KU LEUVEN

FACULTY OF
ENGINEERING SCIENCE

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

Academiejaar 2022 – 2023

Voorwoord

...

Lander Debreyne

Inhoudsopgave

Voorwoord i

Samenvatting iv

Lijst van figuren en tabellen v

1 Inleiding 1
- 1.1 Effecten . . . 1
- 1.2 Monads en effect handlers . . . 2
- 1.3 Programmeertaal-onderzoek en industrie . . . 2
- 1.4 Gecombineerde calculus . . . 3
- 1.5 Doel . . . 4
- 1.6 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 Parallelle algebraïsche effecten . . . 8
- 2.7 Calculi . . . 8
- 2.8 Andere computationele effecten . . . 10

3 Startpunt 13
- 3.1 λ_{sc} : Scoped Effecten . . . 13
- 3.2 λ^{p} : Parallelle algebraïsche effect handlers . . . 18
- 3.3 Gecombineerde calculus . . . 22

4 Motivatie en Uitdagingen 23
- 4.1 Sequentiële vs parallelle uitvoering . . . 23

5 Syntaxis 25
- 5.1 Lijst-structuur . . . 25
- 5.2 Recursieve let . . . 28
- 5.3 Handler clausules . . . 28

6 Operationele Semantiek 31
- 6.1 Letrec: Recursieve functies . . . 31
- 6.2 For clausule . . . 31

6.3 Lijst- en hulp-functies 34

7 Type- en Effect-Systeem **37**
7.1 Lijsten 37

8 Uitgewerkte Voorbeelden **39**
8.1 Voorbeelden met Scoped Effecten 39
8.2 Voorbeelden met Scoped Effecten 39
8.3 Voorbeelden met Beide Effecten 39

9 Metatheorie **41**
9.1 Lemma's 41
9.2 Behoud 41
9.3 Vooruitgang 41

10 Evaluatie **43**
10.1 Bijdrage 43
10.2 Bruikbaarheid 43
10.3 Correctheid 43
10.4 Implementatie 43
10.5 Backwards compatibility 43

11 Gerelateerd Werk **45**

12 Besluit **47**
12.1 Resultaten 47
12.2 Beperkingen 47
12.3 Toekomstig werk 47

Bibliografie **51**

Samenvatting

...

Lijst van figuren en tabellen

Lijst van figuren

4.1 Scope van de verschillende calculi vermeld in deze thesis 23

Lijst van tabellen

2.1 Pure, simpel getypeeerde λ-calculus syntaxis 9
2.2 Operationele semantiek van pure, simpel getypeerde λ-calculus 9
2.3 Typesysteem van pure, simpel getypeerde λ-calculus 10

3.1 Syntaxis voor λ_{sc} 14
3.2 Operationele semantiek van λ_{sc} 16
3.3 Waarde typering van λ_{sc} 18
3.4 Computatie typering van λ_{sc} 19
3.5 Handler typering van λ_{sc} 20
3.6 Syntaxis voor λ^p 20
3.7 Operationele semantiek van λ^p 21

5.1 λ^p_{sc} Syntaxis 26

6.1 Operationele semantiek van λ^p_{sc} 32
6.2 Operationele semantiek van λ^p_{sc}, lijst- en hulp-functies 33

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 taal die bestaat uit een set woorden, symbolen en regels. 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. De syntaxis beschrijft hoe de woorden en symbolen uit de programmeertaal correct geschreven programma's vormen. De operationele semantiek beschrijft hoe programma's uitgevoerd worden. Het type-en effect-systeem beschrijft hoe de termen in de syntaxis een type krijgen.

1.1 Effecten

Effecten in programma's zijn interacties met een omgeving buiten de lokale omgeving waarin het programma wordt uitgevoerd. Mogelijke effecten zijn het weergeven van informatie op het scherm, het opslaan van gegevens in een database. 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 en effect handlers

De meest industrie-relevante aanpak om effecten te modelleren in pure, functionele programmeertalen is de *monad* [9]. 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*[8] populair om modulaire compositie van monads, en bijgevolg effecten, te realiseren. Dit gebeurt door verschillende types monads te combineren tot een enkele, gecombineerde monad.
Een tweede constructie om effecten te modelleren is het gebruik van *algebraïsche effecten [13] 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.

1.3 Programmeertaal-onderzoek en industrie

WebAssembly [6] 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.

Een online effect bibliografie[1] verzamelt toepassingen en applicaties die gerelateerd zijn aan computationele effecten. Koka [7], Eff [2] en Effekt [5] zijn onderzoeksprogrammeertalen ontwikkeld met als doel programmeren met effect handlers. De Haskell-library *fused-effects* [2] biedt functionaliteit om in haskell programma's te maken met effect handlers. Deze library wordt gebruikt door Github in de semantic[3] library. Verder is er de Pyro [3] library voor flexibel en schaalbaar diep probabilistisch programmeren die volgens de auteurs gebouwd is op Poutine, een library om te programmeren met effect handlers, voor de flexibiliteit en separation of concerns die de aanpak met zich meebrengt.

[1] https://github.com/yallop/effects-bibliography
[2] https://github.com/fused-effects/fused-effects
[3] https://github.com/github/semantic

1.4 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.

- Functie compositie: Het modulair combineren van verschillende effecten en effect handlers 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 beperkt aangezien deze effecten 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 effecten 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 [4] beschrijft een calculus voor algebraïsche en scoped effecten. Het resultaat is een calculus die expressiviteit biedt die dichter komt bij de expressiviteit van monads. 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. Parallelle algebraïsche effecten [17] bieden deze mogelijkheid. De λ^p calculus maakt parallelle afhandeling van effecten mogelijk en levert performantieverbeteringen voor parallelliseerbare algebraïsche effecten.

1.5 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 meta-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 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?

1.6 Overzicht

Hoofdstuk 2 geeft meer achtergrondinformatie over de onderwerpen die in deze masterproef worden behandeld. Vervolgens wordt in hoofdstuk 3 de literatuur besproken waarop deze thesis voortbouwt. De motivatie voor dit onderwerp en de uitdagingen zijn het onderwerp van hoofdstuk 4. De hoofdstukken die daarop volgen vormen de bijdrage van deze masterproef en stellen de λ^{p}_{sc}-calculus voor met de syntaxis in hoofdstuk 5, de operationele semantiek in hoofdstuk 6, het type- en effect-systeem in hoofdstuk 7, voorbeelden in hoofdstuk 8 en de metatheorie in hoofdstuk 9. Vervolgens geeft hoofdstuk 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 [2] 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 modulair componeerbare manier, waardoor meer modulaire en herbruikbare programma's kunnen worden gemaakt. Dit is bijzonder interessant wanneer complexe effecten moeten worden gecontroleerd, zoals in parallelle 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 $A \twoheadrightarrow B$, wat aanduidt dat het effect een term van type A omvormt naar type B. Het $choose : () \twoheadrightarrow Bool$ 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 deze masterproef is als volgt:

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

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:

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

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

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

Algemeen opgesteld, ziet deze eigenschap er zo uit:

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

Het effect van deze eigenschap is enerzijds dat de **do** clausule 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. Effect handlers hebben drie interessante eigenschappen, overloading van effecten, functie compositie en effectinteracties.

2.3.1 Overloading van effecten

Overloading van effecten is simpel te demonstreren door twee verschillende handlers te definiëren en deze hetzelfde programma te laten behandelen. Het resultaat zal in beide gevallen anders zijn, wat aangeeft dat handlers een interessante invloed hebben op de semantiek van het programma.
De handler h_{True} selecteert de *true* tak van de computatie voor het *choose* effect:

$$\begin{aligned}h_{True} = \mathbf{handler}\ \{\ &\mathbf{return}\ x \mapsto \mathbf{return}\ x\\ &,\ \mathbf{op}\ choose\ \_\ k \mapsto k\ true\}\end{aligned} \tag{2.5}$$

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

$$\text{"}We\ are\ eating\ pasta\ tonight!\text{"} \tag{2.6}$$

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

$$\begin{aligned} h_{True} = \textbf{handler}\ \{\ &\textbf{return}\ x \mapsto \textbf{return}\ [x] \\ &,\ \textbf{op}\ choose\ \_\ k \mapsto \textbf{do}\ t \leftarrow k\ true\ ;\ \textbf{do}\ f \leftarrow k\ false\ \ t +\!\!+ f\} \end{aligned} \tag{2.7}$$

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

$$[\text{"}We\ are\ eating\ pasta\ tonight!\text{"},\ \text{"}We\ are\ eating\ pizza\ tonight!\text{"}] \tag{2.8}$$

hebben. Deze twee resultaten zijn een interessante toepassing van overloading van effecten door verschillende handlers te definiëren voor hetzelfde effect.

2.3.2 Functie compositie

Functie compositie betekent dat het mogelijk moet zijn om programma's te schrijven die verschillende effecten bevatten.

2.3.3 Effect-interacties

...

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. Controlestroom-veranderende effecten daarentegen vereisen meer verfijnde mechanismen voor hun behandeling, die buiten de mogelijkheden van algebraïsche effecten vallen.

2.5 Effecten met scope

Scoped effecten [4], [16], [18], [12] 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 [4] 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:

$$\textbf{sc}\ l\ v\ (y.\ c_1)\ (z.\ c_2) \tag{2.9}$$

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 bereking 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 clausule hebben van de volgende vorm:

$$\textbf{fwd}\ f\ p\ k \mapsto c_f \tag{2.10}$$

Deze clausule wordt meer in detail besproken in Sectie 3.1 en [4].

2.6 Parallelle algebraïsche effecten

Effect handlers behandelen effecten standaard sequentieel. De λ^p calculus [17] maakt parallelle afhandeling van algebraïsche effecten mogelijk. Deze calculus wordt behandeld in Sectie 3.2. De λ^p-calculus maakt gebruikt van een constructie

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

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

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

Deze **traverse** clausule 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 [11].

2.7.1 Termen, waarden, computaties en operaties

De kleinste bouwblokken voor programma's in calculi zijn termen. Termen zijn opgesplitst in waarden en computaties waarbij een term bij een van beide hoort. Een waarde is een term die niet kan evalueren of reduceren. Een computatie is in tegenstelling tot een waarde een term die wel kan evalueren of reduceren. Effect handlers introduceren operaties die in deze context duiden op syntax-constructies die de oproep van een effect symboliseren.

waarden v	$::=$	x	variabele
	$\mid$	$\lambda\, x : A.\, c$	abstractie
computaties c	$::=$	$v\ v$	applicatie
types A, B	$::=$	$A \to B$	type van functies
contexten Γ	$::=$	$.$	Lege context
	$\mid$	$\Gamma,\, x : A$	term variabele binding

TABEL 2.1: Pure, simpel getypeeerde λ-calculus syntaxis

$\boxed{c \rightsquigarrow c'}$ Computatie reductie

$$\frac{c_1 \rightsquigarrow c_1'}{c_1\ c_2 \rightsquigarrow c_1'\ c_2}\ \text{E-App1}$$

$$\frac{c \rightsquigarrow c'}{v\ c \rightsquigarrow v\ c'}\ \text{E-App2}$$

$$\frac{}{(\lambda x : A.\, c)\ v \rightsquigarrow [x \mapsto v]\, c}\ \text{E-AppAbs}$$

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

2.7.2 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 waarden, computaties, types en contexten en definieert voor elk van deze welke vormen ze kunnen aannemen. Waarden en computaties hebben types. De typering van waarden en computaties is vastgelegd in de type-context. Het verschil tussen waarden en computaties is dat waarden zich in een irreduceerbare, normaalvorm bevinden en computaties kunnen reduceren. De operationele semantiek legt de regels voor reductie vast.

2.7.3 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.

$$\boxed{\Gamma \vdash v \ : \ A} \quad \text{waarde typering}$$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \ : \ A} \ \text{T-Var}$$

$$\frac{\Gamma,\ x : A \vdash c \ : \ B}{\Gamma \vdash \lambda x : A.\ c \ : \ A \to B} \ \text{T-Abs}$$

$$\boxed{\Gamma \vdash c \ : \ A} \quad \text{computatie typering}$$

$$\frac{\Gamma \vdash v_1 \ : \ A \to B \qquad \Gamma \vdash v_2 \ : \ A}{\Gamma \vdash v_1\ v_2 \ : \ B} \ \text{T-App}$$

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

2.7.4 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**).

2.7.5 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 een computatie c en een computatie c' bestaat zodat $c \rightsquigarrow c'$ en voor c' hetzelfde geldt.

Voor deze calculus kan eveneens een bewijs van preservatie geleverd worden wat inhoudt dat voor als voor termen c en c' geldt: $\Gamma \vdash c \ : \ A$ en $c \rightsquigarrow c'$ dan geldt $\Gamma \vdash c' \ : \ A$. Dit betekent dat als een term goed getypeerd is, elke geldige reductie van de term ook goed getypeerd is.

2.8 Andere computationele effecten

Deze masterproef focust op de algebraïsche en scoped effecten en de parallelle behandeling van effecten. Dit zijn niet de enige types computationele effecten. Deze sectie bespreekt enkele andere soorten effecten die verder niet behandeld worden in de calculus.

2.8.1 Latente effecten

Latente effecten [15] zijn een generieke klasse van effecten waarbij de uitvoer van een computatie uitgesteld wordt. Dit is controle-stroom controlerend mechanisme dat verschilt van de mogelijkheden van algebraïsche en scoped effecten.

2.8.2 Asynchrone effecten

De paper *Asynchronous Effects* [1] introduceert een calculus om algebraïsche effecten asynchroon te behandelen.

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 λ_{sc}[4], een calculus die sequentiële behandeling van scoped en algebraïsche effecten ondersteunt. Het startpunt voor de parallelle behandeling van effecten is λ^p[17], een calculus die parallelle algebraïsche effecten ondersteunt.

3.1 λ_{sc} : Scoped Effecten

Deze sectie stelt de calculus voor algebraïsche en scoped effecten [4] die als startpunt dient voor deze masterproef.

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 clausule, geen of meerdere effect clausules en een forwarding clausule. De return clausule $\mathbf{return}\, x \mapsto c_r$ duidt erop dat het resultaat x door de handler wordt behandeld door de computatie c_r. Een algebraïsche effect clausule $\mathbf{op}\, l\, x\, k \mapsto c$ 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 clausule verwerkt door de computatie c toe te passen. Een scoped effect clausule $\mathbf{sc} l\, x\, p\, k \mapsto c$ is gelijkaardig aan een algebraïsche effect clausule met een verschillend sleutelwoord **sc** en bijkomende een bewerking in scope p. De forwarding clausule $\mathbf{fwd}\, f\, p\, k \mapsto c_f$, die elke handler

waarden v	::=	$() \mid (v_1,\, v_2) \mid x \mid \lambda x.\, c \mid h$	
handlers h	::=	$\mathbf{handler}\,\{\ \mathbf{return}\, x \mapsto c_r$	return clausule
		$,\, oprs$	effect clausules
		$,\, \mathbf{fwd}\, f\, p\, k \mapsto c_f\ \}$	forwarding clausule
effect clausules $oprs$	::=	$.$	
	$\mid$	$\mathbf{op}\, l\, x\, k \mapsto c,\, oprs$	algebraïsche effect clausules
	$\mid$	$\mathbf{sc}\, l\, x\, p\, k \mapsto c,\, oprs$	scoped effect clausules
computaties c	::=	$\mathbf{return}\, v$	return waarde
	$\mid$	$\mathbf{op}\, l\, v\, (y.\, c)$	algebraïsch effect
	$\mid$	$\mathbf{sc}\, l\, v\, (y.\, c_1)\, (z.\, c_2)$	scoped effect
	$\mid$	$v \star c$	behandeling
	$\mid$	$\mathbf{do}\, x \leftarrow c_1\ ;\ c_2$	do clausule
	$\mid$	$v_1\, v_2$	applicatie
	$\mid$	$\mathbf{let}\, x = v\, \mathbf{in}\, c$	let
waarde types $A,\, B,\, M$	::=	$() \mid (A,\, B) \mid A \to \underline{C} \mid \underline{C} \Rightarrow \underline{D}$	
	$\mid$	α	type variabele
	$\mid$	$\lambda\, \alpha.\, A$	type operator abstractie
	$\mid$	$M\, A$	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 $E,\, F$	::=	$. \mid \mu \mid l;\, E$	
kinds K	::=	$* \mid K \to K$	
signatuur contexten Σ	::=	$. \mid \Sigma,\, l\ :\ A \rightarrowtriangle B$	
type contexten Γ	::=	$. \mid \Gamma,\, x\ :\ A \mid \Gamma,\, \mu \mid \Gamma,\, \alpha$	

Tabel 3.1: Syntaxis voor λ_{sc}

moet implementeren, zorgt voor expliciete forwarding wanneer de te behandelen scoped effect clausule niet geïmplementeerd wordt door de handler. De computatie c_f specifieert hoe de onbekende scoped effect clausule moet aangepast worden en kan 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 clausule **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 clausules 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 $A \to \underline{C}$, 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.

$\boxed{c \rightsquigarrow c'}$ Reductie van computaties

$$\frac{}{(\lambda x.\, c)\; v \rightsquigarrow c\,[\,v \,/\, x\,]}\ \text{E-AppAbs} \qquad \frac{}{\mathbf{let}\; x \;=\; v\; \mathbf{in}\; c \rightsquigarrow c\,[\,v \,/\, x\,]}\ \text{E-Let}$$

$$\frac{c_1 \rightsquigarrow c_1'}{\mathbf{do}\; x \leftarrow c_1\;;\; c_2 \rightsquigarrow \mathbf{do}\; x \rightsquigarrow c_1'\;;\; c_2}\ \text{E-Do} \qquad \frac{}{\mathbf{do}\; x \leftarrow \mathbf{return}\; v\;;\; c_2 \rightsquigarrow c_2\,[\,v \,/\, x\,]}\ \text{E-DoRet}$$

$$\frac{}{\mathbf{do}\; x \leftarrow \mathbf{op}\; l\; v\; (y.\, c_1)\;;\; c_2 \rightsquigarrow \mathbf{op}\; l\; v\; (y.\, \mathbf{do}\; x \leftarrow c_1\;;\; c_2)}\ \text{E-DoOp}$$

$$\frac{}{\mathbf{do}\; x \leftarrow \mathbf{sc}\; l\; v\; (y.\, c_1)\; (z.\, c_2)\;;\; c_3 \rightsquigarrow \mathbf{sc}\; l\; v\; (y.\, c_1)\; (z.\, \mathbf{do}\; x \leftarrow c_2\;;\; c_3)}\ \text{E-DoSc}$$

$$\frac{c \rightsquigarrow c'}{h \star c \rightsquigarrow h \star c'}\ \text{E-Hand} \qquad \frac{(\mathbf{return}\; x \mapsto c_r) \in h}{h \star \mathbf{return}\; v \rightsquigarrow c_r\,[\,v \,/\, x\,]}\ \text{E-HandRet}$$

$$\frac{(\mathbf{op}\; l\; x\; k \mapsto c) \in h}{h \star \mathbf{op}\; l\; v\; (y.\, c_1) \rightsquigarrow c\,[\,v \,/\, x,\; (\lambda y.\, h \star c_1) \,/\, k]}\ \text{E-HandOp}$$

$$\frac{(\mathbf{op}\; l\; \_\; \_) \notin h}{h \star \mathbf{op}\; l\; v\; (y.\, c_1) \rightsquigarrow \mathbf{op}\; l\; v\; (y.\, h \star c_1)}\ \text{E-FwdOp}$$

$$\frac{(\mathbf{sc}\; l\; x\; p\; k \mapsto c) \in h}{h \star \mathbf{sc}\; l\; v\; (y.\, c_1)\; (z.\, c_2) \rightsquigarrow c\,[\,v \,/\, x,\; (\lambda\, y.\, h \star\; c_1) \,/\, p, (\lambda z.\, h \star c_2) \,/\, k\,]}\ \text{E-HandSc}$$

$$\frac{\begin{array}{c}(\mathbf{sc}\; l\; \_\; \_\; \_) \notin h \\ (\mathbf{fwd}\; f\; p\; k \mapsto c_f) \in h \qquad g \;=\; \lambda(p',\, k')\,.\, \mathbf{sc}\; l\; v\; (y.\, p'\; y)\; (z.\, k'\; z)\end{array}}{h \star \mathbf{sc}\; l\; v\; (y.\, c_1)\; (z.\, c_2) \rightsquigarrow c_f\,[\,(\lambda\, y.\, h \star c_1) \,/\, p,\; (\lambda z.\, h \star c_2) \,/\, k,\; g \,/\, f]}\ \text{E-FwdSc}$$

Tabel 3.2: Operationele semantiek van λ_{sc}

Computaties aan elkaar rijgen

Om computaties aan elkaar te rijgen van de vorm $\mathbf{do}\; x \leftarrow c_1\;;\; c_2$ 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 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 (**return** v, **op** ..., **sc** ...). 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 clausule van de handler opgeroepen, $\rightsquigarrow c_r\,[\,v\,/\,x\,]$ (**E-HandRet**). Komt een handler een algebraïsch effect tegen, $h \star \mathbf{op}\, l\, v\, (y.c_1)$, dan wordt onderscheid gemaakt tussen het geval waar de handler een clausule heeft voor de algebraïsche operatie of niet. Is dat zo, $(\mathbf{op}\, l\, x\, k \mapsto c) \in h$, dan is de volgende stap geldig $\rightsquigarrow c[v/x, (\lambda y.h \star c_1)/k]$ (**E-HandOp**). Is dat niet zo, $(\mathbf{op}\, l\, \_\,\_) \notin h$, dan wordt de volgende stap gemaakt $\rightsquigarrow \mathbf{op}\, l\, v\, (y.\, h \star c_1)$ (**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 $(\mathbf{sc}\, l\, \_\,\_\,\_) \notin h \qquad (\mathbf{fwd}\, f\, p\, k \mapsto c_f) \in h \qquad g = \lambda(p',\, k')\, .\, \mathbf{sc}\, l\, v\, (y.\, p'\, y)\, (z.\, k'\, z)$ 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 gebruikt 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 [4]. **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 clausule dat in een handler kan voorkomen, inclusief de lege handler, namelijk **T-Empty** voor de lege handler, **T-OprOp** voor de algebraïsche

$\boxed{\Gamma \vdash v \,:\, \sigma}$ Waarde typering

$$\frac{(x \,:\, \sigma) \in \Gamma}{\Gamma \vdash x \,:\, \sigma}\ \text{T-Var} \qquad \frac{}{\Gamma \vdash (\,) \,:\, (\,)}\ \text{T-Unit}$$

$$\frac{\Gamma \vdash v_1 \,:\, A \qquad \Gamma \vdash v_2 \,:\, B}{\Gamma \vdash (v_1,\, v_2) \,:\, (A,\, B)}\ \text{T-Pair}$$

$$\frac{\Gamma,\, x \,:\, A \vdash c \,:\, \underline{C}}{\Gamma \vdash \lambda\, x.\, c \,:\, A \to \underline{C}}\ \text{T-Abs} \qquad \frac{\Gamma \vdash v \,:\, A \qquad A \equiv B}{\Gamma \vdash v \,:\, B}\ \text{T-Eqv}$$

$$\frac{\Gamma \vdash A \,:\, * \qquad \Gamma \vdash v \,:\, \forall\, \alpha.\, \sigma}{\Gamma \vdash v \,:\, \sigma\,[\,A \,/\, \alpha\,]}\ \text{T-Inst} \qquad \frac{\Gamma,\, \alpha \vdash v \,:\, \sigma \qquad \alpha \notin \Gamma}{\Gamma \vdash v \,:\, \forall\, \mu.\, \sigma}\ \text{T-Gen}$$

$$\frac{\Gamma \vdash v \,:\, \forall\, \mu.\, \sigma}{\Gamma \vdash b \,:\, \sigma\,[\,E \,/\, \mu\,]}\ \text{T-InstEff} \qquad \frac{\Gamma,\, \mu \vdash v \,:\, \sigma \qquad \mu \notin \Gamma}{\Gamma \vdash v \,:\, \forall \mu.\, \sigma}\ \text{T-GenEff}$$

Tabel 3.3: Waarde typering van λ_{sc}

effecten, **T-OprSc** voor de scoped effecten, **T-Fwd** voor de forwarding clausules en **T-Handler** voor de handler als verzameling van deze clausules.

3.1.4 Metatheorie

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

3.1.5 Observaties over de calculus

De grootste veranderingen van de λ_{sc}-calculus tegenover een calculus voor algebraïsche effecten zijn enerzijds de introductie van de **sc** clausule voor scoped effecten en de syntaxis en semantiek hierrond, inclusief de nood voor niet-generieke forwarding doormiddel van de **fwd** clausule. Anderzijds is er de introductie van let-polymorfisme door het **let** sleutelwoord. Het **let** sleutelwoord is nodig om polymorfe handlers te introduceren, welke nodig zijn om scoped effect clausules te typeren, specifiek de scoped computatie binnen de clausule.

3.2 λ^p : Parallelle algebraïsche effect handlers

Deze sectie presenteert een calculus voor de parallelle behandeling van algebraïsche effecten. De calculus hieronder gepresenteerd is een lichte aangepaste versie van de calculus beschreven in "Parallel Algebraic Effect Handlers"[17], aangepast om dichter

$\boxed{\Gamma \vdash c \,:\, \underline{C}}$ Computatie typering

$$\frac{\Gamma \vdash v_1 \,:\, A \to \underline{C} \qquad \Gamma \vdash v_2 \,:\, A}{\Gamma \vdash v_1\, v_2 \,:\, \underline{C}} \text{ T-App}$$

$$\frac{\Gamma \vdash c_1 \,:\, A!\langle E\rangle \qquad \Gamma,\, x \,:\, A \vdash c_2 \,:\, B!\langle E\rangle}{\Gamma \vdash \mathbf{do}\, x \leftarrow c_1; c_2 \,:\, B!\langle E\rangle} \text{ T-Do}$$

$$\frac{\Gamma \vdash c \,:\, \underline{C} \qquad \underline{C} \equiv \underline{D}}{\Gamma \vdash c \,:\, \underline{D}} \text{ T-EqC}$$

$$\frac{\Gamma \vdash v \,:\, \sigma \qquad \Gamma,\, x \,:\, \sigma \vdash c \,:\, \underline{C}}{\Gamma \vdash \mathbf{let}\, x = v\, \mathbf{in}\, c \,:\, \underline{C}} \text{ T-Let}$$

$$\frac{\Gamma \vdash v \,:\, A}{\Gamma \vdash \mathbf{return}\, v \,:\, A!\langle E\rangle} \text{ T-Ret}$$

$$\frac{\Gamma \vdash v \,:\, \forall\, \alpha.\, \alpha!\langle E\rangle \Rightarrow M\, \alpha!\langle F\rangle \qquad \Gamma \vdash c \,:\, A!\langle E\rangle}{\Gamma \vdash v \star c \,:\, M\, A!\langle F\rangle} \text{ T-Hand}$$

$$\frac{(l \,:\, A_l \twoheadrightarrow B_l) \in \Sigma \qquad \Gamma \vdash v \,:\, A_l \qquad \Gamma,\, y \,:\, B_l \vdash c \,:\, A!\langle l; E\rangle}{\Gamma \vdash \mathbf{op}\, l\, v\, (y.\, c) \,:\, A!\langle l; E\rangle} \text{ T-Op}$$

$$\frac{\begin{array}{c}(l \,:\, A_l \twoheadrightarrow B_l) \in \Sigma \qquad \Gamma \vdash v \,:\, A_l \\ \Gamma,\, y \,:\, B_l \vdash c_1 \,:\, B!\langle l; E\rangle \qquad \Gamma,\, z \,:\, B \vdash c_2 \,:\, A!\langle l; E\rangle\end{array}}{\Gamma \vdash \mathbf{sc}\, l\, v\, (y.\, c_1)\, (z.\, c_2) \,:\, A!\langle l; E\rangle} \text{ T-Sc}$$

Tabel 3.4: Computatie typering van λ_{sc}

aan te leunen bij de vorm van calculi gepresenteerd in deze thesis. De calculus 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 $\lambda\, x.\, c$, een lijst van waarden $\langle v_0, ..., v_n\rangle$ of **perform** *op* die een algebraïsche effect uitvoert. Voor waarden wordt f gebruikt voor lambda-functies, n voor literals. 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 parallelle behandeling. Computaties zijn ofwel een waarde v, een applicatie $c\, c$, een for constructie **for** $x\, n.c$ om een lijst te maken van lengte n of een behandel frame **handle** $h\, c$.

$\boxed{\Gamma \vdash \mathit{oprs} \,:\, \underline{C}}$ $\boxed{\Gamma \vdash \mathbf{fwd}\, f\, p\, k \mapsto c \,:\, \underline{C}}$ $\boxed{\Gamma \vdash h \,:\, \underline{C} \Rightarrow \underline{D}}$ Handler typering

$$\frac{}{\Gamma \vdash . \,:\, \underline{C}}\ \text{T-Empty}$$

$$\frac{\Gamma \vdash \mathit{oprs} \,:\, \underline{C} \qquad (l \,:\, A_l \twoheadrightarrow B_l) \in \Sigma \qquad \Gamma,\, x \,:\, A_l,\, k \,:\, B_l \to \underline{C} \vdash c \,:\, \underline{C}}{\Gamma \vdash \mathbf{op}\, l\, x\, k \mapsto c,\, \mathit{oprs} \,:\, \underline{C}}\ \text{T-OprOp}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{oprs} \,:\, M\, A!\langle E\rangle \qquad (l \,:\, A_l \twoheadrightarrow b_l) \in \Sigma) \\ \Gamma,\, \beta,\, x \,:\, A_l,\, p \,:\, B_l \to M\, \beta!\langle E\rangle,\, k \,:\, \beta \to M\, A!\langle E\rangle \vdash c \,:\, M\, A!\langle E\rangle\end{array}}{\Gamma \vdash \mathbf{sc}\, l\, v\, p\, k \mapsto c,\, \mathit{oprs} \,:\, M\, A!\langle E\rangle}\ \text{T-OprSc}$$

$$\frac{\begin{array}{c}A_p = \alpha \to M\, \beta!\langle E\rangle \qquad A_k = \beta \to M\, A!\langle E\rangle \qquad A'_k = M\, \beta \to M\, A!\langle E\rangle \\ \Gamma,\, \alpha,\, \beta,\, p \,:\, A_p,\, k \,:\, A_k,\, f \,:\, (A_p,\, A'_k) \to M\, A!\langle E\rangle \vdash c_f \,:\, M\, A!\langle E\rangle\end{array}}{\Gamma \vdash \mathbf{fwd}\, f\, p\, k \mapsto c_f \,:\, M\, A!\langle E\rangle}\ \text{T-Fwd}$$

$$\frac{\begin{array}{c}\langle E\rangle = \langle \mathit{labels}\,(\mathit{oprs});\, F\rangle \qquad \Gamma,\, \alpha \vdash \mathbf{return} x \mapsto c_r \,:\, M\, \alpha!\langle F\rangle \\ \Gamma,\, \alpha \vdash \mathit{oprs} \,:\, M\, \alpha!\langle F\rangle \qquad \Gamma,\, \alpha \vdash \mathbf{fwd}\, f\, p\, k \mapsto c_f \,:\, M\, \alpha!\langle F\rangle\end{array}}{\Gamma \vdash \mathbf{handler}\,\{\,\mathbf{return},\, \mathit{oprs},\, \mathbf{fwd}\,\} \,:\, \forall\, \alpha.\, \alpha!\langle E\rangle \Rightarrow M\, \alpha!\langle F\rangle}\ \text{T-Handler}$$

Tabel 3.5: Handler typering van λ_{sc}

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

Tabel 3.6: Syntaxis voor λ^p

$\boxed{c \rightsquigarrow c'}$ Reductie van computaties

$$\frac{}{(\lambda\, x.\, c)\; v \rightsquigarrow c\,[\,v\,/\,x\,]} \text{ E-App}$$

$$\frac{}{\langle v_0,\, ...,\, v_n\rangle\; i \rightsquigarrow v_i} \text{ E-Index}$$

$$\frac{(\mathbf{return} \mapsto f_r) \in h}{\mathbf{handle}\; h\; v \rightsquigarrow f_r\; v} \text{ E-Return}$$

$$\frac{\begin{array}{c}\mathbf{op} \notin bop(E) \wedge (\mathbf{op} \mapsto f_p) \in h \\ where\; k = \lambda\, x.\, \mathbf{handle}\; h\; E[x]\end{array}}{\mathbf{handle}\; h\; E[\mathbf{perform}\; op\; v] \rightsquigarrow f_p\; v\; k} \text{ E-Perform}$$

$$\frac{\begin{array}{c}(\mathbf{traverse} \mapsto f_t) \in h \\ where\; l\; =\; \mathbf{for}\; x\; :\; n.\, \mathbf{handle}\; h\; c \\ k\; =\; \lambda\, xs.\, \mathbf{handle}\; h\; F[xs]\end{array}}{\mathbf{handle}\; h\; F[\mathbf{for}\; x\; :\; n.\, c] \rightsquigarrow f_t\; n\; l\; k} \text{ E-Traverse}$$

$$\frac{c \rightsquigarrow c'}{E[c] \rightsquigarrow E[c']} \text{ E-Step}$$

$$\frac{\forall\, 0\; \leq\; i < n.\, c[x\; ::=\; i] \mapsto v_i}{F[\mathbf{for}\; x\; :\; n.\, c] \mapsto F[\langle v_0,\, ...,\, v_{n-1}\rangle]} \text{ E-Parallel}$$

TABEL 3.7: Operationele semantiek van λ^p

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.

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 clausule $(\mathbf{return} \mapsto f_r) \in h$ in de handler toe wanneer de handler een waarde tegenkomt $\mathbf{handle}\; h\; v \rightsquigarrow f_r\; v$. **E-Perform** past een algebraïsche operatie clausule toe wanneer deze in de handler zit $(\mathbf{op} \mapsto f_p) \in h$ en de handler een effect tegenkomt: $\mathbf{handle}\; h\; E[\mathbf{perform}\; op\; v] \rightsquigarrow f_p\; v\; k$. **E-Parallel** is de regel waarin een **for**-constructie in parallel naar een lijst van waarden wordt geëvalueerd, de computatie c wordt hierbij in parallel toegepast voor elke x in n. De meest essentiële regel voor de λ^p-calculus is **E-Traverse**. Deze regel behandelt de

for-constructie die geïntroduceerd wordt door de paper. In essentie vervangt deze regel een **for**-constructie in het programma door een handler-specifieke computatie f_t die geen, een of meerdere nieuwe **for**-constructies introduceert. Essentieel wordt bij deze behandeling de handler doorgeschoven in de continuatie en de computatie die in parallel wordt behandeld.

3.2.3 Observaties over de calculus

De λ^p-calculus modelleert geen type- en effect-systeem. In de semantiek definieert deze calculus 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. Doordat elke handler in de **E-Traverse** clausule nieuwe **for** constructies kan introduceren is het nodig om als buitenste handler het programma te evalueren met een handler die de resterende pure **for**-constructies puur behandelt.

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^p_{sc} die sequentiële en parallelle behandeling van algebraïsche en scoped effecten ondersteunt.

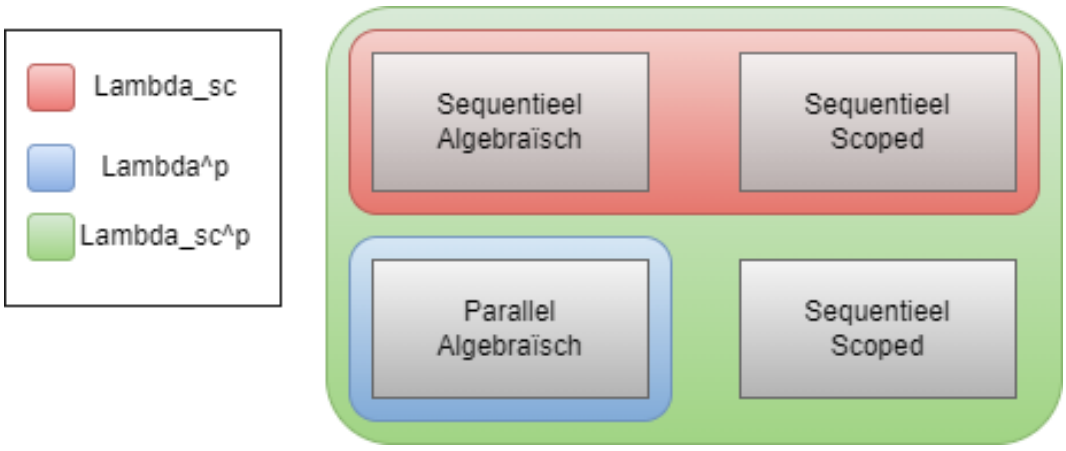
Hoofdstuk 4

Motivatie en Uitdagingen

Dit hoofdstuk behandelt de motivatie voor dit onderwerp en de uitdagingen die vermeldenswaardig 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 [4] ondersteunt sequentiële uitvoering van algebraïsche en scoped effecten. De λ^p-calculus ondersteunt parallelle uitvoering van algebraïsche effecten. Het doel van de λ^p_{sc}-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^{p}_{sc}-calculus voor de semantiek van uitvoeringswijze. Concreet zal elke handler een clausule uitwerken die bepaalt hoe een lijst van computatie met de effecten die de handler behandelt, behandeld worden.

4.1.2 Voorbeeld

$$\begin{aligned}\textbf{for}\ (1:2:3:4:5:[\,])\ (x.\ \textbf{if}\ x \equiv 2\\ &\textbf{then op}\ \mathit{throw}\ ''\mathit{error}''\ (x.\ \textbf{return}\ x);\\ &\textbf{else op}\ \mathit{accum}\ x\ (x.\ \textbf{return}\ x))\ (x.\ \textbf{return}\ x)\end{aligned} \tag{4.1}$$

Het voorbeeld programma in Eq. 4.1 is een programma dat een accumulatie doet in een string waarbij een functie mapt over een lijst met getallen van 1 tot 5. Dit is een voorbeeld uit [17] met herwerkte syntaxis. 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 is dan "1". Bij parallelle afhandeling gaat de exception bij input 2 geen effect kunnen hebben op de andere accumulaties die parallel gebeuren en zal het resultaat "1345"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^p_{sc} 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 clausules 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:

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

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^p_{sc}-calculus.

5.1.1 Lijst van waarden

Een lijst van waarden is zelf een waarde. 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** $x \mapsto c_r$	return clausule
		, $oprs$	effect clausules
		, **fwd** $f\ p\ k \mapsto c_f$	forwarding clausule
		, **for** $lstv\ p\ k \mapsto c$}	for clausule
lijst waarden $lstv$	::=	$[\ ] \mid v : lstv$	lijst van waarden
effect clausules $oprs$	::=	.	
	|	**op** $l\ x\ k \mapsto c,\ oprs$	algebraïsch effect
	|	**sc** $l\ x\ p\ k \mapsto c,\ oprs$	scoped effect
computaties c	::=	**return** v	return waarde
	|	**op** $l\ v\ (y.\ c)$	algebraïsch effect
	|	**sc** $l\ v\ (y.\ c_1)\ (z.\ c_2)$	scoped effect
	|	**for** $v\ (y.\ c_1)\ (z.\ c_2)$	for effect
	|	$v \star c$	behandeling
	|	**do** $x \leftarrow c_1\ ;\ c_2$	do clausule
	|	$v_1\ v_2$	applicatie
	|	**let** $x = v$ **in** c	let
	|	**let rec** $f\ =\ c_1$ **in** c_2	let rec
	|	**head** v	head lstv
	|	**tail** v	tail lstv
	|	**empty** v	test lege lstv
	|	**head** c	head lstc
	|	**tail** c	tail lstc
	|	**empty** c	test lege lstc
	|	**fst** v	first paar
	|	**snd** v	second paar
	|	**map** f v	map over lstv
	|	$lstc$	lijst computaties
lijst computaties $lstc$	::=	$[\ ] \mid c\ :\ lstc$	lijst computaties
waarde types $A,\ B,\ M$	::=	$() \mid (A,\ B) \mid A \to \underline{C} \mid \underline{C} \Rightarrow \underline{D}$	
	|	α	type variabele
	|	$\lambda\,\alpha.\ A$	type operator abstractie
	|	$M\ A$	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 $E,\ F$	::=	$. \mid \mu \mid l;\ E$	
kinds K	::=	$* \mid K \to K$	
signatuur contexten Σ	::=	$. \mid \Sigma,\ l\ :\ A \twoheadrightarrow B$	
type contexten Γ	::=	$. \mid \Gamma,\ x\ :\ A \mid \Gamma,\ \mu \mid \Gamma,\ \alpha$	

Tabel 5.1: λ^{p}_{sc} Syntaxis

5.1.2 Lijst van computaties

Een lijst van compuaties 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^{p}_{sc}-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.

$$\begin{aligned} &\textbf{head}\ (5 : 3 : 8 : 4 : [\ ]) \\ &\rightsquigarrow \textbf{return}\ 5 \end{aligned} \tag{5.2}$$

$$\begin{aligned} &\textbf{head}\ ((\textbf{double}\ 5) : (\textbf{double}\ 3) : (\textbf{double}\ 8) : (\textbf{double}\ 4) : [\ ]) \\ &\rightsquigarrow \textbf{return}\ (\textbf{double}\ 5) \end{aligned} \tag{5.3}$$

$$\begin{aligned} &\textbf{tail}\ (5 : 3 : 8 : 4 : [\ ]) \\ &\rightsquigarrow \textbf{return}\ (3 : 8 : 4 : [\ ]) \end{aligned} \tag{5.4}$$

$$\begin{aligned} &\textbf{tail}\ ((\textbf{double}\ 5) : (\textbf{double}\ 3) : (\textbf{double}\ 8) : (\textbf{double}\ 4) : [\ ]) \\ &\rightsquigarrow \textbf{return}\ ((\textbf{double}\ 3) : (\textbf{double}\ 8) : (\textbf{double}\ 4) : [\ ]) \end{aligned} \tag{5.5}$$

$$\begin{aligned} &\textbf{empty}\ (5 : 3 : 8 : 4 : [\ ]) \\ &\rightsquigarrow \textbf{return}\ \mathit{false} \end{aligned} \tag{5.6}$$

$$\begin{aligned} &\textbf{emmpty}\ ((\textbf{double}\ 5) : (\textbf{double}\ 3) : (\textbf{double}\ 8) : (\textbf{double}\ 4) : [\ ]) \\ &\rightsquigarrow \textbf{return}\ \mathit{false} \end{aligned} \tag{5.7}$$

$$\begin{aligned} &\textbf{empty}\ [\ ] \\ &\rightsquigarrow \textbf{return}\ \mathit{true} \end{aligned} \tag{5.8}$$

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.

$$\begin{aligned} &\textbf{map}\ (+1)\ (5:3:8:4:[\,]) \\ &\rightsquigarrow \textbf{return}\ (6:4:9:5:[\,]) \end{aligned} \tag{5.9}$$

5.2 Recursieve let

let rec helpt om recursieve functies te implementeren in de calculus. De syntaxis is gebaseerd op de syntaxis voor **let rec** beschreven in hoofdstuk twee van het boek "Principles of Programming Languages"[10] en les 11 van CS6110 aan de Cornell-universiteit [14]. Voor simpliciteit is gekozen om slechts 1 recursieve functie toe te laten omdat die semantische regels vereenvoudigt en voldoende expressief is om de gewenste voorbeelden uit te werken. Recursieve functies zijn nodig om complexere functies te definiëren zoals bijvoorbeeld een implementatie van foldr:

$$\begin{aligned} &\textbf{let rec}\ foldr\ = l\ op\ mempty. \\ &\qquad \textbf{do}\ n \leftarrow \textbf{empty}\ l; \\ &\qquad \textbf{if}\ n\ \textbf{then return}\ mempty \\ &\qquad \textbf{else} \\ &\qquad\quad \textbf{do}\ h \leftarrow \textbf{head}\ l; \\ &\qquad\quad \textbf{do}\ t \leftarrow \textbf{tail}\ l; \\ &\qquad\quad \textbf{do}\ y \leftarrow \textbf{foldr}\ t\ op\ mempty; \\ &\qquad\quad \textbf{return}\ (op\ h\ y)\ \textbf{in} \\ &\qquad\qquad \textbf{foldr}\ (1:2:5:6:[\,])\ (+)\ [\,]; \end{aligned} \tag{5.10}$$

Dit voorbeeld programma telt de waarden in de lijst (1:2:5:6:[ ]) op tot 14.

5.3 Handler clausules

Deze sectie behandelt de clausules die moeten toegevoegd worden aan de handlers om parallelle behandeling van effecten mogelijk te maken. Het doel is om effecten parallel te behandelen in een lijst, maar enkel dit doen is niet voldoende.

$$\begin{aligned} &((\textbf{effect}\ input_1) : (\textbf{effect}\ input_2) : (\textbf{effect}\ input_3) : [\,]); \\ &\textbf{rest van het programma} \\ &\rightsquigarrow ((output_1) : (output_2) : (output_3) : [\,]); \\ &\textbf{rest van het programma} \\ &\rightsquigarrow ? \end{aligned} \tag{5.11}$$

De lijst van effecten moet linken aan de rest van het programma. Hiervoor is een nieuwe clausule nodig. De **for** clausule 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. De **for** clausule heeft in tegenstelling tot de **op** of **sc** clausule geen label omdat deze clausule geen specifiek effect voorsteld maar een generieke **for-each** constructie over een lijst.

$$\textbf{for} \quad (input_1 : input_2 : input_3 : [\,]) \quad (input.\ \textbf{effect}\ input) \quad (result.\ \textbf{rest}) \tag{5.12}$$

Een handler heeft de vrije keuze over hoe een **for** constructie te behandelen. De standaard benadering voor handlers die geen specifieke **for** clausule implementeren is om de **for** clausule terug te geven, waarbij de handler binnen de mapping-functie en de resumptie geschoven worden, zie **E-FwdFor** in Tabel 6.1. Het is raadzaam dat elk programma met **for** constructies ls buitenste handler een **hPure** handler heeft om eventuele overgebleven pure **for** constructies af te handelen.

$$\begin{aligned}
&\textbf{hPure} = \textbf{handler}\\
&\{\\
&\quad \ldots\\
&\quad \textbf{for}\ lst\ p\ k \rightarrow \textbf{do}\ res \leftarrow map\ p\ lst\\
&\qquad\qquad\qquad k\ res\\
&\}
\end{aligned} \tag{5.13}$$

Hoofdstuk 6

Operationele Semantiek

Tabellen 6.1 en 6.2 toont de operationele semantiek voor de λ^{p}_{sc}-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. De bestaande regels uit de $\lambda_s c$-calculus [4] blijven onveranderd.

6.1 Letrec: Recursieve functies

De **let rec** constructie biedt de mogelijk om recursieve functies te definiëren in de calculus. De clausule bestaat uit het **let rec** sleutelwoord gevolgd door een variabelenaam 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 variable f kan voorkomen. Deze clausule wordt gereduceerd naar c_2 met f in c_2 vervangen door c_1 met f vervangen door **let rec** $= c_1$ **in** f. **E-Letrec** is een aangepaste versie van de regel beschreven in hoofdstuk twee van het boek "Principles of Programming Languages"[10].

6.2 For clausule

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

$$\frac{}{(\lambda x.\, c)\ v \rightsquigarrow c\,[\,v\,/\,x\,]}\ \text{E-AppAbs} \qquad \frac{}{\textbf{let}\ x\ =\ v\ \textbf{in}\ c \rightsquigarrow c\,[\,v\,/\,x\,]}\ \text{E-Let}$$

$$\frac{}{\textbf{let rec}\ f\ =\ c_1\ \textbf{in}\ c_2 \rightsquigarrow c_2\,[c_1\,[(\textbf{let rec}\ f\ =\ c_1\ \textbf{in}\ f)\,/\,f]\,/\,f]}\ \text{E-Letrec}$$

$$\frac{c_1 \rightsquigarrow c_1'}{\textbf{do}\ x \leftarrow c_1\ ;\ c_2 \rightsquigarrow \textbf{do}\ x \rightsquigarrow c_1'\ ;\ c_2}\ \text{E-Do} \qquad \frac{}{\textbf{do}\ x \leftarrow \textbf{return}\ v\ ;\ c_2 \rightsquigarrow c_2\,[\,v\,/\,x\,]}\ \text{E-DoRet}$$

$$\frac{}{\textbf{do}\ x \leftarrow \textbf{op}\ l\ v\ (y.\, c_1)\ ;\ c_2 \rightsquigarrow \textbf{op}\ l\ v\ (y.\, \textbf{do}\ x \leftarrow c_1\ ;\ c_2)}\ \text{E-DoOp}$$

$$\frac{}{\textbf{do}\ x \leftarrow \textbf{sc}\ l\ v\ (y.\, c_1)\ (z.\, c_2)\ ;\ c_3 \rightsquigarrow \textbf{sc}\ l\ v\ (y.\, c_1)\ (z.\, \textbf{do}\ x \leftarrow c_2\ ;\ c_3)}\ \text{E-DoSc}$$

$$\frac{}{\textbf{do}\ x \leftarrow \textbf{for}\ lstv\ (y.\, c_1)(z.\, c2);\ c_3 \rightsquigarrow \textbf{for}\ lstv\ (y.\, c_1)\ (z.\, \textbf{do}\ x \leftarrow c_2;\ c_3)}\ \text{E-DoFor}$$

$$\frac{c \rightsquigarrow c'}{h \star c \rightsquigarrow h \star c'}\ \text{E-Hand} \qquad \frac{(\textbf{return}\ x \mapsto c_r) \in h}{h \star \textbf{return}\ v \rightsquigarrow c_r\,[\,v\,/\,x\,]}\ \text{E-HandRet}$$

$$\frac{(\textbf{op}\ l\ x\ k \mapsto c) \in h}{h \star \textbf{op}\ l\ v\ (y.\, c_1) \rightsquigarrow c\,[\,v\,/\,x,\ (\lambda y.\, h \star c_1)\,/\,k]}\ \text{E-HandOp}$$

$$\frac{(\textbf{op}\ l\ \_\ \_) \notin h}{h \star \textbf{op}\ l\ v\ (y.\, c_1) \rightsquigarrow \textbf{op}\ l\ v\ (y.\, h \star c_1)}\ \text{E-FwdOp}$$

$$\frac{(\textbf{sc}\ l\ x\ p\ k \mapsto c) \in h}{h \star \textbf{sc}\ l\ v\ (y.\, c_1)\ (z.\, c_2) \rightsquigarrow c\,[\,v\,/\,x,\ (\lambda\, y.\, h \star\ c_1)\,/\,p, (\lambda z.\, h \star c_2)\,/\,k\,]}\ \text{E-HandSc}$$

$$\frac{\begin{array}{c}(\textbf{sc}\ l\ \_\ \_\ \_) \notin h \\ (\textbf{fwd}\ f\ p\ k \mapsto c_f) \in h \qquad g\ =\ \lambda(p',\ k')\ .\ \textbf{sc}\ l\ v\ (y.\, p'\ y)\ (z.\, k'\ z)\end{array}}{h \star \textbf{sc}\ l\ v\ (y.\, c_1)\ (z.\, c_2) \rightsquigarrow c_f\,[\,(\lambda\, y.\, h \star c_1)\,/\,p,\ (\lambda z.\, h \star c_2)\,/\,k,\ g\,/\,f]}\ \text{E-FwdSc}$$

$$\frac{(\textbf{for}\ lstv_1\ p\ k \mapsto c_{for}) \in h}{h \star \textbf{for}\ lstv_2\ (y.\, c_1)\ (z.\, c_2) \rightsquigarrow c_{for}[lstv_2\,/\,lstv_1,\ (y.\, h \star c_1)\,/\,p, (z.\, h \star c_2)\,/\,k]}\ \text{E-HandFor}$$

$$\frac{(\textbf{for}\ \_\ \_\ \_\ \notin h)}{h \star \textbf{for}\ lstv\ (y.\, c_1)\ (z.\, c_2) \rightsquigarrow \textbf{for}\ lstv\ (y.\, h \star c_1)\ (z.\, h \star c_2)}\ \text{E-FwdFor}$$

Tabel 6.1: Operationele semantiek van λ^{p}_{sc}

$\boxed{c \rightsquigarrow c'}$ Computatie reductie

$$\frac{}{\textbf{map}\ f\ (v_1 : v_2 : \ \ldots\ : v_n : [\ ]) \rightsquigarrow ((f\ v_1) : (f\ v_2) : \ \ldots\ : (f\ v_n) : [\ ])}\ \text{E-Map}$$

$$\frac{1 \leq i \leq n \qquad c_i \rightsquigarrow c_i'}{(c_1 : \ldots : c_i : \ldots : c_n : [\ ]) \rightsquigarrow (c_1 : \ldots : c_i' : \ldots : c_n : [\ ])}\ \text{E-ParList}$$

$$\frac{}{((\textbf{return}\ v_1) : (\textbf{return}\ v_2) : \ldots : (\textbf{return}\ v_n) : [\ ]) \rightsquigarrow \textbf{return}\ (v_1 : v_2 : \ldots : v_n : [\ ])}\ \text{E-ListRet}$$

$$\frac{}{\textbf{head}\ (v_1 : lstv) \rightsquigarrow \textbf{return}\ v_1}\ \text{E-HeadLstv} \qquad \frac{}{\textbf{head}\ (c_1 : lstc) \rightsquigarrow \textbf{return}\ c_1}\ \text{E-HeadLstc}$$

$$\frac{}{\textbf{tail}\ (v_1 : lstv) \rightsquigarrow \textbf{return}\ lstv}\ \text{E-TailLstv} \qquad \frac{}{\textbf{tail}\ (c_1 : lstc) \rightsquigarrow \textbf{return}\ lstc}\ \text{E-TailLstc}$$

$$\frac{}{\textbf{empty}\ [\ ] \rightsquigarrow \textbf{return}\ true}\ \text{E-EmptyTrue}$$

$$\frac{}{\textbf{empty}\ (v : lstv) \rightsquigarrow \textbf{return}\ false}\ \text{E-EmptyLstvFalse}$$

$$\frac{}{\textbf{empty}\ (c : lstc) \rightsquigarrow \textbf{return}\ false}\ \text{E-EmptyLstcFalse}$$

$$\frac{}{\textbf{fst}\ (v_1,\ v_2) \rightsquigarrow \textbf{return}\ v_1}\ \text{E-First} \qquad \frac{}{\textbf{snd}\ (v_1,\ v_2) \rightsquigarrow \textbf{return}\ v_2}\ \text{E-Second}$$

$$\frac{c_1 \rightsquigarrow c_1'}{\textbf{if}\ c_1\ \textbf{then}\ c_2\ \textbf{else}\ c_3 \rightsquigarrow \textbf{if}\ c_1'\ \textbf{then}\ c_2\ \textbf{else}\ c_3}\ \text{E-If}$$

$$\frac{}{\textbf{if}\ true\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \rightsquigarrow c_1}\ \text{E-IfTrue}$$

$$\frac{}{\textbf{if}\ false\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \rightsquigarrow c_2}\ \text{E-IfFalse}$$

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-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. Elke handler kan de clausule implementeren of laten forwarden (**E-FwdFor**, hoofdstuk 6.2.3). Aangezien deze behandeling nieuwe **for** constructies kan introduceren is het raadzaam om vooraan het programma een handler te implementeren die overgebleven pure **for** constructies afhandelt. De implementatie van deze clausule specifieert of de handler de constructie in parallel of sequentieel afhandelt.

6.2.3 E-FwdFor

E-FwdFor behandelt de forwarding voor het geval dat de **for** clausule niet door de handler geïmplementeerd wordt. De functie van de forwarding is tweevoudig. Deze regel schuift de handler binnen de computatie die te mappen is en de resumptie.

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** clausule 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 stelt dat een element in de lijst dat een stap kan maken, deze stap zet. Merk op dat dit een niet-deterministisch regel is. De voorwaarde om nog een stap te kunnen zetten is dat de computatie niet in normaalvorm is, met andere woorden niet in de vorm:

$$\textbf{return}\ \_ \mid \textbf{op}\ \_\ \_\ \_ \mid \textbf{sc}\ \_\ \_\ \_\ \_ \mid \textbf{for}\ \_\ \_\ \_\ \_ \tag{6.1}$$

6.3.3 E-ListRet

Als een lijst van computaties gereduceerd kan worden naar een lijst van **return** clausules van waarden, dan kan de lijst vervangen worden door een **return** clausule van de lijst van waarden. Deze clausule 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** clausule.

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] D. Ahman and M. Pretnar. Asynchronous effects. *CoRR*, abs/2003.02110, 2020.

[2] A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84:108–123, 1 2015.

[3] E. Bingham, J. P. Chen, M. Jankowiak, F. Obermeyer, N. Pradhan, T. Karaletsos, R. Singh, P. A. Szerlip, P. Horsfall, and N. D. Goodman. Pyro: Deep universal probabilistic programming. *J. Mach. Learn. Res.*, 20:28:1–28:6, 2019.

[4] R. Bosman, B. van den Berg, W. Tang, and T. Schrijvers. A calculus for scoped effects& handlers. *under review*, 2022.

[5] J. I. Brachthäuser, P. Schuster, and K. Ostermann. Effekt: Lightweight effect polymorphism for handlers (technical report), 2020.

[6] 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.

[7] D. Leijen. Type directed compilation of row-typed algebraic effects. *SIGPLAN Not.*, 52:486–499, 1 2017.

[8] 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.

[9] E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991. Selections from 1989 IEEE Symposium on Logic in Computer Science.

[10] Z. Palmer, M. Grant, and S. Smith. *Principles of Programming Languages*. John Hopkins University, 2009.

[11] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[12] 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.

[13] M. Pretnar. An introduction to algebraic effects and handlers. invited tutorial paper. *Electronic Notes in Theoretical Computer Science*, 319:19–35, 12 2015.

[14] A. Sampson. Cs 6110: Advanced programming languages spring 2018 lecture 11. https://www.cs.cornell.edu/courses/cs6110/2018sp/lectures/lec11.pdf, 2018. Accessed on: 03/01/2023.

[15] B. van den Berg, T. Schrijvers, C. B. Poulsen, and N. Wu. Latent effects for reusable language components: Extended version. *CoRR*, abs/2108.11155, 2021.

[16] N. Wu, T. Schrijvers, and R. Hinze. Effect handlers in scope. *SIGPLAN Not.*, 49:1–12, 9 2014.

[17] N. Xie, D. D. Johnson, D. Maclaurin, and A. Paszke. Parallel algebraic effect handlers. *CoRR*, abs/2110.07493, 2021.

[18] 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.