigarrow \mathbf{return} \ lstc} \text{E-TailLstc}$
$\frac{}{\mathbf{empty} \ [\] \rightsquigarrow \mathbf{return} \ true} \text{E-EmptyTrue}$	
$\frac{}{\mathbf{empty} \ (v : lstv) \rightsquigarrow \mathbf{return} \ false} \text{E-EmptyLstvFalse}$	
$\frac{}{\mathbf{empty} \ (c : lstc) \rightsquigarrow \mathbf{return} \ false} \text{E-EmptyLstcFalse}$	
$\frac{}{\mathbf{fst} \ (v_1, v_2) \rightsquigarrow \mathbf{return} \ v_1} \text{E-First}$	$\frac{}{\mathbf{snd} \ (v_1, v_2) \rightsquigarrow \mathbf{return} \ v_2} \text{E-Second}$
$\frac{c_1 \rightsquigarrow c'_1}{\mathbf{if} \ c_1 \ \mathbf{then} \ c_2 \ \mathbf{else} \ c_3 \rightsquigarrow \mathbf{if} \ c'_1 \ \mathbf{then} \ c_2 \ \mathbf{else} \ c_3} \text{E-If}$	
$\frac{}{\mathbf{if} \ true \ \mathbf{then} \ c_1 \ \mathbf{else} \ c_2 \rightsquigarrow c_1} \text{E-IfTrue}$	
$\frac{}{\mathbf{if} \ false \ \mathbf{then} \ c_1 \ \mathbf{else} \ c_2 \rightsquigarrow c_2} \text{E-IfFalse}$	

TABEL 6.2: Operationele semantiek van λ_{sc}^p , lijst- en hulp-functies

6.2.1 E-DoFor

E-DoFor laat toe om een **do** statement door te schuiven naar de continuatie van de **for** constructie en tegelijk de computatie na de **for** constructie binnen de continuatie te brengen. Deze regel is nodig om programma's correct aan elkaar te rijgen rond het gebruik van **for** constructies en is zeer gelijkaardig aan **E-DoRet**, **E-DoOp** en **E-DoSc**.

6.2.2 E-FwdFor

E-FwdFor behandelt de forwarding voor het geval dat de **for** clause niet door de handler geïmplementeerd wordt. De functie van de forwarding is tweevoudig. Enerzijds schuift deze regel de handler binnen de computatie die te mappen is en binnen de resumptie. Anderzijds dicteert de **red** $lstv\ k \mapsto c_{red}$ clause hoe de continuatie moet veranderen.

6.2.3 E-HandFor

E-HandFor laat toe om een **for** constructie te behandelen door een handler. De **for** constructie wordt hierbij vervangen door de c_{for} computatie in de handler waarbij de handler eveneens naar binnen geschoven wordt in de te mappen functie en de continuatie. Een **for** clause wordt behandeld door een handler die enkel deze effect clause behandelt, geen algebraïsche of scoped effecten. Aangezien deze behandeling slechts mag gebeuren wanneer alle handlers binnen de **for** constructie zijn geschoven, is het raadzaam om de handler die deze clause implementeert als buitenste handler rond het programma te gebruiken.

6.3 Lijst- en hulp-functies

Deze sectie behandelt toevoegingen aan de calculus die nodig zijn voor lijst-manipulatie en hulp-functies die de implementatie van de semantiek van de **for** clause vergemakkelijken.

6.3.1 E-Map

Via deze regel kan in de calculus een lijst van waarden worden omgevormd naar een lijst van computaties met behulp van een computatie f . Deze regel is essentieel om deze omvorming te maken. De **E-ListRet**-regel helpt bij de omvorming in de andere richting.

6.3.2 E-ParList

E-ParList laat parallelle reductie van elementen in een lijst van computaties toe. De regel laat toe dat alle elementen in de lijst die een stap kunnen maken, deze stap zetten. De voorwaarde om nog een stap te kunnen zetten is dat de computatie niet

in normaalvorm is, met andere woorden niet in de vorm:

$$\mathbf{return} _ \mid \mathbf{op} _ _ _ \mid \mathbf{sc} _ _ _ \mid \mathbf{for} _ _ _ \quad (6.1)$$

6.3.3 E-ListRet

Als een lijst van computaties gereduceerd kan worden naar een lijst van **return** clauses van waarden, dan kan de lijst vervangen worden door een **return** clause van de lijst van waarden. Deze clause laat, in combinatie met **E-DoRet**, toe om een lijst van computaties om te vormen naar een lijst van waarden.

6.3.4 Lijst-manipulatie

E-HeadLstv, **E-HeadLstc**, **E-TailLstv**, **E-TailLstc**, **E-EmptyTrue**, **E-EmptyLstvFalse** en **E-EmptyLstcFalse** zijn eenvoudige functies die lijst-manipulatie van lijsten van waarden en lijsten van computaties vergemakkelijken.

6.3.5 Paar-manipulatie

E-First en **E-Second** laten manipulatie van paren van waarden toe.

6.3.6 If constructie

E-If, **E-IfTrue** en **E-IfFalse** implementeren op standaardwijze een **if** clause.

Hoofdstuk 7

Type- en Effect-Systeem

...

7.1 Lijsten

...

Hoofdstuk 8

Uitgewerkte Voorbeelden

...

8.1 Voorbeelden met Scoped Effecten

...

8.2 Voorbeelden met Scoped Effecten

...

8.3 Voorbeelden met Beide Effecten

...

Hoofdstuk 9

Metatheorie

...

9.1 Lemma's

...

9.2 Behoud

...

9.3 Vooruitgang

...

Hoofdstuk 10

Evaluatie

...

10.1 Bijdrage

...

10.2 Bruikbaarheid

...

10.3 Correctheid

...

10.4 Implementatie

...

10.5 Backwards compatibility

...

10.5.1 λ_{sc}

...

10.5.2 λ_p

...

Hoofdstuk 11

Gerelateerd Werk

...

Hoofdstuk 12

Besluit

...

12.1 Resultaten

...

12.2 Beperkingen

...

12.3 Toekomstig werk

...

Bijlagen

Bibliografie

- [1] A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84:108–123, 1 2015.
- [2] R. Bosman, B. van den Berg, W. Tang, and T. Schrijvers. A calculus for scoped effects& handlers. *under review*, 2022.
- [3] A. Haas, A. Rossberg, D. Schuff, B. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the web up to speed with webassembly. *PLDI 2017: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 12 2017.
- [4] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 333–343, 1995.
- [5] E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991. Selections from 1989 IEEE Symposium on Logic in Computer Science.
- [6] Z. Palmer, M. Grant, and S. Smith. *Principles of Programming Languages*. John Hopkins University, 2009.
- [7] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.
- [8] M. Piróg, T. Schrijvers, N. Wu, and M. Jaskelioff. Syntax and semantics for operations with scopes. *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 809–818, 2018.
- [9] M. Pretnar. An introduction to algebraic effects and handlers. invited tutorial paper. *Electronic Notes in Theoretical Computer Science*, 319:19–35, 12 2015.
- [10] N. Wu, T. Schrijvers, and R. Hinze. Effect handlers in scope. *SIGPLAN Not.*, 49:1–12, 9 2014.
- [11] N. Xie, D. D. Johnson, D. Maclaurin, and A. Paszke. Parallel algebraic effect handlers. *CoRR*, abs/2110.07493, 2021.

- [12] Z. Yang, M. Paviotti, N. Wu, B. van den Berg, and T. Schrijvers. Structured handling of scoped effects: Extended version. *CoRR*, abs/2201.10287, 2022.