KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Abstract Syntax Tree Code Generator for Haskell-based Com- and Transpilers

Bram Belpaire

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

Promotor:
Prof. dr. ir. Tom Schrijvers

Assessoren:
Prof. dr. ir. Bart De Decker
Prof. dr. ir. Gerda Janssens

Academiejaar 2018 – 2019

Voorwoord

Ik wens dit voorwoord te gebruiken om natuurlijk mijn familie en vrienden die mij door alle jaren heen gesteund hebben, te bedanken. In het bijzonder bedank ik mijn begeleider en promotor, prof. dr. ir. Tom Schrijvers.

Bram Belpaire

Inhoudsopgave

Voorwoord i

Samenvatting iv

Lijst van figuren en tabellen v

1 Inleiding 1
- 1.1 Onderzoeksvraag . . . 2
- 1.2 Aanpak . . . 2
- 1.3 Resultaten . . . 3
- 1.4 Structuur . . . 3

2 Achtergrond 5
- 2.1 Simply Typed Lambda Calculus (STLC) . . . 5
- 2.2 Substitutie . . . 7
- 2.3 De Bruijn Indices . . . 9
- 2.4 Haskell . . . 11
- 2.5 Abstract Syntax . . . 12
- 2.6 Samenvatting . . . 12

3 Probleem 13
- 3.1 Inleiding . . . 13
- 3.2 Implementatie van Shift en Substitutie . . . 14
- 3.3 Benodigdheden . . . 17
- 3.4 Samenvatting . . . 17

4 Oplossing 19
- 4.1 Overzicht . . . 19
- 4.2 Aut'Bound: basisversie . . . 20
- 4.3 Aut'Boundspecificatietaal : Nieuwe Concepten . . . 22
- 4.4 Parser . . . 26
- 4.5 Verificaties en Foutmeldingen . . . 26
- 4.6 Gegenereerde Code . . . 27
- 4.7 Voorbeeld gegenereerde code . . . 28
- 4.8 Gebruik bij Evaluatie en Typering . . . 29
- 4.9 Samenvatting . . . 30

5 Evaluatie 39
- 5.1 System F met Coercions . . . 39

5.2 Fi+ . . . 43
5.3 Explicit Effects . . . 45
5.4 Testmethode . . . 47
5.5 Codegeneratie . . . 47
5.6 Experimenten . . . 48
5.7 Samenvatting . . . 49

6 Gerelateerd Werk **55**
6.1 Unbound . . . 55
6.2 Ott . . . 55
6.3 Inbound . . . 55
6.4 Samenvatting . . . 56

7 Conclusie **57**
7.1 Verder werk . . . 57

A Poster **61**

B Gegenereerde code voor lambda calculus **63**

C Complete specificatie van exeff **67**

Bibliografie **73**

Samenvatting

Bij het schrijven van calculi wordt vrij veel tijd verloren aan het schrijven van de code rond variabelebinding. Er zijn al tools hierrond maar de meeste gebruiken geen naamloze representatie. De tool voorzien in dit werk doet dit wel. We zien dat in dit onderzoeksdomein al verschillende manieren zijn gevonden voor het presenteren van de info nodig om de code te laten genereren. Wij werken verder gebaseerd op een van deze implementaties namelijk Inbound [14]. We nemen echter een andere aanpak bij het produceren van de effectieve code. Ook zorgen we voor meer mogelijkheden dan er waren in de originele taal. Deze aanpak werd geschreven in de functionele taal Haskell en is te vinden op https://github.com/Belpaire/ASTTool.

Lijst van figuren en tabellen

Lijst van figuren

3.1 Datatypes 14
3.2 Vergelijking shift en substitutie 15
3.3 Gemeenschappelijke structuur 15
3.4 Implementatie substitutie applicatie 16
3.5 Implementatie evaluatie 16

4.1 Overzicht van de werking van de tool 19
4.2 Originele specificatietaal 20
4.5 Implementatie aan de hand van lijsten 23
4.8 Specificatie van System F 28
4.11 Code voor tuples en patterns 28
4.12 System F with Base Type 29
4.3 STLC met tuples 31
4.4 Implementatie met een record 32
4.6 Implementatie met Vectorvoorbeeld 33
4.7 Voorbeeld voor Haskellcode 33
4.9 Specificatie Lambda Calculus met Booleans 34
4.10 Deel van de code gegenereerd voor lambda calculus met booleans 35
4.13 Substitutiegebruik bij Evaluatie 36
4.14 Substitutiegebruik bij Typering 37

5.1 System F met coercions 39
5.2 Invoer van System F Co 40
5.3 code in system F met coercions 41
5.4 Fi+ 43
5.5 Specificatie van Fi+ in de tool 44
5.6 ExEff 45
5.7 Codevoorbeelden uit ExEff 46
5.8 Case Studies 48
5.9 Meting voor plus 50
5.10 Meting voor shift (grote variabelen) 50
5.11 Meting voor shift (kleine variabelen) 51

5.12 Meting voor substitutie (grotere term) 51
5.13 Meting voor substitutie (kleinere shift) 52
5.14 Meting voor substitutie (klein aantal binders) 52
5.15 Meting voor substitutie (middelgroot aantal binders) 53
5.16 Meting voor substitutie (groot aantal binders) 53

A.1 poster zoals ingediend op 26 april 62

Lijst van tabellen

5.1 Aantal lijnen horende bij specificatie, generatie en zelfgeschreven code . 47

Hoofdstuk 1

Inleiding

De meeste praktische programmeertalen worden niet formeel gedefinieerd maar slechts informeel beschreven of zelfs enkel geïmplementeerd. De reden hiervoor is dat praktische talen gewoonweg te veel constructen bevatten en met de huidige technieken is het niet haalbaar om deze in zijn geheel te formuleren.

Desondanks kan het formaliseren van programmeertalen van groot praktisch nut zijn. Vaak worden talen enkel geïmplementeerd wat leidt tot onverwachte eigenschappen. Bijvoorbeeld, type checking in java is turing compleet [13]. Dit was onverwachts geïntroduceerd. Het domein van formele programmeertaaltheorie houdt zich bezig met dit soort problemen te bestuderen en te voorkomen.

Daarom vereenvoudigen we de taal naar een compactere versie door enkel een kern te behouden. Zo'n kern van de taal wordt ook wel calculus genoemd. Zo'n vereenvoudigde taal wordt dan bestudeerd in plaats van de programmeertaal zelf. Een praktisch voorbeeld hiervan is hoe Haskell gereduceerd kan worden tot de Core taal [1]. Een ander voorbeeld is ook Feitherweight Java [9] dat dan gezien kan worden als een minimale kern calculus voor Java.

Het uitwerken van deze calculi bestaat meestal uit twee delen: enerzijds, het wiskundig bewijzen van een aantal eigenschappen, anderzijds uit de ontwikkeling van een prototype-implementatie van de calculus. Zo'n prototype laat toe programma's te schrijven en uit te voeren.

De meeste niet-triviale calculi bevatten constructies voor het introduceren van variabelen. Variabelebinding is een overkoepelende term die slaagt op constructies die variabelen introduceren. Deze termen kunnen variëren van simpele anonieme functies tot ingewikkeldere bindingen zoals case statements. Het probleem dat dit werk behandelt, is het volgende: de code met betrekking tot variabelen is vrij uitgebreid en heeft bovendien weinig eigen inbreng van de programmeur nodig. Deze code omvat operaties zoals het substitueren van variabelen met andere termen. Vaak zijn deze operaties geïmplementeerd met behulp van structurele recursie. Dit maakt het

juist zo vervelend om deze code manueel te schrijven. Dankzij de grote gelijkaardigheid in de stukken code, zou het efficiënter zijn om deze code automatisch te genereren.

Hiertoe presenteren we in dit werk een tool die een definitie van de calculus inleest en code in verband met variabelen genereert. De definitie van de calculus wordt uitgedrukt in een DSL (Domain Specific Language). Deze DSL noemen we vanaf nu Aut'Bound. Onze tool, de Aut'Bound compiler vertaalt Aut'Bound bestanden naar Haskell code die de operaties voor variabelen implementeert.

Er bestaan al tools, zoals Inbound en Ott [14, 17], die ook code in verband met variabelebinding genereren.

De meeste tools voor het werken met variabelen in calculi richten zich echter niet op het genereren van prototype-implementaties, maar eerder op generatie voor theorem provers zoals Coq [5], met het oog op het computergesteund bewijzen van eigenschappen voor de calculi.

Unbound en Inbound [14, 19] zijn tools die het mogelijk maken de code voor variabelebinding niet zelf te hoeven schrijven.

In dit werk ontwerpen we een DSL genaamd Aut'Bound en de tool die deze taal interpreteert, genaamd de Aut'Bound compiler. De Aut'Bound compiler leest een Aut'Bound specificatie in en is specifiek gericht op het genereren van functionaliteit naar de gebruiker toe, op een zo expressieve en gebruiksvriendelijke mogelijke manier.

Dit werk onderscheidt zich van Unbound en Inbound op verschillende manieren. Op vlak van variabelevoorstelling hebben we een andere aanpak dan deze. Ook verschilt dit werk van Unbound vanwege het feit dat Unbound een Haskell library is en deze tool de nodige code genereert op basis van een specificatie.

1.1 Onderzoeksvraag

De onderzoeksvraag luidt als volgt:

> **Een tool te ontwerpen die automatisch code genereert ter ondersteuning van variabelen in prototype-implementaties van calculi. Een bijkomende vereiste is dat de tool meer calculi ondersteunt en gebruiksvriendelijker is dan bestaande tools.**

1.2 Aanpak

We vertrekken van de Inboundspecificatietaal om calculi in te beschrijven. Deze is geschikt omdat ze toelaat complexe bindingmechanismen te schrijven. Inbound is echter wat onhandig om gevorderde taalconcepten in te schrijven. Daarom voegen

we enkele structuren toe aan de specificatietaal om ze gebruiksvriendelijker te maken. Vervolgens testen we onze tool op enkele praktische calculi. Deze case studies zullen ons dan het al dan niet werken van de tool aantonen.

1.3 Resultaten

Het hoofdresultaat van dit werk is de Aut'Bound compiler die de datastructuren en alle benodigdheden voor substitutie en substitutie genereert. We geven een volledige beschrijving van de onderdelen van wat de tool als invoer verwacht, namelijk de verschillende constructies die Aut'Bound ondersteunt en hoe de Aut'Bound compiler die constructies omzet naar concrete code.

We ondersteunen alle constructies van Inbound [14] maar vullen ook aan met eigen constructies voor de gebruiksvriendelijkheid te verhogen.

Overigens evalueren we Aut'Bound door enkele calculi in Aut'Bound te beschrijven en gebruiken we onze Aut'Bound compiler om de functionaliteit te genereren voor de implementatie. We maken dan van deze calculi prototype-implementaties en testen of deze correct evalueren. Op deze manier kunnen we zien of de achterliggende functies correct gegenereerd zijn. Wat impliceert dat onze Aut'Bound compiler op zijn beurt ook correct geïmplementeerd moet zijn

Uiteindelijk is het natuurlijk de bedoeling om toe te laten anderen deze tool te laten gebruiken. Tot dat doel zijn er in dit werk verschillende voorbeelden die telkens bepaalde ideeën uitdrukken. Hiermee kan een gebruiker afleiden hoe men de tool zelf zou gebruiken.

Daarnaast verschilt onze tool voorgesteld in dit werk van de bestaande tools. De tool onderscheidt doordat wij een andere variabelevoorstelling hanteren dan de meeste andere tools. Daarnaast is de tool geschreven in puur Haskell. Andere tools gebruiken soms zaken zoals de Utrecht university attribute grammar compiler [15]. Er is geen exact gelijke tool. De Aut'Bound tool is te vinden op `https://github.com/Belpaire/ASTTool`.

1.4 Structuur

Hoofdstuk 2 beschrijft de reden om calculi te bestuderen en legt enkele basisbegrippen uit.

Hoofdstuk 3 formuleert de probleemstelling en toont hoe calculi gewoonlijk manueel geïmplementeerd worden. Er wordt aangetoond dat we met een beperkte hoeveelheid informatie ook alle nodige code kunnen genereren.

Hoofdstuk 4 beschrijft de Aut'Bound compiler, de tool die wij ontwikkeld heb-

ben om aan de probleemstelling uit Hoofdstuk 3 tegemoet te komen. Gebaseerd op vorig onderzoek in dit domein bouwen we voort aan een manier om de invoer van onze tool te specifiëren door extra mogelijkheden aan de invoer toe te voegen.

Hoofdstuk 5 evalueert de Aut'Bound compiler op verschillende manieren, waaronder op gebied van performantie.

Hoofdstuk 6 beschrijft gerelateerd werk en hoe onze tool zich positioneert binnen het onderzoek in dit domein.

Hoofdstuk 7 concludeert het werk en geeft ideeën voor verdere toekomstige uitbreidingen.

Hoofdstuk 2

Achtergrond

Aangezien implementaties van calculi de reden zijn waarvoor dit werk ontworpen is, leggen we in dit hoofdstuk uit hoe men calculi moet lezen en waarom men ze bestudeert. We beginnen bij de meest eenvoudige calculus, de lambda calculus. Ook tonen we aan waarom we calculi bestuderen. Daarna beschrijven we hier ook enkele beginselen van de programmeertaal Haskel, de taal waarin we code genereren. Ten laatste leggen we het begrip AST (Abstract Syntax Tree) uit. Onze uitleg omtrent abstract syntax, calculi en de Bruijn indices is gebaseerd op de het boek Typing and Programming Languages [9].

2.1 Simply Typed Lambda Calculus (STLC)

De lambda calculus staat aan de basis van de meeste talen die we willen beschrijven. In de pure ongetypeerde lambda calculus heeft men enkel functies, variabelen en functie-applicaties. Meestal wordt de lambda calculus verder uitgebreid met extra structuren zoals tupels of lijsten.

Een uitbreiding die vaak wordt toegepast, is om de lambda calculus met types uit te rusten. De syntax van de uitbreiding heeft twee soorten onderdelen, namelijk Termen en Types. Men voegt een basistype toe en een functietype toe. Een functietype is het type dat een functie heeft die invoer van een type T_1 als argument krijgt en die uitvoer heeft van het type T_2. Een functietype wordt geschreven als $T_1 \to T_2$. De reden voor het basistype toe te voegen is als volgt uit te leggen: zonder basistype zou er geen eindig type te schrijven zijn omdat het functietype recursief naar zichzelf verwijst.

term t ::= x	variable
|λ x : T . t	abstractie
| t t	applicatie
type T ::= B	basistype
| T → T	functietype

2.1.1 Typering in STLC

De context Γ bevat de type-informatie voor vrije variabelen. De relatie $\Gamma \vdash t : T$ duidt aan dat dat term t type T heeft in context Γ. De typeringsrelatie is gedefinieerd aan de hand van de onderstaande typeringsregels.

$$\Gamma ::= \varnothing \mid \Gamma, x : T$$

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\, t_2 : T_2} \qquad \text{[T-App]}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{[T-Var]}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2} \qquad \text{[T-Abs]}$$

De regels hebben de volgende betekenis/

- Bij de T-App regel moet de linkerterm een functietype hebben en de rechterterm moet het invoertype van het functietype hebben.
- De T-Var regel specifieert dat een variabele het type heeft dat gespecifieerd is in de context.
- De T-Abs regel specifieert dan weer dat het type van t_2, met de variabele x toegevoegd aan de context, het resultaattype is van het functieype. Het invoertype is het type zoals geannoteerd bij de geïntroduceerde variabele.

2.1.2 Evaluatie in STLC

De evaluatie van termen wordt uitgedrukt met de relatie t $\to$ t', die zegt dat term in 1 stap kan vereenvoudigd worden tot t'. De v in de evaluatieregels slaagt op het feit dat een term een waarde is. Een waarde is altijd een term die niet meer verder geëvalueerd kan worden. In de pure lambda calculus zijn enkel functies waardes. In andere calculi kunnen er bijkomende waardes zijn.

$$\text{v} ::= \lambda x : T.t$$

$$\frac{t_1 \to t_1'}{t_1\, t_2 \to t_1'\, t_2} \qquad \text{[E-App1]}$$

$$\frac{t_2 \to t_2'}{v_1\, t_2 \to v_1\, t_2'} \qquad \text{[E-App2]}$$

$$(\lambda x.t_1)v_2 \to [x \mapsto v_2]\, t_1 \qquad \text{[E-AppAbs]}$$

De evaluatieregels zijn als volgt:

- De regel E-App1 specifieert dat een applicatie kan geëvalueerd worden als de linkerterm kan geëvalueerd worden.
- De regel E-App2 is gelijkaardig aan E-App1, behalve dat de linkerterm al geëvalueerd moet zijn tot een waarde voordat deze regel kan gebruikt worden.
- Regel E-AppAbs zegt dat als de rechterterm geëvalueerd is tot een waarde, deze kan gesubstitueerd worden in de body van de functie.

Deze regels voor evaluatie volgen de call-by-value semantiek. Ze proberen altijd een term volledig te reduceren tot een waarde. Er bestaan andere mogelijkheden, maar om daarop ingaan zou ons te ver leiden.

Welke eigenschappen kunnen bestudeerd worden uit de lambda calculus? Eigenschappen die meestal bestudeerd worden zijn meestal in verband met typering. Voorbeelden hiervan zijn:

- Heeft een term altijd een unieke typering?
- Zal een term die getypeerd kan worden, in een eindig aantal stappen gereduceerd worden tot een value?

2.2 Substitutie

Laat ons wat dieper ingaan op substitutie, het centrale thema van deze thesis. Hier behandelen we het geval waarbij variabelen met een naam worden voorgesteld. Eerst leggen we nog de volgende twee termen uit.

- Alpha equivalentie: Wanneer men in de lambda calculus termen vergelijkt, zouden namen niet mogen uitmaken. Bijvoorbeeld, de functie $\lambda a.a$ is functioneel identiek aan de functie $\lambda b.b$.

- Capture avoiding substitution: Capture avoiding substitutions slaagt opnieuw op het feit dat de namen van variabelen geen betekenis zouden mogen hebben. Het probleem bestaat erin. Dat, als men een variabele substitueert in een term, deze van betekenis kan veranderen zonder dat we dat verwachten. Een voorbeeld hiervan is dat wanneer men de variabele b substitueert met een variabele a in de term $\lambda a.a\,b$. Deze nieuwe a wordt opeens een gebonden variabel, wat fout zou zijn want de betekenis van de anonieme functie is nu aangepast.

We zullen eerst een naïeve poging doen om substitutie te definiëren.

$[x \mapsto s]x$	$=$ s
$[x \mapsto s]y$	$=$ y if x $\neq$ y
$[x \mapsto s](\lambda y.t_1)$	$= \lambda y.[x \mapsto s]\, t_1$
$[x \mapsto s]t_1\, t_2$	$= ([\text{x} \mapsto s]t_1)\,([x \mapsto s]t_2)$

Alhoewel dit werkt in de meeste gevallen, kunnen we nog altijd problemen vinden. Neem nu het volgende voorbeeld:

$$[x \mapsto y](\lambda x.x) = \lambda x.y \tag{2.1}$$

Dit is natuurlijk ongewenst gedrag. De fout die we maken, is dat we geen rekening houden of dat de variabele gebonden wordt in de term waarin we substitueren. We lossen dit als volgt op:

$[x \mapsto s]x$	$=$ s
$[x \mapsto s]y$	$=$ y if x $\neq$ y
$[x \mapsto s](\lambda y.t_1)$	$=$ λ y. [x$\mapsto$s] t_1 if x$\neq y$
$[x \mapsto s](\lambda y.t_1)$	$=$ λ y. t_1 if x $=$ y
$[x \mapsto s]t_1\, t_2$	$= ([\text{x} \mapsto s]t_1)([x \mapsto s]t_2)$

Echter, deze definitie bevat nog altijd een fout.

$$[x \mapsto z](\lambda z.x) = \lambda z.z \tag{2.2}$$

Hier gebeurt de volgende fout: We hadden capture-avoiding substitution al vermeld en dat is exact wat hier voorkomt. Ook dit valt op te lossen door volgende aanpassing te maken:

$[x \mapsto s]x$	= s
$[x \mapsto s]y$	= y if x $\neq$ y
$[x \mapsto s](\lambda y.t_1)$	= λ y. [x$\mapsto$s] t$_1$if x $\neq y$ and y $\notin FV(s)$
$[x \mapsto s](\lambda y.t_1)$	= λ y. t$_1$ if x $= y$
$[x \mapsto s]t_1\, t_2$	= ([x $\mapsto$ s]t_1)([$x \mapsto s]t_2$)

Helaas is substitutie hier niet een totale functie:

$$[x \mapsto yz]\,(\lambda y.x\,y) \tag{2.3}$$

x is niet hetzelfde als de variabele y, maar y komt wel voor in de substitutieterm. Een oplossing is door te werken met namen die altijd 'vers' zijn. Namelijk onder 'vers' verstaan we het vervangen van namen in lambda-abstracties met namen waarvan we zeker zijn dat ze ongebruikt zijn. We kunnen de term $\lambda y.xy$ bijvoorbeeld transformeren naar $\lambda w.xw$. Hiermee kunnen we de uiteindelijke substitutiefiguur geven.

$[x \mapsto s]x$	= s
$[x \mapsto s]y$	= y if x $\neq$ y
$[x \mapsto s](\lambda y.t_1)$	= λ y. [x$\mapsto$ s]t$_1$ if y $\neq$ x and y $\notin$ FV(s)
$[x \mapsto s]t_1\, t_2$	= ([x $\mapsto$ s]t_1)([$x \mapsto s]t_2$)

Door altijd variabelebindingen en variabelen te hernoemen, zorgen we ervoor dat we een totale functie voor substitutie verkrijgen.

2.3 De Bruijn Indices

We gebruikten namen om variabelen voor te stellen in de vorige sectie. Dit is niet de enige manier om variabelen voor te stellen en in dit werk maken we gebruik van een andere manier.

Onze voorstelling zorgt ervoor dat we een representatie voorzien van variabelen waarbij het hernoemen van variabelen niet nodig is. Dit is de bedacht door Nicolas de Bruijn, de zogenaamde de Bruijn indices. De Bruijn indices gebruiken geen symbolen. In de plaats daarvan verwijzen variabelen naar hun binders via een afstandnummer. Als voorbeeld schrijven we de identiteitsfunctie.

$$\lambda x.x = \lambda.0 \tag{2.4}$$

Een andere naam die we verder ook soms gebruiken voor deze notatie met de Bruijn indices is de naamloze notatie.

2.3.1 Shift

Substitutie moet nu opnieuw gedefinieerd worden, voor de de Bruijn representatie. We hebben hiervoor een hulpfunctie nodig genaamd shift. We tonen met een voorbeeld aan waarom deze operatie nodig is. Stel de volgende substitutie voor zowel naamloos als met namenrepresentatie:

$$[1 \mapsto 3\,\lambda.0](\lambda.2) \tag{2.5}$$

$$[x \mapsto c\,\lambda x.x](\lambda y.x) \tag{2.6}$$

Hierbij aannemende dat 1 de index is van x in de context. Als we de substitutieterm zouden substitueren zonder enige aanpassing, dan zullen de variabelen die gesubstitueerd worden niet meer naar de juiste binder wijzen. Dit komt omdat de substitutie de lambda-abstractie passeert. Die introduceert een variabele waardoor de de Bruijn indices niet meer correct zijn als we de term zonder aanpassingen zouden substitueren. Dan zou gewoon substitueren zonder aanpassingen de volgende term geven $\lambda.3\,(\lambda.0)$. Dit is niet correct. Namelijk de variabele verwijzing 3 klopt niet omdat ze gesubstitueerd is in een lambda-abstractie. Ze zou met 1 verhoogd moeten zijn om nog altijd de juiste variabele aan te duiden. We zullen een naïeve poging doen om de shiftfunctie te beschrijven.

shift d c k	= k+d
shift d c $\lambda.t_1$	= λ. (shift d (c+1) t_1)
shift d c (t_1 t_2)	= (shift d c t_1) (shift d c t_2)

Deze definitie is echter fout want $\lambda.4\,(\lambda.1)$ is niet het correcte resultaat. De functie zoals hiervoor gedefinieerd, verhoogt elke variabele met een bepaalde waarde. De variabele die gebonden is door de lambda in de substitutieterm moet echter niet verhoogd worden aangezien deze dan niet meer naar zijn binder zou verwijzen. We lossen dit op door een parameter bij te houden die aangeeft welke variabelen verhoogd moeten worden. De shift operatie is dus als volgt te verwoorden in pseudocode:

shift d c k	= if k < c then k else k+d
shift d c $\lambda.t_1$	= λ. (shift d (c+1) t_1)
shift d c (t_1 t_2)	= (shift d c t_1) (shift d c t_2)

2.3.2 Substitutie

Nu kunnen we de substitutie verwoorden. In woorden gezegd, we verhogen de variabelen in de substitutieterm met 1 als we onder een lambda gaan. Bij variabelen substitueren we enkel wanneer de variabelen gelijk zijn en anders blijft de variabele hetzelfde.

subst j s k	= if k ==j then s else k
subst j s $\lambda.t_1$	= λ. (subst (j+1) (shift 0 1 s) t_1)
subst j s (t_1 t_2)	= (subst j s t_1) (subst j s t_2)

Technisch gezien is de echte voorstelling zoals ze in dit werk gebruikt wordt licht verschillend van de de Bruijn indices zoals wij ze hier zullen voorstellen. Dit verschil wordt uitgelegd in Hoofdstuk 4.

2.4 Haskell

De programmeertaal Haskell [2] is een functionele programmeertaal met een sterk typeringssysteem. Dit zorgt ervoor dat in een groot aantal gevallen runtime errors vaak te voorkomen zijn. In plaats van klasses gebruiken we in Haskell algebraic datatypes. Deze zijn te schrijven als :

```
data Bool = True | False
data List a = Nil | Cons a (List a)
```

Functies in Haskell zien er als volgt uit:

```
sumlist :: List Int ->Int
sumlist Nil = 0
sumlist (Cons x xs)= x + sumlist xs
```

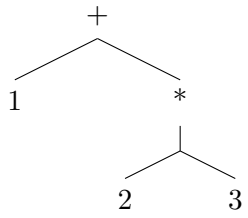
Hierbij wordt opgemerkt dat we aan pattern matching doen. Afhankelijk van welke data constructor we tegenkomen, voeren we een verschillende actie uit. Vlak boven de functie hebben we een functie-annotatie toegevoegd. Deze hoeft vaak niet expliciet aan Haskell meegegeven te worden omdat Haskell een vrij sterk inferentie algoritme heeft om te bepalen welk type een functie heeft. Als het niet uitmaakt welke waarde de pattern match heeft, kan men _ typen. Dan zal men meteen de de functie achter het gelijkheidsteken uitvoeren ongeacht de waarde van de invoer.

Een ander construct om de control flow te bepalen is aan de hand van guards. Dit valt te zien in volgende figuur. Guards worden afgegaan totdat de conditie van een guard true is of wanneer men de otherwise clause tegenkomt.

```
isNil::List a -> Bool
isNil Nil = True
isNil _= False
listToMessage :: List a -> String
listToMessage x
  | isNil x = "Empty List"
  | otherwise = "Not Empty List"
```

2.5 Abstract Syntax

Als we een programma schrijven, zien we enkel tekst of met andere woorden de concrete syntax, maar niet hoe deze geïnterpreteerd wordt door de computer. Deze leest de code uit en genereert dan een boomstructuur. Deze noemen we de abstract syntax tree. In de calculi die we bestuderen, zullen we het nooit over concrete syntax hebben. Men gaat er van uit dat de concrete syntax al omgezet is naar de abstracte syntax omgezet. De boom van bijvoorbeeld de term 1+2*3 ziet er als volgt uit:



2.6 Samenvatting

We hebben in dit hoofdstuk van de programmeertaal Haskell uitgelegd enkele basisbegrippen uitgelegd. We gebruiken Haskell verder in deze thesis om code fragmenten te tonen van functies die we genereren. We lichten hier ook de lambda calculus met types toe. Daarna legden we het probleem uit dat optreedt bij variabelen bij substitutie. We toonden eerst een voorbeeld met symbolische representatie. Daarna toonden we een representatie aan de hand van de Bruijn indices die we verder in dit werk gebruiken. De generatie van deze functies is het hoofddoel van dit werk.

Hoofdstuk 3

Probleem

In dit hoofdstuk bespreken we een voorbeeldimplementatie in Haskell van de getypeerde lambda calculus zoals we formeel behandeld hebben in Hoofdstuk 2. Stapsgewijze tonen we hierop hoe we tot de probleemstelling en bijhorende oplossing komen. Eerst geven we de code voor substitutie om dan te zien dat deze op basis van bepaalde informatie kan gegenereerd worden. Dit leidt ons tot de oplossing zoals die besproken wordt in het volgende hoofdstuk.

De codevoorbeelden in dit hoofdstuk zijn grotendeels gebaseerd op de voorbeeldimplementatie in Types and Programming Languages [9].

3.1 Inleiding

Het probleem in deze thesis is dat het correct implementeren van werken met mechanismes om variabelen te binden moeilijk is om correct te implementeren. Het is ook niet interessant om deze code zelf te schrijven, aangezien er weinig inventiviteit komt bij kijken. Alhoewel de meeste regels in verband met variabelen in calculi misschien kort formeel geschreven kunnen worden, is het in code omzetten hiervan heel wat uitgebreider dan men op het eerste gezicht zou denken. Een voorbeeld van zo'n formele regel is bijvoorbeeld de E-AppAbs regel besproken in Hoofdstuk 2. Deze regel definieert evaluatie van een functie-applicatie als de substitutie van een argument in de body van de functie.

Het schrijven van de code voor deze substitutie van variabelen neemt, zelfs bij eenvoudige calculi zoals de lambda calculus, vaak minstens de helft van alle code van een prototype in beslag. Dit toont aan dat een tool die deze code automatisch genereert handig is voor onderzoekers die een calculus in Haskell willen implementeren. Het bespaart met name veel tijd.

```
Type Index = Int
data Term  = TermLam Type Term | TermVar Index  | TermApp Term Term
data Type  = BaseType | ArrType Type Type
```

Figuur 3.1: Datatypes

3.2 Implementatie van Shift en Substitutie

Om het probleem duidelijk aan te tonen zullen we de getypeerde lambda calculus implementeren zoals formeel besproken in Hoofdstuk 2. De abstracte syntax kunnen we in Haskell makkelijk als algebraïsche datatypes beschrijven zoals te zien in Figuur 3.1.

Bij dit werk kiezen we voor de representatie met de Bruijn indices zoals gezegd in Hoofdstuk 2. Een mogelijke uitwerking in Haskell zou er dan kunnen uitzien zoals in Figuur 3.1:
Het algebraïsche datatype Term is recursief aangezien het ook zichzelf bevat in enkele van de constructors. Ook Type is een recursief datatype. Een functie om te substitueren, doorloopt daarom de recursieve structuur van de termen. Wanneer een abstractie wordt tegengekomen, moeten we rekening houden dat de verwijzing naar de variabele met 1 verhoogd moet worden.

Zowat de meest gebruikte operatie die geïmplementeerd moet worden, is de substitutie-operatie. Deze operatie heeft op zijn beurt ook de shiftoperatie nodig. We vergelijken substitutiefunctie en de shiftfunctie in Figuur 3.2.

Er valt te zien dat zowel de substitutie-operatie als de shiftoperatie zeer gelijkaardig zijn, namelijk: ze overlopen de AST, opereren enkel op variabelen en laten de andere structuren onaangeraakt.

Om code uit te sparen kunnen we de gemeenschappelijke logica voor het overlopen van de abstracte syntax boom schrijven. Deze kunnen we vatten in de hogere orde functie `tmmap` als te zien in Figuur 3.3.

Hierbij kunnen we opmerken dat de logica van variabelebinding te zien is in de code `tmmap` en dat afhankelijk van de operatie (shift of substitutie) de onvar functie zal verschillen. De logica voor de variabelebinding in de `tmmap`-functie is te zien door de parameter c. Die parameter wordt enkel bij de lambda-abstractie verhoogt. Het is namelijk de parameter die aangeeft bij shifts wanneer men moet verhogen. Dit is c<=var voor shifts en c==var voor substitutie. Voor shift zal de shiftOnVar functie meegegeven worden en bij de substitutie de substOnVar operatie.
Interessant om op te merken is dat de substitutiefunctie licht verschilt van hoe ze in het vorige hoofdstuk behandeld werd. Namelijk in plaats van de substitutieterm telkens te shiften per keer we een binder tegenkomen, passen we alle shifts pas toe

```
shiftOnVar d c x
  | x >= c    = TermVar (x + d)
  | otherwise = TermVar x

shift d c (TermVar x )    = shiftOnVar d c x
shift d c (TermApp t1 t2) =
    TermApp (shift d c t1) (shift d c t2)
shift d c (TermLam ty t)  =
    (TermLam ty (shift d (c + 1) t))


substOnVar j s c x
  | x == j + c = shift c 0 s
  | otherwise  = TermVar x

substitute j s c (TermVar x )    = substOnVar j s c x
substitute j s c (TermApp t1 t2) =
    TermApp (substitute j s c t1) (substitute j s c t2)
substitute j s c (TermLam ty t)  =
    (TermLam ty (substitute j s (c + 1) t))
```

Figuur 3.2: Vergelijking shift en substitutie

```
tmmap onvar c (TermVar x )    = onvar c x
tmmap onvar c (TermApp t1 t2) =
    TermApp (tmmap onvar c t1) (tmmap onvar c t2)
tmmap onvar c (TermLam ty t)  =
    (TermLam ty (tmmap onvar (c + 1) t))

shiftOnVar d c x
  | x >= c    = TermVar (x + d)
  | otherwise = TermVar x

substOnVar j s c x
  | x == j + c = termshift c s
  | otherwise  = TermVar x
```

Figuur 3.3: Gemeenschappelijke structuur

```
termSubst :: Int -> Term -> Term -> Term
termSubst j s t  = tmmap (substOnVar j s) 0 t

termshift :: Int -> Term -> Term
termshift d    t = tmmap (shiftOnVar d) 0 t

termSubstTop :: Term -> Term -> Term
termSubstTop s t = termshift (-1) (termSubst 0 (termshift 1 s) t)
```

Figuur 3.4: Implementatie substitutie applicatie

```
isVal :: Term -> Bool
isVal (TermLam _ _ ) = True
isVal _              = False


eval1 ::  Term -> Maybe Term
eval1  ((TermApp (TermLam ty t1 ) t2))
  | isVal t2 = Just (termSubstTop t2 t1)
eval1  ((TermApp t1 t2))
  | isVal t1 = do
    t3 <- eval1  t2
    return (TermApp t1 t3)
eval1  ((TermApp t1 t2)) = do
  t3 <- eval1  t1
  return (TermApp t3 t2)
eval1 _ _ = Nothing

eval ::  Term ->  Term
eval  t = maybe t  (eval  ) t2
  where
    t2 = eval1  t
```

Figuur 3.5: Implementatie evaluatie

bij het effectief substitueren van een term. Op te merken valt ook dat types geen termvariabelen bevatten en dus nooit een substitutie-operatie nodig zullen hebben.

Figuur 3.4 toont de code voor substitutie in het Haskell prototype.

Verdere uitleg van hoe de code in elkaar zit valt intuïtief kort te zeggen. We gebruiken de substitutiefuncties om de functie termSubstTop te creëren. Deze voert

de logica uit om substituties voor applicaties uit te voeren zoals al gezien is in Hoofdstuk 2. Eerst verhogen we elke term in de substitutieterm met 1 omdat we een binder passeren. Dan voeren we de substitutie uit. Uiteindelijk verminderen we alles met 1 omdat een binder is weggevallen. Dit resulteert niet in een negatief cijfer omdat de term die we substitueren 0 is en dus nergens meer mag voorkomen. We zien dat indien er ergens een fout in deze functies had gezeten en we vervolgens een negatief cijfer hadden bekomen, we automatisch fout bezig zijn. Dit is een voordeel van De Bruijn indices, namelijk dat het vaak zeer hard opvalt wanneer er fouten als deze optreden. Het fout verwijzen naar variabelen na substituties zal snel optreden zelfs al testen we maar enkele termen.

Bij het implementeren van verschillende Haskellprototypes is het niet ongewoon dat de code in verband met variabelebinding $\sim 50\%$ van de geschreven code in beslag neemt bij kleine calculi. Bij grotere en complexere calculi kan dat zelfs nog meer innemen. De code waar we echter geïnteresseerd in zijn is die in verband met typering/evaluatie. Met andere woorden, we schrijven veel code die niet interessant is. Dit is wat ervoor zorgt dat het een grote tijdswinst is om die functies te genereren.

3.3 Benodigdheden

Er valt alvast op te merken dat met de informatie die hierna opgesomd wordt praktisch alles qua informatie gegeven wordt om deze operaties te kunnen genereren. Deze informatie is dezelfde als degene dat Inbound vereist. Eerst hebben we een grammatica nodig van alle AST-structuren. Daarna hebben we info rond variabelebinding nodig:

- Structuren die wel/niet variabelen introduceren
- Welke structuren variabelen zijn

Deze info is nodig, omdat ze toelaat de functie `tmmap` terug te creëren. Door te weten welke structuur een variabele is, weten we waar we de onvar functie moeten gebruiken. Door te weten welke structuren variabelen introduceren weten we met hoeveel we onze parameter c in `tmmap` moeten verhogen. Dit is het vertrekpunt van onze oplossing.

3.4 Samenvatting

In dit hoofdstuk hebben we een prototype van de lambda calculus, geïmplementeerd om aan te tonen dat variabelebinding veel code in beslag neemt. Dit ondanks het feit dat ze eigenlijk bij evaluatie maar op selecte plaatsen voorkomt. Dit toont aan dat het gebruik van code generatie een oplossing kan zijn. Praktische implementaties kunnen namelijk al snel enkele honderden lijnen aan code aan variabelebinding spenderen. We beschreven ook dat er maar een beperkte hoeveelheid informatie nodig is om al deze code te laten genereren.

Hoofdstuk 4

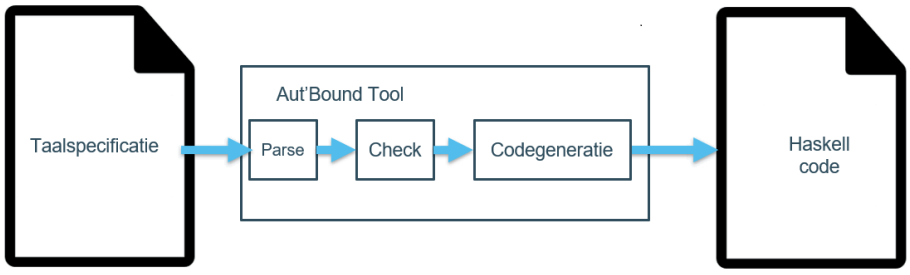
Oplossing

Dit hoofdstuk beschrijft de Aut'Bound compiler die we ontwikkeld hebben om de probleemstelling aan te pakken. Eerst geven we een hoogniveau overzicht van de tool en vervolgens bespreken we alle onderdelen in detail.

4.1 Overzicht

Een overzicht van hoe de Aut'Bound compiler zich gedraagt kan gezien worden in de Figuur 4.1. De Aut'Bound compiler in kwestie werkt als volgt. Het leest een bestand uit geschreven in de Aut'Bound specificatietaal en vertaalt die naar een interne representatie waar dan enkele verificaties op worden uitgevoerd. Deze verificaties zorgen dat namen voor syntaxconstructies uniek zijn.

Daarna genereert de compiler op basis van de interne representatie de Haskell code en schrijft deze uit naar een bestand. Tot slot merken we op dat de tool een minimaal aantal afhankelijkheden van andere libraries heeft. Deze zijn een pretty printing library, een parser library en een code formatter voor Haskell. De Aut'Bound tool is te vinden op `https://github.com/Belpaire/ASTTool`.



Figuur 4.1: Overzicht van de werking van de tool

4.2 Aut'Bound: basisversie

Bij dit werk zijn we vertrokken uit de taal zoals origineel gedefinieerd voor de Inbound tool [14].

Omdat deze specificatietaal variabelebindingen kan beschrijven die meer dan 1 variabele tegelijkertijd kan binden. Dit is belangrijk omdat we anders een heel groot assortiment van complexere talen niet zouden kunnen uitdrukken. Hierna bespreken we kort de verschillende syntactische structuren van de taal, en daarna de structuren die deze thesis toevoegt.

De originele specificatie zoals in Inbound [14] beschreven, is te zien in Figuur 4.2.

Labels	
S,T,U	Sort label
K	Constructor label
F	Field label
X,Y,Z	Meta-variable
A,B,C	Attribute label
α, β, γ	Namespace label
Declaration and definitions	
spec ::= namespace α S sortdecl	Specification
sortdecl ::= sort S syndecl inhdecl cdecl	Sort
syndecl ::= synA: [α]	Inherited attr.
inhdecl ::= inhA: [α]	Synthesized attr.
cdecl ::= K vfield sfield attrdef	Term constructor
|K(X@A)	Var. constructor
vfield ::= X:α	Variable binder
sfield ::= F:S	Subterm
attrdef ::= N.A=e	Attribute def.
N ::= lhs | F	Node label
Context expressions	
expr,e,a,b,c::i=N.A	Context reference
| [ ]	Empty context
| e,X	Context extension

Figuur 4.2: Originele specificatietaal

4.2.1 Namespaces

De syntax voor het opnoemen van namespaces is als volgt: namespace (naam van namespace) (naam van sort). Hier is het dus zo dat een namespace zegt dat variabelen naar deze namespace kunnen verwijzen en dat variabelen dat ernaar verwijzen enkel door de sort in kwestie kunnen gesubstitueerd worden. Bij de getypeerde lambda calculus zal er maar 1 namespace nodig zijn namelijk die voor Terms. Bij uitgebreidere talen kunnen ook variabelen bij types voorkomen.

4.2.2 Sorts

Sorts zijn de structuren die overeenkomen met de verschillende algebraïsche datatypes in de output. Ze komen overeen met de datatypes die we in Hoofdstuk 3 definieerden. In de getypeerde lambda calculus zouden dit dus types en termen zijn.

4.2.3 Inherited en synthesized contexts

Vlak na de declaratie van een sort worden de namespaces gedeclareerd waartoe een sort toegang heeft. De inherited contexts slagen op het feit dat sorts enkel de contexten waar ze toegang tot hebben kunnen opbouwen naarmate men dieper gaat in de subtermen. Als voorbeeld kunnen we de Figuur 4.3 bekijken op hun contexten.

Synthesized contexts worden gebruikt om het mogelijk te maken meer dan 1 variabele te binden in een term. Dit wordt dan uitgedrukt in het geval van Patterns, waarbij de synthesized context steeds uitgebreid wordt met de linkersubpattern en de rechtersubpattern. Deze kunnen we dan bijvoorbeeld in de TmLet gebruiken om op deze manier aan te duiden dat meerdere variabelen tegelijkertijd zullen worden geïntroduceerd. Synthesized contexts werken dus in hun datastroom tegenovergesteld ten opzichte van inherited contexts, die van top naar bottom gaan terwijl synthesized contexts bottom-up werken.

4.2.4 Constructordeclaratie

Dit gedeelte van de syntax komt overeen met de effectieve constructors van de datatypes. Er zijn drie verschillende soorten constructors. Ten eerste de variabele-constructors (aangeduid met een @ en een verwijzing naar de context), ten tweede de constructors die geen variabelen introduceren, en ten laatste de constructors die dat wel doen (aangeduid met [variabelenaam:Namespace] na de subtermen van de constructor). Bij de niet-variabele constructors duidt (x : SortX) aan dat er een subterm is van het type SortX.

4.2.5 Regels

Na de constructordeclaratie komen de regels in verband met variabelebinding. Deze specifiëren of er een variabele in scope wordt gebracht. In het geval van synthesized

contexts wordt dan weer beschreven hoe deze contexten opgebouwd worden. Aangezien de inherited context vaak hetzelfde blijft, wordt er in het geval dat er niets gespecifieerd wordt over de context, aangenomen dat deze onaangepast blijft.

Voor het geval van een inherited contextregel kunnen we volgend voorbeeld geven. Te zien na de TmAbs lijn is de regel die specifieert dat de context van de subterm x uitgebreid wordt met een variabele. Voor de synthesized context zien we een voorbeeld na de PVar lijn. Deze specifieert dat de synthesized context sCtx opbouwt met 1 variabele meer dan de inherited context iCtx.

4.3 Aut'Boundspecificatietaal : Nieuwe Concepten

In dit werk zijn aanvullingen op de oorspronkelijke taal zoals hiervoor beschreven, gemaakt die we hier bespreken. Bij de keuze van deze verschillende toegevoegde constructies in de grammatica van de taal is er vooral gelet om een syntax te behouden die logisch is aan de kant van de programmeurs en die bovendien geen ambiguïteit in de grammatica zou laten verschijnen.

4.3.1 Voorgedefinieerd Type

De hiervoor besproken taal is praktisch gezien voldoende voor de meeste eenvoudige calculi. Er is echter geen mogelijkheid om bijvoorbeeld primitieve types in constructors van datatypes te hebben. Daarom voorziet de uitgebreide taal hierin door een extra soort parameter in de specificatietaal te voorzien. Deze wordt geschreven als: Constructorname (velden) { String }
Een praktisch voorbeeld is te zien in Figuur 4.4, namelijk de specificatie dat records bevat. Hierbij zijn de labels van de Records voorgesteld door Strings. Een record is een datastructuur die een aantal labels met elk een geassocieerde term bevat. Het moet opgemerkt worden dat op constructies van het geïmporteerde type geen verificaties worden uitgevoerd, Het is de verantwoordelijkheid van de programmeur zelf om deze juist te schrijven.

4.3.2 Lijsten

Bij het zien van de vorige specificatie valt op te merken dat de Sort Record in onze metataal feitelijk gezien dezelfde structuur als een lijst heeft. Daarom voorzien wij ook de mogelijkheid om in plaats van (X:SortX) ook (X: [SortX]) te schrijven. Hierbij worden wel enkele assumpties geplaatst op het inheriten van de context, namelijk elke subterm binnen de lijst zal dezelfde inherited context hebben. In de meeste gevallen is dit meer dan genoeg.

```
namespace TermVar : Term
sort Term
  inh ctx TermVar
  | TmVar (x@ctx)
  | TmAbs (x:Term) [z:TermVar]
    x.ctx = lhs.ctx,z
  | TmTrue
  | TmFalse
  | TmApp (t1 : Term) (t2 : Term)
    t1.ctx= lhs.ctx
    t2.ctx= lhs.ctx
  | TmRecord (d : [RecordTerm])
sort RecordTerm
  inh rctx TermVar
  | RecStringTerm ( t : Term) {String}
```

Figuur 4.5: Implementatie aan de hand van lijsten

Is er een meer geavanceerde vorm van variabelebinding nodig, dan kan met synthesized contexts deze toch nog beschreven worden. Door middel van deze methode kan men bijvoorbeeld de Figuur 4.4 herschrijven naar een methode die lijsten gebruikt. De implementatie met lijsten is te vinden in Figuur 4.5.

4.3.3 Extra info aan Env toevoegen

Bij het aanmaken van een namespace kunnen we na de namespace namespacename : sortname een komma gevolgd door een sort schrijven. Dit genereert een extra veld in de Env datastructuur. Hiermee kunnen we daar dus extra info in plaatsen. Dit is handig voor extra informatie te hebben in het Env datatype voor functies in verband met het bepalen van het type van een term. Hoe men dit kan gebruiken is te zien in de functie getTypeFromEnv in Figuur 4.14.

4.3.4 Algemene datastructuren

Overigens is de mogelijkheid om naast lijsten ook andere datastructuren te gebruiken, op voorwaarde dat ze de Foldable en Functor typeclass implementeren. Een voorbeeld is te zien in Figuur 4.6.

De Foldable typeclass zorgt ervoor dat men over functies beschikt die over de elementen van een datastructuur kunnen itereren. Foldable heeft verschillende functies maar de belangrijkste ervan zijn de foldr en foldl functies. De type signaturen

van de folds voor lijsten zijn als volgt.

$$foldl :: (a \to b \to a) \to a \to [b] \to a \tag{4.1}$$

$$foldr :: (a \to b \to b) \to b \to [a] \to b \tag{4.2}$$

Deze signatuur valt vrij intuïtief te begrijpen. De foldfuncties hebben startwaardes, namelijk het tweede argument. Dan zal men over de lijst itereren en de binaire functie, het eerste argument van de foldfuncties, toepassen. Een resultaat van hoe dit eruitziet valt hieronder te zien [3].

$$foldl \quad (-) \quad 0 \quad [1,2,3,4] \tag{4.3}$$

$$((((0-1)-2)-3)-4) = -10 \tag{4.4}$$

Het verschil tussen deze twee functies is dat foldl van het eerste element begint en foldr van het laatste element.

De functor typeclass is een voorstelling van de types waar men een mapfunctie op kan toepassen. Een mapfunctie is een functie die een structuur en een functie als argument heeft en de functie dan vervolgens op elk element van de structuur toepast. Deze mapfunctie die dan geïmplementeerd moet worden, is de fmap-functie.

$$fmap :: (a \to b) \to fa \to fb \tag{4.5}$$

Bij de fmap typering valt weinig op te merken. Ze neemt een functie en op basis van deze functie wordt dan op elk element van een datastructuur deze functie toegepast.

Waarom moet de datastructuur deze twee typeclasses implementeren? Wel de fmap-functie, voorzien door de functor typeclass, geeft de mogelijkheid om net zoals bij lijsten onze substitutiefuncties en shiftingsfuncties op elk element toe te passen. De Foldable typeclass is nodig voor de code in verband met de vrije variabelen. Dit om de reden dat de Foldable typeclass een operatie foldmap voorziet. De typesignatuur van foldMap is de volgende:

$$foldMap :: Monoid\ \ m => (a \to m) \to t\ \ a \to m \tag{4.6}$$

Opnieuw kunnen we de typesignatuur interpreteren. Monoid m specifieert dat het type m de typeclass Monoid moet implementeren. We weten dat lijsten deze typeclass implementeren dus er is geen probleem. Als we kijken naar de Haskell Source Code zien we de volgende code:

$$foldMap\ \ f = foldr \quad (mappend.f) \quad mempty \tag{4.7}$$

Dit lezende zien we twee termen die geïmplementeerd moeten worden in de typeclass Monoid: mappend en mempty. De code is vrij simpel te interpreteren. Deze twee termen zijn gewoon de concatenatie-operator (mappend) en de lege lijst (mempty)

in het geval van lijsten. Omdat onze datastructuur ook een Functor moet zijn, zal foldmap de volgende regel volgen:

$$foldMap\ f = fold \quad . \quad fmap\ f \tag{4.8}$$

We hebben foldr al gezien in het vorige stuk dus op zich kennen we alle elementen van deze functie. Eerst wordt de functie meegegeven in foldMap toegepast op het element. Hierna wordt deze geconcateneerd met de rest van de resultaten.

In ons geval hebben we een functie die dus lijsten berekent van vrije variabelen voor elk element van de datastructuur. De functie foldMap zorgt ervoor dat voor elk element van de datastructuur die lijst berekend wordt en dat al deze lijsten dan geconcateneerd worden. Technisch gezien bestaat er al een functie die dit specifiek voor lijsten doet genaamd concatMap. Maar indien we van lijsten naar een ander datatype zouden overschakelen dan is het handig om dit alvast generieker te schrijven. Een korte implementatie hier kunnen we maken door in plaats van een lijst zoals in de vorige versie bijvoorbeeld een ander datatype zoals Vector te gebruiken. De vraag blijft nu echter hoe we datatypes die niet automatisch geïmporteerd worden in onze code krijgen. Dit lossen we op in de volgende sectie. De uitleg in verband met folds en functors is gebaseerd op de Haskell documentatie [3, 6].

4.3.5 Import Declaraties

Het is mogelijk om in het gegenereerde bestand ook import statements te gebruiken. Op deze manier kan men dit als een voorgedfinieerd type gebruiken dat ofwel gedefinieerd is in een zelfgeschreven file ofwel gedefinieerd is in de standaard Haskell installatie. Importeren doet men door voor de namespaces alle imports te schrijven. `import (name)` waarbij name dan de naam van hetgeen men wil importeren is. Het gevaar bestaat natuurlijk dat er een loop zou zijn van imports, dit is echter aan de programmeur zelf om dan te voorkomen, al is de kans dat men veel imports nodig heeft bij de gegenereerde code vrij klein. Op deze manier kunnen we de datastructuren en functies importeren die men nodig heeft. Indien men niet alles vanuit een module wil importeren maar enkel bepaalde functies of datastructuren, dan kan men een tweede paar haakjes achter de import zetten. Hiermee duidt men dan de specifieke functies en datastructuren aan waardoor enkel deze zullen worden geïmporteerd.

4.3.6 Letterlijke Haskell Code

Overigens is het ook mogelijk om zelf letterlijk code in de gegenereerde file te laten komen. Dit doet men door op het einde van de specificatie nog HaskellCode te schrijven met daarna de lijnen letterlijke code. Deze lijnen zullen dan op het einde van het gegenereerde bestand worden toegevoegd. Een voorbeeld vinden we in Figuur 4.7. Deze calculus beschrijft A-normal form beschreven in de paper The Essence

of Compiling with Continuations [11]. In A-normal form staan, betekent dat het argument aan een functie enkel een variabele, constante of functie kan zijn. Anders moet men de term zo herschrijven dat een argument door een let-construct gebonden is. Tijdens evaluatie kan het voorkomen dat een term niet meer in ANF staat. Dit gebeurt in ons geval enkel bij reducties. Willen we dit dus voorkomen, kunnen we na elke substitutie de term herschrijven zodat ze weer in ANF staat.

4.3.7 Rewrite

Rewrite operaties kunnen na een substitutie toegevoegd worden. In sommige calculi is het nodig dat na een substitutie de term opnieuw geschreven moet worden. Die functies moeten dan oftewel via een import komen of via Haskell Code zelf geschreven zijn, en moeten gedefinieerd zijn voor alle gevallen van de specifieke term die men na substitutie wil herschrijven.

4.4 Parser

Aangezien we met behulp van onze eigen specificatietaal code genereren, moeten we een manier vinden om deze naar een interne representatie om te vormen. Dit proces is het geheel van lexen en parsing. Lexing zorgt ervoor dat de specificatie, die origineel gezien wordt als een hoop karakters, naar tokens wordt getransformeerd. Parsing slaagt dan op het omvormen van deze hoop tokens naar een interne representatie. Deze definities komen rechtstreeks uit het boek essentials of programming languages [12].

De taal wordt geparsed naar een interne representatie. Deze is geïmplementeerd via parsec [7], een parser library. Het lexen van de input file gebeurt ook via deze library.

4.5 Verificaties en Foutmeldingen

Nadat we onze interne representatie bemachtigd hebben, kunnen we enkele verificaties hierop uitvoeren. Het grootste gedeelte hiervan heeft als doel te voorkomen dat er geen overlap qua namen is in de output. Enkele voorbeelden zijn:

- Elke Sort/Constructor/namespace heeft een verschillende naam ten opzichte van elkaar.
- Synthesized en inherited contexts kunnen niet zomaar op eender welke kant van een regel staan, e.g. x.ctx=y.ctx is geen zinnige uitspraak, behalve als y een lijst is.
- Subtermen in constructors moeten unieke namen hebben binnen de constructor zelf. Anders zou men de regels in verband met contexten niet kunnen afleiden.
- Wanneer een subterm naar een sort verwijst, moet deze ook in de declaratie voorkomen, idem ook voor alle namespaces en contexts.

Deze verificaties hebben als doel om ervoor te zorgen dat de code die uiteindelijk gegenereerd zal worden dus effectief zonder compilatiefouten zal zijn.

Er zijn twee soorten foutmeldingen: zij die voorkomen bij het parsen en zij die voorkomen bij het effectief verifiëren van de interne representatie. Deze worden dan via tekst meegedeeld aan de gebruiker moest er zich dus ergens een fout bevinden. Deze meldingen geven een korte string mee van het soort fout dat er gebeurde.

4.6 Gegenereerde Code

De gegenereerde code is een Haskellmodule. Hiervan kan de programmeur alles importeren dat allicht nodig is om een implementatie van een calculus te hebben. De tool voorziet de volgende functies aan de programmeur:

- Shiftingsfuncties en substitutiefuncties (de tool kan zien welke algebraïsche datatypes geen variabelen bevatten, en zal voor deze ze ook niet genereren, gezien die functies nutteloos zouden zijn geweest).
- Functies voor de vrije variabelen te verkrijgen uit een term.
- Alle algebraïsche datatypes die als sorts stonden in de specificatie.
- Een datatype genaamd Env en HNat, die in de volgende paragrafen meer zullen geduid worden.
- Functies in verband met getallen op te tellen, af te trekken en te vergelijken. Deze heten plus, minus en compare respecievelijk.

4.6.1 Heterogene natuurlijke getallen

Gezien wij vertrekken vanuit een variabelerepresentatie met De Bruijn indices, gebruiken we natuurlijke getallen en geen namen voor variabelen. Echter stel dat we in bijvoorbeeld een calculus met meerdere soorten variabelen een implementatie willen maken. We nemen hier nu een concreet voorbeeld namelijk System F in Figuur 4.8. Hier zien we dat er dus twee contexten zijn, 1 voor termvariabelen en 1 voor typevariabelen. Stel dat we een typering voor deze calculus zouden willen schrijven, dan zou het handig zijn om in deze natuurlijke getallen meer informatie over de typering in de omgeving mee te geven. Onze oplossing hiervoor is om niet gehele getallen te gebruiken maar onze eigen datastructuren. Een index 1 zou erop duiden dat de variabele verwijst naar de binder met afstand 1. Om verder te gaan in het geval van typevariabelen en termvariabelen zouden we dan bijvoorbeeld 1 kunnen schrijven als volgt.

$$STermVarZ \quad | \quad STypeVarZ \tag{4.9}$$

Dit duidt aan dat de afstand nog altijd 1 is, maar nu weten we ook meer over de binder die men moet passeren alvorens daar te geraken. Dit is ofwel een binder die een termvariabele introduceert ofwel een binder die een typevariabele introduceert.

Z is het equivalent van 0, het geval waarbij men geen binders passeert. In onze tool wordt de naam van de verschillende heterogene variabelen gegenereerd als S+<namespacename>. Om het eventjes duidelijker te stellen:

$$\lambda\ f : B \to B.\forall\alpha.\lambda\ x : B.f\ x \tag{4.10}$$

$$\lambda\ .B \to B.\forall.\lambda\ B.2\,0 \tag{4.11}$$

$$\lambda\ .B \to B.\forall\alpha.\lambda\ B.(STermVar(STypeVarZ))\ Z \tag{4.12}$$

term t ::= x	variable
|λ x : T .t	abstractie
| t t	applicatie
| t T	type-applicatie
| $\forall\alpha.t$	type abstractie
type T ::= B	basetype
| T → T	functietype
| α	typevariabele
|$\forall\alpha.T$	universeel type

Figuur 4.8: Specificatie van System F

4.7 Voorbeeld gegenereerde code

We tonen een stuk code gegenereerd voor de volgende specificatie, te zien in Figuur 4.9. Hier enkel getoond is de code voor de datastructuur en de functie die over de datastructuur loopt in Figuur 4.10. De rest van de code daarvoor gegenereerd is te vinden in Appendix B. Wegens het feit dat we in het komende Hoofdstuk 5 geen gebruik maken van synthesized contexts zullen we ook even tonen welke code gegenereerd wordt als men deze gebruikt. Hiervoor nemen we terug het voorbeeld van STLC met tupels 4.3. Het resultaat dat bekomen wordt is te zien in Figuur 4.11. Wat valt op te merken bij de functie addToEnvironmentPatsCtx is dat deze dus exact overeenkomt met de specificatie in de sort Pat. De functie wordt gemaakt op basis van de informatie van de synthesized context. Dit door te beginnen bij lhs.sCTX en pas te stoppen wanneer men lhs.iCtx tegenkomt. Op basis van de regels die men tegenkomt kan men de functie laten genereren.

```
addToEnvironmentPatsCtx :: Pat -> HNat -> HNat
addToEnvironmentPatsCtx (PVar) c = STermVar c
addToEnvironmentPatsCtx (PProd p1 p2) c =
addToEnvironmentPatsCtx p2 ((addToEnvironmentPatsCtx p1 c))
```

Figuur 4.11: Code voor tuples en patterns

4.8 Gebruik bij Evaluatie en Typering

Het doel van dit werk is om deze tools ter beschikking van programmeurs te stellen. Nemen we even System F als voorbeeld. Deze kunnen we in onze tool als volgt schrijven. Een voorbeeld van hoe deze specificatie eruit ziet is te vinden in Figuur 4.12.

```
namespace TermVar : Term ,Type
namespace TypeVar : Type
sort Term
    inh ctx TermVar
    inh tctx TypeVar
 | TmVar (x@ctx)
 | TmAbs (x:Term) (t:Type) [z:TermVar]
           x.tctx = lhs.tctx
           x.ctx = lhs.ctx,z
           t.tyctx = lhs.tctx
 | TmApp (t1 : Term) (t2 : Term)
 | TmTApp (t1 : Term) (T : Type)
 | TmTAbs (t1 : Term) [T:TypeVar]
       t1.ctx= lhs.ctx
       t1.tctx = lhs.tctx, T

sort Type
    inh tyctx TypeVar
    | TyVar (t@tyctx)
    | TyArr (t1 : Type) (t2 : Type)
           t1.tyctx = lhs.tyctx
            t2.tyctx = lhs.tyctx
    | TyAll (t1:Type) [T:TypeVar]
                  t1.tyctx = lhs.tyctx ,T
    |TyBase
```

Figuur 4.12: System F with Base Type

De code die we we verkrijgen uit de specificatie zullen we dan kunnen gebruiken. We zullen slechts op enkele specifieke lijnen verder inzoomen, de rest van de code is te vinden op https://github.com/Belpaire/ASTTool.

4.8.1 Evaluatie

Hier te zien is de letterlijke vertaling van de evaluatierelatie van termapplicaties en type-applicaties. Wat we moeten doen om de substitutie hierbij juist te laten verlopen is om ervoor te zorgen dat de term die we willen substitueren met 1 naar boven geshift wordt. Dan substitueren we de variabele Z met onze term. Uiteindelijk

shiften we onze hele term ook nog eens met 1 naar beneden. Deze shiftoperatie naar beneden lijkt vreemd op het eerste zicht maar is logisch. We moeten namelijk sommige variabelen met 1 verlagen aangezien een binder is weggevallen. Er is geen gevaar een variabele negatief te maken, want de enige term die bij een negatieve shift van 1 negatief zou worden is de variabele met waarde nul. Maar de nulvariabele is de variabele die we juist gesubstitueerd hebben. Een soortgelijke redenering geldt voor type-applicaties.

4.8.2 Typering

Hier moeten we specifiëren hoe we info uit de Env datastructuur halen. Aangezien enkel termen een type hebben, geven we het type terug dat een variabeleterm heeft in de functie getTypeFromEnv. In de andere gevallen geven we foutmeldingen terug. Als we dan kijken naar enkele voorbeelden in de typeOf code dan zien we weer dezelfde substitutiestructuur terugkomen bij het typeren van de type-applicatieterm. In het geval dat we een type-abstractie tegenkomen voegen we aan de environment toe dat de context uitgebreid wordt met een typevariabele om dan met deze uitgebreide context verder te gaan.

De andere functiegedeeltes zijn dan weer vrij voor de hand liggend. Namelijk een functie moet enkel invoer aanvaarden bij applicatie met een term die het type heeft dat gespecifieerd is. Een termabstractie breidt net zoals de type-abstractie de context uit, maar dan met een termvariabele die type-informatie bevat.

4.9 Samenvatting

In dit deel hebben we een meer specifieke behandeling gegeven van hoe exact de methodiek achter het genereren van de code verloopt. Ook gaven we enkele voorbeelden. Ook de achterliggende beslissingen gemaakt bij het uitbreiden van de syntax werden verder uitgelicht. De taal met uitbreidingen noemen we Aut'Bound en de tool die de Aut'Bound taal interpreteert de Aut'Bound compiler.

```
namespace TermVar : Term
sort Term
inh ctx TermVar
  | TmVar (x@ctx)
  | TmAbs (x:Term) [z:TermVar]
    x.ctx = lhs.ctx,z
  | TmApp (t1 : Term) (t2 : Term)
    t1.ctx = lhs.ctx
    t2.ctx = lhs.ctx
  |TmLet (t1:Term) (t2:Term) (p:Pat)
    p.iCtx=lhs.ctx
    t1.ctx=lhs.ctx
    t2.ctx=p.sCtx
  | TmProd (t1 : Term) (t2 : Term)
    t1.ctx = lhs.ctx
    t2.ctx = lhs.ctx

sort Pat
  syn sCtx TermVar
  inh iCtx TermVar
  | PVar [x:TermVar]
    lhs.sCtx = lhs.iCtx,x
  | PProd(p1 : Pat) (p2 : Pat)
    p1.iCtx=lhs.iCtx
    p2.iCtx=p1.sCtx
    lhs.sCtx= p2.sCtx
```

Figuur 4.3: STLC met tuples

```
namespace TermVar : Term
sort Term
  inh ctx TermVar
  | TmVar (x@ctx)
  | TmAbs (x:Term) [z:TermVar]
    x.ctx = lhs.ctx,z
  | TmTrue
  | TmFalse
  | TmApp (t1 : Term) (t2 : Term)
    t1.ctx= lhs.ctx
    t2.ctx= lhs.ctx
  | TmRecord (d : Record)
sort Record
  inh ctxRec TermVar
  | Empty
  | TermCons (t : RecordTerm) (d : Record)
    t.rctx = lhs.ctxRec
    d.ctxRec = lhs.ctxRec
sort RecordTerm
  inh rctx TermVar
  | RecStringTerm ( t : Term) {String}
```

Figuur 4.4: Implementatie met een record

```
import (Data.Vector) (Vector)
namespace TermVar : Term
sort Term
inh ctx TermVar
  | TmVar (x@ctx)
  | TmAbs (x:Term) [z:TermVar]
    x.ctx = lhs.ctx,z
  | TmApp (t1 : Term) (t2 : Term)
    t1.ctx = lhs.ctx
    t2.ctx = lhs.ctx
  |TmLet (t1:Vector Term) (t2:Term)
    t1.ctx=lhs.ctx
    t2.ctx=t1.ctx
```

Figuur 4.6: Implementatie met Vectorvoorbeeld

```
namespace TermVar : Exp
sort Exp rewrite
    inh ctx TermVar
    | TmLet (x:Exp) (y:Exp)[z:TermVar]
           y.ctx = lhs.ctx,z
    | TmApp (t1 : Exp) (t2 : Exp)
    | Var (x@ctx)
    |Abs (y:Exp)[z:TermVar]
                   y.ctx = lhs.ctx,z
HaskellCode

rewriteExp, ... -->code die de herschrijvingsoperatie in ANF
    doorvoert
```

Figuur 4.7: Voorbeeld voor Haskellcode

```
namespace TermVar : Term
sort Term
    inh ctx TermVar
 | TmVar (x@ctx)
 | TmAbs (x:Term) [z:TermVar]
           x.ctx = lhs.ctx,z
 | TmApp (t1 : Term) (t2 : Term)

 |TmTrue
 |TmFalse
 |TmIf (t:Term) (t2:Term) (t3:Term)
```

Figuur 4.9: Specificatie Lambda Calculus met Booleans

```
data Term
  = TmVar HNat
  | TmAbs Term
  | TmApp Term
          Term
  | TmTrue
  | TmFalse
  | TmIf Term
         Term
         Term
  deriving (Show, Eq)

data HNat
  = Z
  | STermVar HNat
  deriving (Show, Eq)

data Env
  = Nil
  | ETermVar Env
  deriving (Show, Eq)

termmap :: (HNat -> Term -> Term) -> HNat -> Term -> Term
termmap onTermVar c (TmVar hnat) = onTermVar c (TmVar hnat)
termmap onTermVar c (TmAbs x) =
    TmAbs (termmap onTermVar (STermVar c) x)
termmap onTermVar c (TmApp t1 t2) =
  TmApp (termmap onTermVar c t1) (termmap onTermVar c t2)
termmap onTermVar c (TmTrue) = TmTrue
termmap onTermVar c (TmFalse) = TmFalse
termmap onTermVar c (TmIf t t2 t3) =
  TmIf (termmap onTermVar c t)
  (termmap onTermVar c t2) (termmap onTermVar c t3)
```

Figuur 4.10: Deel van de code gegenereerd voor lambda calculus met booleans

```
isVal :: Term -> Bool
isVal (TmAbs x t) = True
isVal (TmTAbs t1) = True
isVal _ = False

stepEval :: Term -> Maybe Term
stepEval (TmApp (TmAbs t ty) t2)
  | isVal t2 =
    Just
      (termshiftminus
         (STermVar Z)
         (termtermSubstitute (termshiftplus (STermVar Z) t2) Z t))
stepEval (TmTApp (TmTAbs t) ty) =
  Just
    (termshiftminus
       (STypeVar Z)
       (termtypeSubstitute (typeshiftplus (STypeVar Z) ty) Z t))
```

Figuur 4.13: Substitutiegebruik bij Evaluatie

```
getTypeFromEnv :: Env -> HNat -> Either String Type
getTypeFromEnv (ETermVar ty _) Z = return ty
getTypeFromEnv _ Z = Left "wrong or no binding for term"
getTypeFromEnv (ETermVar _ rest) (STermVar h) = getTypeFromEnv rest h
getTypeFromEnv _ (STermVar h) = Left "wrong term binding"
getTypeFromEnv (ETypeVar rest) (STypeVar h) = getTypeFromEnv rest h
getTypeFromEnv _ (STypeVar h) = Left "No variable type"

typeOf :: Term -> Env -> Either String Type
typeOf (TmVar nat) ctx = getTypeFromEnv ctx nat
typeOf (TmAbs t ty) ctx = do
  ty2 <- typeOf t (ETermVar ty ctx)
  return (TyArr ty ty2)
typeOf (TmApp t1 t2) ctx =
  case (typeOf t1 ctx) of
    Right (TyArr ty1 ty2) ->
      case (typeOf t2 ctx) of
        Right ty ->
          if ty == ty1
            then Right ty2
            else Left "different parameter expected"
        Left a -> Left a
    Left a -> Left a
    _ -> Left "arrow type expected"
typeOf (TmTApp t ty) ctx =
  case (typeOf t ctx) of
    Right (TyAll ty2) ->
      return
        (typeshiftminus
           (STypeVar Z)
           (typetypeSubstitute (typeshiftplus (STypeVar Z) ty) Z ty2))
    Left a -> Left a
    _ -> Left "not a type abstraction"
typeOf (TmTAbs t) ctx = do
  ty <- typeOf t (ETypeVar ctx)
  return (TyAll ty)
```

Figuur 4.14: Substitutiegebruik bij Typering

Hoofdstuk 5

Evaluatie

In dit hoofdstuk bespreken we drie case studies waarin we een calculus modelleren met behulp van onze Aut'Bound specificatietaal en een prototype-implementatie ontwikkelen met onze Aut'Bound tool.

In deze sectie bespreken we drie calculi, hoe we ze gemodelleerd hebben en een prototype gemaakt hebben.

5.1 System F met Coercions

5.1.1 Achtergrond Calculus

Types	$\tau ::= \text{Int} \mid \langle\rangle \mid \tau \to \tau \mid \tau \times \tau \mid \alpha \mid \forall.\tau$
Terms	$\text{e} ::= \text{x} \mid \text{i} \mid \langle\rangle \mid \lambda x.e \mid e_1 e_2 \mid \langle e_1, e_2\rangle \mid \Lambda\alpha.e \mid e\tau \mid \text{co e}$
Coercions	$\text{co} ::= \text{id} \mid co_1 \circ co_2 \mid \text{top} \mid \text{bot} \mid co_1 \to co_2 \mid \langle co_1, co_2\rangle \mid$ $\pi_1 \mid \pi_2 \mid co_\forall \mid dist_\to \mid top_\to \mid top_\forall \mid dist_\forall$
Values	$\text{v} ::= \text{i} \mid \langle\rangle \mid \lambda x.e \mid \langle v_1, v_2\rangle \mid \Lambda\alpha.e \mid co_1 \to co_2\ v \mid$ $co_\forall\ v \mid dist_\to\ v \mid top_\to\ v \mid dist_\forall\ v$
Term contexts	$\Psi ::= \bullet \mid \Psi, x : \tau$
Type contexts	$\Phi ::= \bullet \mid \Phi, \alpha$
Evaluation contexts	$\varepsilon ::= [\cdot] \mid \varepsilon\ \text{e} \mid \text{v}\ \mid \langle \varepsilon, e\rangle \mid \langle v, \varepsilon\rangle \mid co\ \varepsilon \mid \varepsilon\ \tau$

Figuur 5.1: System F met coercions

De eerste calculus die we bespreken is System F met coercions [10], waarin de syntax te zien is in Figuur 5.1. De calculus bevat termen, types en coercions. Een coercion laat toe om een term naar een andere term om te zetten, soortgelijk aan hoe men in een taal als java een operatie 'cast' kan hebben. De taal heeft enkel type- en termvariabelen. De taal bevat nog enkele andere constructen naast puur system F zoals tuples, een Top term en gehele getallen.

```
namespace TermVar : Term , Type
namespace TypeVar : Type
sort Term
    inh ctx TermVar
    inh tctx TypeVar
 | TmVar (x@ctx)
 | TmAbs (x:Term) (t:Type) [z:TermVar]
           x.ctx = lhs.ctx,z
 | TmApp (t1 : Term) (t2 : Term)
 | TmTApp (t1 : Term) (T : Type)
 | TmTAbs (t1 : Term) [T:TypeVar]
       t1.ctx= lhs.ctx
       t1.tctx = lhs.tctx, T
 |TmTop
 |TmTuple (e1:Term) (e2:Term)
 |TmInt {Int}
 |TmCast (co:Coercion) (e:Term)

sort Type
    inh tyctx TypeVar
    | TyVar (t@tyctx)
    | TyArr (t1 : Type) (t2 : Type)
    | TyAll (t1:Type) [T:TypeVar]
                  t1.tyctx = lhs.tyctx ,T
    |TyInt
    |TyTop
    |TyProd (t1:Type) (t2:Type)

sort Coercion
    inh tycoctx TypeVar
    | CoId
    |CoTrans (co1:Coercion) (co2:Coercion)
    |CoTop (ty:Type)
    |CoBot (ty:Type)
    |CoArrow (co1:Coercion) (co2:Coercion)
    |CoTuple (co1:Coercion) (co2:Coercion)
    |CoProj1 (ty2:Type)
    |CoProj2 (ty1:Type)
    |CoAll (co1:Coercion)
    |CoDistArrow
    |CoTopArrow
    |CoTopAll
    |CoDistAll
```

Figuur 5.2: Invoer van System F Co

5.1.2 Specificatie en Gegenereerde Code

Er zijn enkele kleine verschillen tussen onze specificatie in Figuur 5.2 en de originele calculus zoals gegeven in Figuur 5.1. We hebben namelijk extra type-informatie toegevoegd aan enkele coercions. Deze informatie is niet nodig voor onze tool, maar maakt de implementatie van een typechecker mogelijk, iets waar de auteurs van FCo geen rekening mee hadden gehouden. Zonder deze extra type-informatie zouden sommige termen geen eenduidig type hebben. Bekijken we de term: (TmCast (CoArrow CoTop CoId) (TmAbs (TmVar Z) TyTop)). Deze gedraagt zich als een functie die eerst elk argument naar de Top-term transformeert, om dan deze als argument te aanvaarden. Om de ambiguïteit te voorkomen voegen we type annotaties toe aan sommige coercions. Dit heeft het interessante gevolg dat coercions nu types bevatten. Hierdoor wordt er opeens een substitutie-operatie voorzien voor coercions. In de originele calculus was dit niet het geval.

Overigens zien we ook de notatie voor voorgedefinieerde types terugkomen bij de datastractuur van de TmInt structuur. We zullen zien dat deze syntax voor voorgedefinieerde types in de case studies ook opduiken. Als we kijken naar de gegenereerde code, dan zien we dat ze enkel verschilt bij de code die betrekking heeft tot abstracties.

```
termmap onTermVar onTypeVar c (TmTAbs t1) =
     TmTAbs (termmap onTermVar onTypeVar (STypeVar c) t1)

typemap onTypeVar c (TyAll t1) =
    TyAll (typemap onTypeVar (STypeVar c) t1)

termmap onTermVar onTypeVar c (TmAbs x t) =
  TmAbs (termmap onTermVar onTypeVar (STermVar c) x)
  (typemap onTypeVar c t)

termmap onTermVar onTypeVar c (TmInt int0) =
    TmInt int0

termmap onTermVar onTypeVar c (TmVar hnat) =
    onTermVar c (TmVar hnat)

freeVariablesType c (TyAll t1) =
    nub ((freeVariablesType (STypeVar c) t1))
```

Figuur 5.3: code in system F met coercions

De twee eerste stukken code in Figuur 5.3, namelijk de code van typemap voor TyAll en de code van termmap voor TmTabs, duiden op het toevoegen van een type-

Variabele in een term en een type respectievelijk. De vertaling van specificatie naar code, valt hier dus duidelijk op. We zien namelijk dat beide een variabele in scope brengen in de specificatie en we zien de successorconstructor ook altijd in die gevallen terugkomen. De code klopt dus in dit geval. Elke sort heeft inherited contexts en voor al deze inherited contexts wordt een verschillende hulpfunctie gedefinieerd. Deze hulpfunctie kan zowel voor substitutie als shifting dienen. Daardoor kunnen we onze functie die mapt over de datastructuur voor beide gebruiken.

Een ander interessante zaak om te zien is hoe we omgaan met de structuur voor voorgedefinieerde types. Om ervoor te zorgen dat deze een unieke naam heeft ten opzichte van andere constructies van voorgedefinieerde types voegen we telkens een cijfer toe aan het einde. Aangezien deze datatypes primitieve datastructuren in Haskell zijn, is er overduidelijk niets in verband met substitutie of shifting nodig voor deze structuren. Ze komen in de output dus gewoonweg exact terug. Voor elke variabele wordt de juiste hulpfunctie gebruikt dus in dit geval onTermVar voor termvariabelen en onTypeVar voor typevariabelen. De code voor vrije variabelen in een term te berekenen is vrijwel hetzelfde als voor de map functie, met het verschil dat de output een lijst van variabelen geeft. De gegenereerde code is te vinden in https://github.com/Belpaire/ASTTool.

5.1.3 Prototype

De functies binnen prototypes zijn meestal de evaluatie-of typeringsfuncties samen met enkele hulpfuncties. We maakten een evaluator, die telkens een regel evalueert in de calculus als hij die tegenkomt. Als er geen evaluatie-regels meer zijn stopt de evaluator. Ook maakten we een typeringsfunctie. Deze geeft aan welk type de term heeft, indien ze een type heeft.

Om deze substitutiecode te testen gebruiken we ze in een evaluator en laten we enkele termen evalueren. Een van de termen dat we testen is (TmApp (TmAbs (TmVar Z) TyInt) (TmInt 5)). Als resultaat van deze term krijgen we het gehele getal 5. Dit klopt met wat we verwachten van de substitutie. Testen in andere hoofdstukken verlopen op een gelijkaardige manier.

Hierbij valt er op te merken dat er geen nood is om een context van variabelen bij te houden voor de evaluatie functie, zoals bij sommige evaluators wel het geval is. Dit komt doordat variabelen niet apart worden bijgehouden in een aparte contextvariabele waar alle variabelen instaan maar omdat we bij evaluatie meteen de variabelen zullen substitueren. Waar we wel het environment datatype voor gebruiken is bij het typeren van een term. Kijken we naar de code (zowel hier als in de andere implementaties) dan zien we dat de nodige stappen altijd ongeveer dezelfde zijn:

- We schrijven zelf een boolean functie voor wat values zijn (de calculi zijn hier altijd in call-by-value geïmplementeerd)
- Indien we typeren met onze environment, hebben we een functie nodig die het type van een variabele uit dit environment kan halen. Deze functie moet ook falen als de meegegeven informatie uit het heterogene natuurlijke getal niet overeenstemt met de binding van variabelen in de term.
- Hierna volgt dan gewoon de implementatie van de evaluatie en typering.

5.2 Fi+

Types	A,B,C ::= Int $\mid \top \mid \bot \mid$ A$\rightarrow$ B $\mid$ A & B $\mid$ { l : A} $\mid \alpha \mid \forall.(\alpha * \text{A}).\text{B}$
Expressions	e::= x $\mid$ i $\mid \top \mid \lambda x.E \mid E_1 E_2 \mid \text{E}_1, , E_2 \mid E : A \mid \{l = E\}$ $\mid$ E.l $\mid \Lambda(\alpha * \text{A}).$ E $\mid$ E A
Term contexts	$\Gamma ::= \bullet \mid \Gamma, x : \text{A}$
Type contexts	$\Delta ::= \bullet \mid \Delta, \alpha * \text{A}$

Figuur 5.4: Fi+

5.2.1 Achtergrond calculus

Fi+ [10] is een taal die betekenisvol is, omdat we ze kunnen laten elaboreren naar FCo. Elaboreren betekent hier dat in plaats van de taal rechtstreeks uit te voeren, we ze omzetten naar een andere calculus. Een term in Fi+ is dus betekenisvol als we ze naar FCo kunnen omzetten. Een omzetting kan indien de Fi+-term een typering heeft. De Fi+-calculus was ontworpen voor het bestuderen van zeer modulaire programmeertalen.

5.2.2 Specificatie en Gegenereerde Code

De specificatie is te vinden in Figuur 5.5. Deze is vrijwel rechtstreeks een omzetting van Figuur 5.4. De calculus gebruikt geen andere methodes dan die al besproken in de vorige sectie voor system F met coercions. Het enige verschil is dat er bij lambda abstracties in de specificatie wel een type annotering is en in de originele calculus niet.

5.2.3 Prototype

Een implementatie is al eens geschreven geweest voor deze taal, en is te vinden op https://github.com/bixuanzju/ESOP2019-artifact [4]. Die implementatie was op basis van de unbound tool gemaakt [19]. Onze oplossing is ook voor zowel

```
namespace TermVar : FiTerm,FiType
namespace TypeVar : FiType,FiType

sort FiTerm
    inh ctx TermVar
    inh tyctx TypeVar
    |TmVar (x@ctx)
    |TmInt {Int}
    |TmTop
    |TmAbs (t:FiTerm) (ty:FiType) [x:TermVar]
           t.ctx=lhs.ctx,x
    |TmApp (t1:FiTerm) (t2:FiTerm)
    |TmMerge (t1:FiTerm) (t2:FiTerm)
    |TmAnn (t:FiTerm) (ty:FiType)
    |TmRecord (t:FiTerm) {String}
    |TmProj (t:FiTerm) {String}
    |TmAll (ty:FiType) (t:FiTerm) [alpha:TypeVar]
           t.tyctx=lhs.tyctx,alpha
    |TypeApp (t:FiTerm) (ty:FiType)

sort FiType
    inh tyctx TypeVar
    |TyTop
    |TyBot
    |TyInt
    |TyArr (ty1:FiType) (ty2:FiType)
    |TyAnd (ty1:FiType) (ty2:FiType)
    |TyRecord (ty:FiType) {String}
    |TyVar(x@tyctx)
    |TyAll (tyStar:FiType) (ty:FiType) [alpha:TypeVar]
              ty.tyctx= lhs.tyctx,alpha
```

Figuur 5.5: Specificatie van Fi+ in de tool

elaboratie naar System F met coercions (hierna FCo genoemd) als typing. Het enige verschil is dat onze vorige implementatie van system FCo extra informatie in de coercions bevat. De elaboratie gebeurt dus niet naar onze eigen FCo implementatie maar naar de vorm zoals ze origineel in de paper beschreven staat.

De Aut'Boundtool is bij elaboratie minder een meerwaarde. Dit wegens het feit dat het enige geval waar een substitutie nodig is, bij het typeren van een term is. Bij de elaboratie wordt er helemaal nergens een substitutie toegepast. In het geval van elaboreren naar een andere calculus is onze tool dus niet zo nuttig. Er valt weinig bij deze calculus op te merken qua variabelebinding dat interessant is om te bespreken.

5.3 Explicit Effects

5.3.1 Achtergrond calculus

Terms	
Value	v ::= x | unit | fun (x : T) $\rightarrow$ c | h
	| $\Lambda\zeta$. v | $v\ \tau$ | $\Lambda\alpha : \tau$. v | v T | $\Lambda\delta$. v | v Δ
	| $\Lambda\omega : \pi$. v | v γ | v $\triangleright\ \gamma$
Handler	h::= {return (x:T) $\rightarrow c_r$, Op_1 xk $\rightarrow c_{Op1}$,...,Op_n xk $\rightarrow c_{Opn}$}
Computation	c::= return v | Op v (y : T. c) | do x $\leftarrow c_1$; c_2
	| handle c with v | $v_1 v_2$ | let x=v in c | c $\triangleright\ \gamma$
Types	
Skeletons	τ::= ζ | Unit | $\tau \rightarrow \tau$ | $\tau \Rrightarrow \tau$ | $\forall$. τ
Value type	T::= α | Unit | T $\rightarrow$ C
	| $C_1 \Rrightarrow C_2$ | $\forall\zeta$. T | $\forall\alpha : \tau$. T | $\forall\delta$. T | $\pi \Rightarrow$ T
Simple coercion type	π::= $T_1 \ll T_2$ |$\Delta_1 \ll \Delta_2$
Coercion type	ρ ::= π | $C_1 \ll C_2$
Computation type	C ::= T ! Δ
Dirt	Δ ::=δ | $\emptyset$ | {Op} $\bigcup\ \Delta$
Coercion	γ ::= ω | $\langle$Unit$\rangle$ | $\langle\alpha\rangle$ | $\langle\Delta\rangle$ | $\gamma_1 \rightarrow \gamma_2$ | $\gamma_1 \Rrightarrow \gamma_2$
	| $\emptyset_\delta$ | { Op} $\bigcup\ \gamma$ | $\forall\zeta$. γ | $\Lambda\alpha : \tau$. γ
	| $\Lambda\delta$. γ | $\pi \Rightarrow \gamma$ | γ_1 ! γ_2

Figuur 5.6: ExEff

Hier bespreken we een interessante calculus [16] voor hetgene we wilden testen in dit werk. Deze calculus was origineel gedefinieerd voor het verbeteren van de compilatie van Eff-code naar ML-code.

5.3.2 Specificatie en Gegenereerde Code

Deze calculus bevat vijf verschillende namespaces. Dit groot aantal namespaces maakt het interessant voor onze tool op te testen. Het is namelijk interessant om te testen hoe de tool omgaat met zo'n groot aantal namespaces. Indien de tool een calculus zo groot als deze calculus zonder fouten kan genereren, dan geeft dit ons een groot vertrouwen in het nagenoeg foutloos werken van de tool. Het kunnen verwerken van een calculus die zo groot is als deze, geeft ons vertrouwen dat we voor vele talen foutloos code kunnen genereren. Bij het nader bekijken van de code gegenereerd voor deze taal, zien we dat er een groot aantal hulpfuncties constant meegegeven wordt. De specificatie is te vinden in de Appendix C.

```
handlermap onTermVar onTypeVar onSkelTypeVar onDirtVar onCoVar
c (ReturnHandler opsc ty cr) =
  ReturnHandler
    (map
       (operationCompTuplemap
          onTermVar
          onTypeVar
          onSkelTypeVar
          onDirtVar
          onCoVar
          (STermVar c))
       opsc)

coercioncoercionSubstitute sub orig t =
  rewriteCoercion $
  coercionmap
    (coercionSubstituteHelp sub orig)
    (\c x -> x)
    (\c x -> x)
    (\c x -> x)
    Z
    t
```

Figuur 5.7: Codevoorbeelden uit ExEff

De enige interessante zaken hier die nog niet in system F met coercions aan bod kwamen zijn:

- Het grote aantal functies dat meegegeven wordt wegens het groot aantal namespaces.
- Het gebruik van een lijststructuur. Dit toont ook de meerwaarde van de lijststructuur aan. Deze structuur telkens in elke specificatie zelf opnieuw schrijven zou inefficiënt zijn.
- Er wordt een rewrite operatie gebruikt omdat anders coercions in de calculus niet helemaal correct zouden blijven onder substitutie.

Als we kijken naar de bepaalde fragmenten kunnen we eruit halen dat we inderdaad weer de notatie voor voorgedefinieerde types kunnen gebruiken, de import statement en de rewrite operatie alsook het letterlijk invoegen van code. Ook de lijstnotatie is gebruikt.

5.3.3 Prototype

Bij dit prototype hebben we enkel de evaluatie geïmplementeerd. Deze is vrij uitgebreid wegens het grote aantal constructies in de taal. Maar dankzij het feit dat de substitutiecode gegenereerd is valt deze implementatie nog praktisch te doen. Ook is een groot deel van de code voor bijvoorbeeld de abstracties praktisch altijd hetzelfde, met als enige uitzondering dat enkel de introductie van de betreffende variabele verschilt.

5.4 Testmethode

De testmethode voor de calculi is als volgt. We implementeren de evaluatie- of typeringsregels. Daarna testen we hen op een beperkt aantal termen. Dit werkt omdat we met een de Bruijn variabelevoorstelling werken. Zoals al in eerdere hoofdstukken vermeld zorgt deze ervoor dat fouten in substitutie zeer snel zich laten tonen. We hebben vier tests bij ExEff, veertien testen bij Fi+ calculus en veertien testen system F met coercions. Dit lijkt op eerste zicht niet al te veel om de correctheid van de tool aan te tonen, maar eigenlijk zien we vrij snel als er een fout in de tool/specificatie optreedt wegens het structurele aspect van de substitutiecode. Daarom hoeven we niet grote aantallen code te schrijven voor testen.

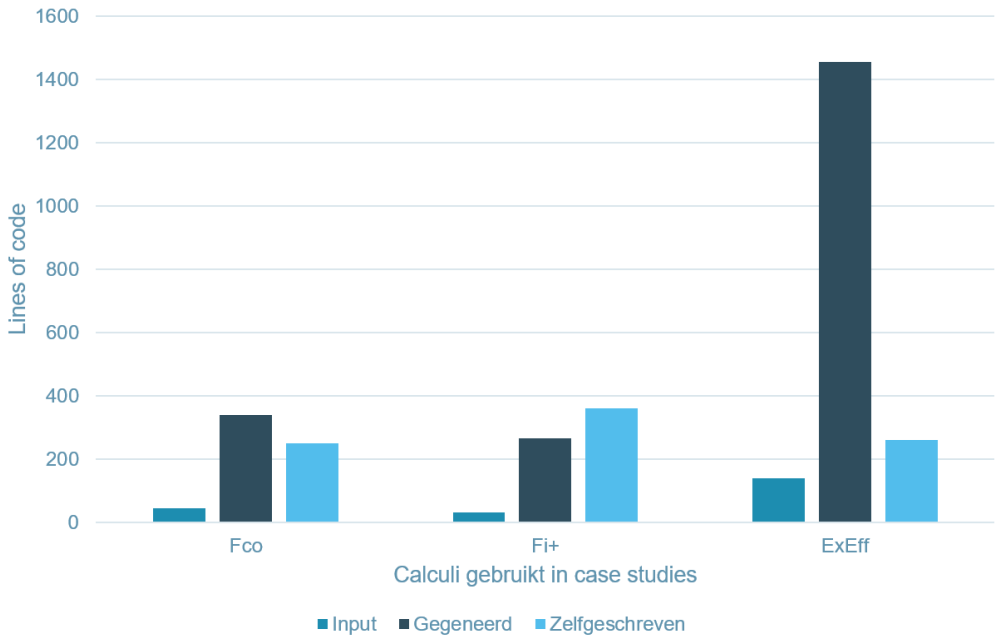
5.5 Codegeneratie

In dit stuk kijken we naar het aantal lijnen code dat we genereren en naar de computationele complexiteit van deze code.

calculi	**Specificatie**	**Gegenereerde**	**Zelfgeschreven**
`Fco`	43	340	250
`Fi+`	30	265	360
`ExEff`	140	1440	261

Tabel 5.1: Aantal lijnen horende bij specificatie, generatie en zelfgeschreven code

Zoals uit de figuur 5.8 blijkt, genereert onze code 8-10 aantal lijnen code per lijn uit de specificatie. Overigens valt op te merken dat onze specificatie qua input minder complex is dan de lijnen die gegenereerd zullen worden, zeker in de gevallen dat er veel verschillende soorten namespaces zijn. De tijdswinst die hierbij verkregen wordt is dus groot. Ook gezien de grote overeenkomst in Haskell tussen sorts en algebraïsche datatypes is het een vrij natuurlijke overgang tussen de twee talen. Het zou dus voor Haskellprogrammeurs niet zo moeilijk zijn om deze snel onder de knie te krijgen. Deze case studies zijn op calculi gebaseerd die effectief geformuleerd zijn in onderzoek[16, 20].



Figuur 5.8: Case Studies

5.5.1 Computationele performantie

We geven een overzicht van hoe performant de verschillende operaties zijn:

- De plus/minus/compare operaties voor heterogene natuurlijke getallen zijn lineair in de grootte. Meer formeel is dit dus O(n) met n de grootte van het heterogene getal.
- Aangezien de shiftoperatie over de term itereert is ze lineair in de grootte van de term. Wanneer de term variabelen bevat zal er hier dus ook de factor van plus/minus/compare een factor spelen. Meer formeel is dit dus een worst case complexiteit van $O(m * n)$. Dit is het geval waarbij alle eindtermen variabelen zijn met hun heterogene index een grootte van n.
- Substituties zijn lineair in de grootte van de termen, maar performantie is ook afhankelijk van de grootte van de term die in de andere gesubstitueerd wordt aangezien deze geshift moet worden. De worst case complexiteit is $O(m * n * b)$ waarbij m en n de grootte van de twee termen respectievelijk zijn bij substitutie en b het aantal binders in de term waarin gesubstitueerd wordt. Dit is dus ook gedeeltelijk gecorreleerd met de grootte van die term.

5.6 Experimenten

We hebben de volgende experimenten gedaan om na te gaan of onze ideëen omtrent computationele performantie correct zijn. Deze zijn te zien in Figuren 5.10, 5.11, 5.9,

5.15, 5.14 en 5.16. Verdere uitleg over de experimenten wordt hieronder uitgelicht.

5.6.1 Plus Operatie

Voor het experiment van de plus operatie hebben we telkens de term die toegevoegd moest worden met een factor 10 vergroot, de linkerterm lieten we onaangetast. Het resultaat is te zien in Figuur 5.9. We hadden vermeld dat de computationele performantie lineair was. Dit valt inderdaad te zien in de resultaten die we hier bekwamen.

5.6.2 Shifting Operatie

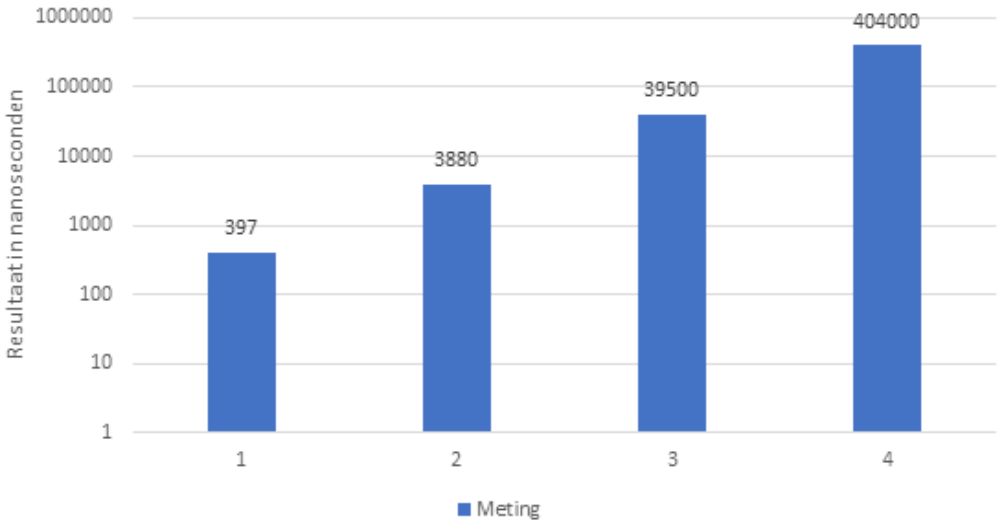
Voor het testen van de shiftoperatie lieten we termen genereren die steeds groter werden. Onze keuze voor een term die dit deed was een steeds dieper wordende if-term. Voorgesteld als een boomstructuur zou elke term een if-term zijn met als leaves variabelen. Aangezien een if-term van deze soort een grootte heeft van 3^n waarbij n de diepte is, verwachten we dat uit de figuren te zien zal zijn dat het evalueren hiervan met telkens ongeveer een factor 3 langer zal duren. Bij het bekijken van de resultaten bekomen uit Figuren 5.10 en 5.11, valt dit inderdaad te zien. Het verschil tussen Figuur 5.10 en Figuur 5.11 is overigens dat de grootte van de shift een factor 10^3 groter is in Figuur 5.10 dan in Figuur 5.11. Als we kijken naar de tijd in beide Figuren dan zien we dit inderdaad terugkomen.

5.6.3 Substitutie operatie

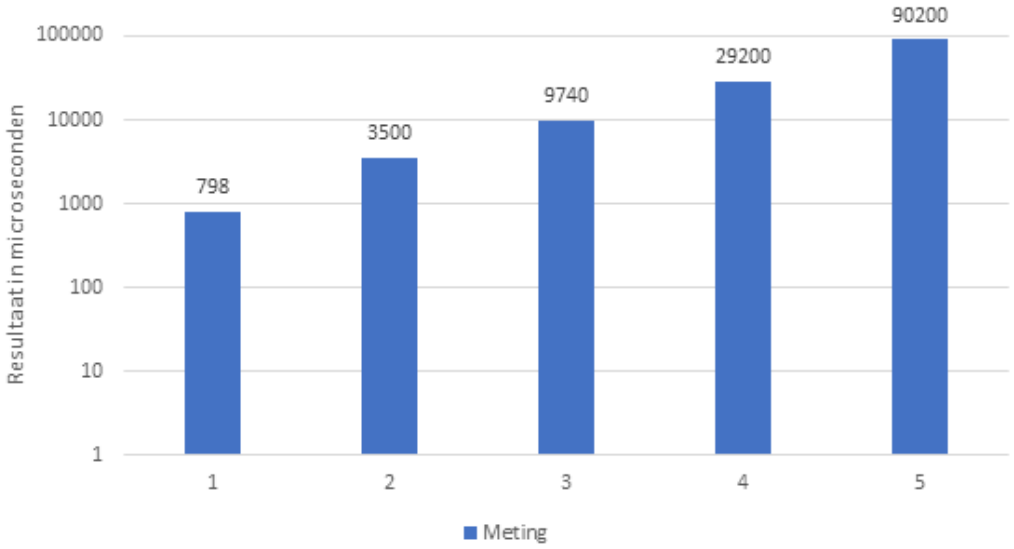
Eerst kijken we naar het effect van de grootte van de term die in de andere wordt gesubstitueerd terwijl we de grootte van de term waarin gesubstitueerd wordt laten variëren met een factor 9 per meting. In Figuur 5.12 is de term die in de andere gesubstitueerd wordt 3 maal groter dan in Figuur 5.13. Dit zien we terugkomen in de resultaten die we bekomen.
Verder testen we ook nog de invloed van de hoeveelheid waarmee we moeten shiften. Deze is te zien in Figuren 5.14, 5.15 en 5.16 . Het verschil tussen de twee metingen is dat in de respectievelijke figuren 10, 50 en 100 binders zijn die variabelen introduceren. Uit deze metingen blijkt dat het verband inderdaad lineair blijkt te zijn, waarbij de middelste figuur ongeveer halverwege de waarden lijkt te liggen tussen Figuur 5.14 en 5.16.

5.7 Samenvatting

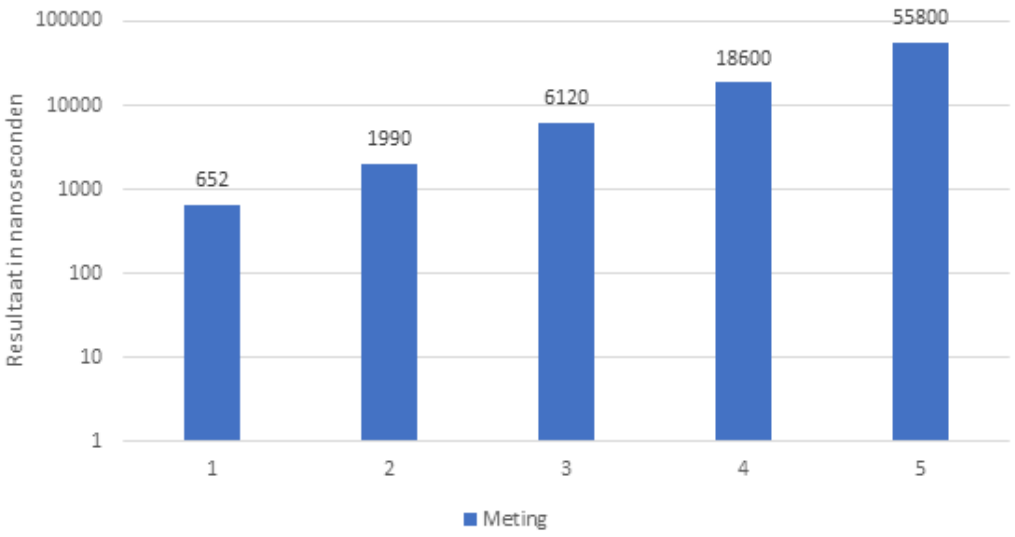
Uit dit hoofdstuk besluiten we dat er een nadeel is aan onze aanpak van een datatype te gebruiken in plaats van een primitief geheel getal dat ingebouwd is, gezien vanuit een puur computationeel oogpunt. De grootte van een specificatie zou er normaal nooit voor zorgen dat het genereren van een file lang zou duren, gezien dat gegenereerde tekst nooit meer dan enkele KB in beslag zal nemen.



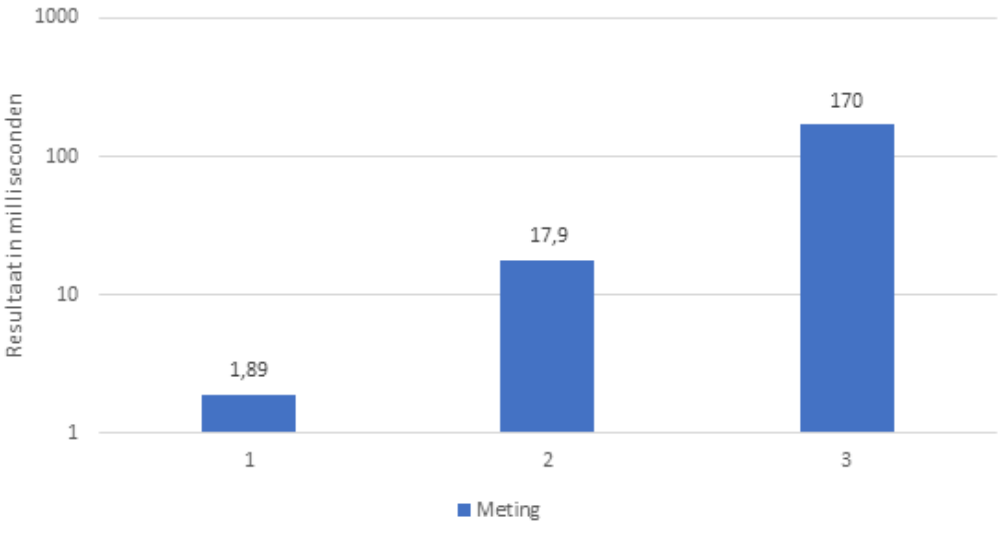
Figuur 5.9: Meting voor plus



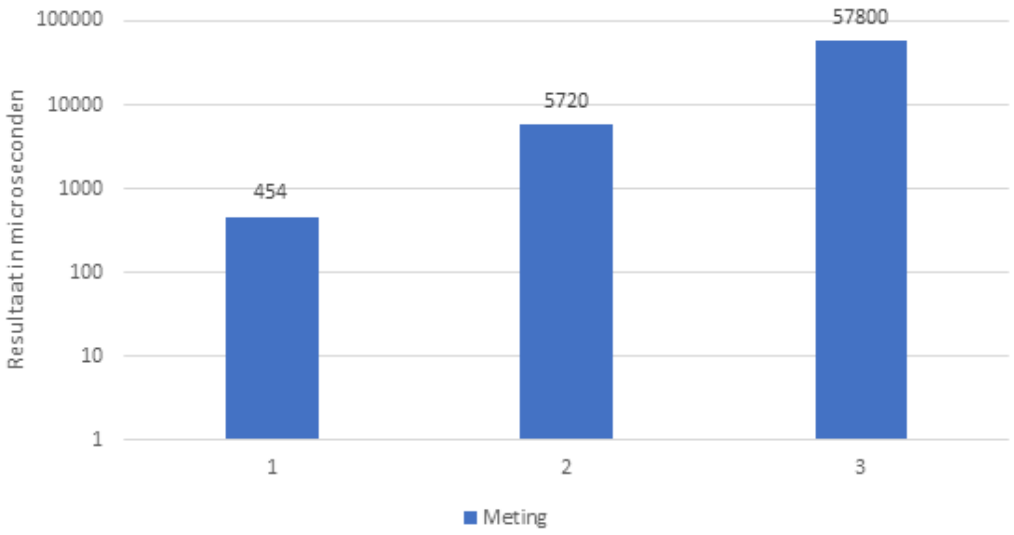
Figuur 5.10: Meting voor shift (grote variabelen)



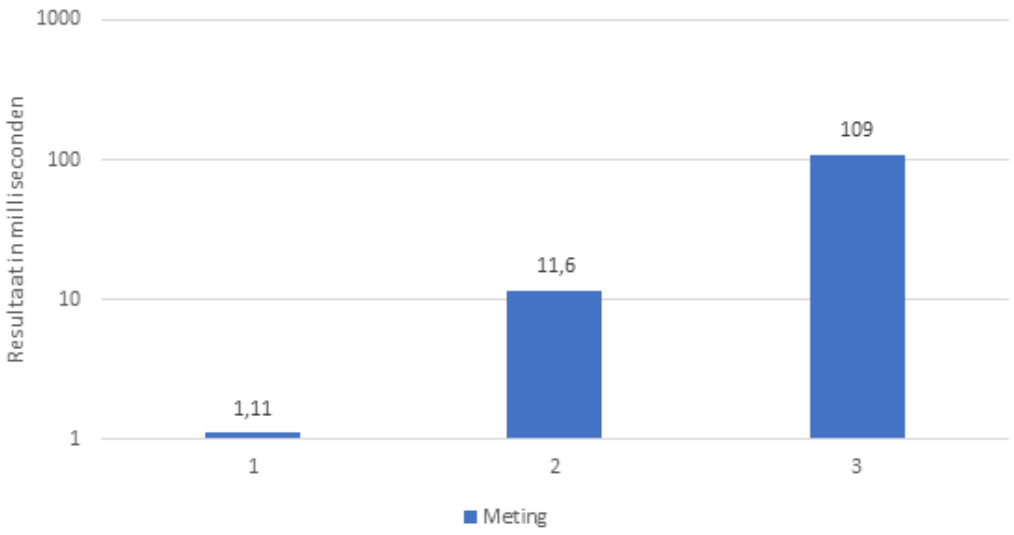
Figuur 5.11: Meting voor shift (kleine variabelen)



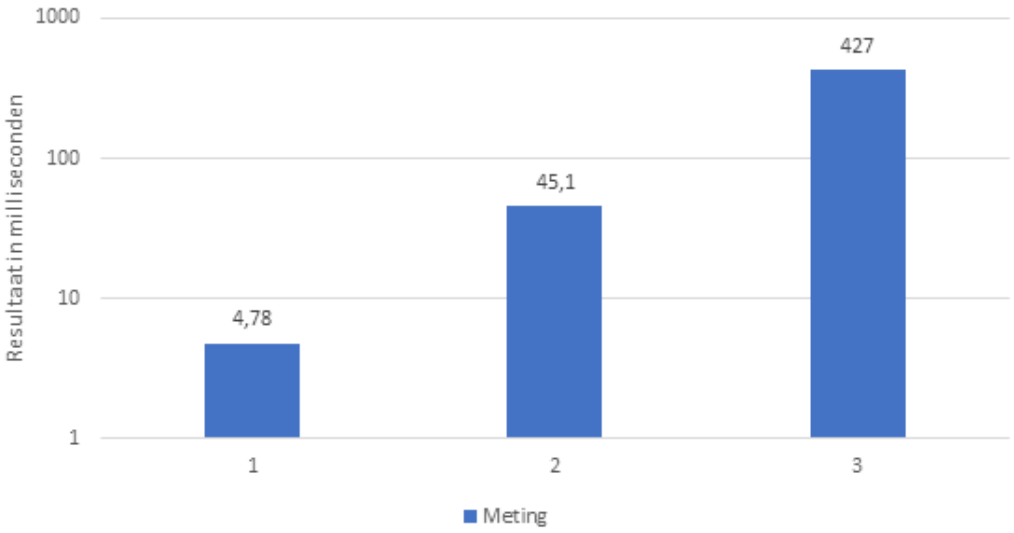
Figuur 5.12: Meting voor substitutie (grotere term)



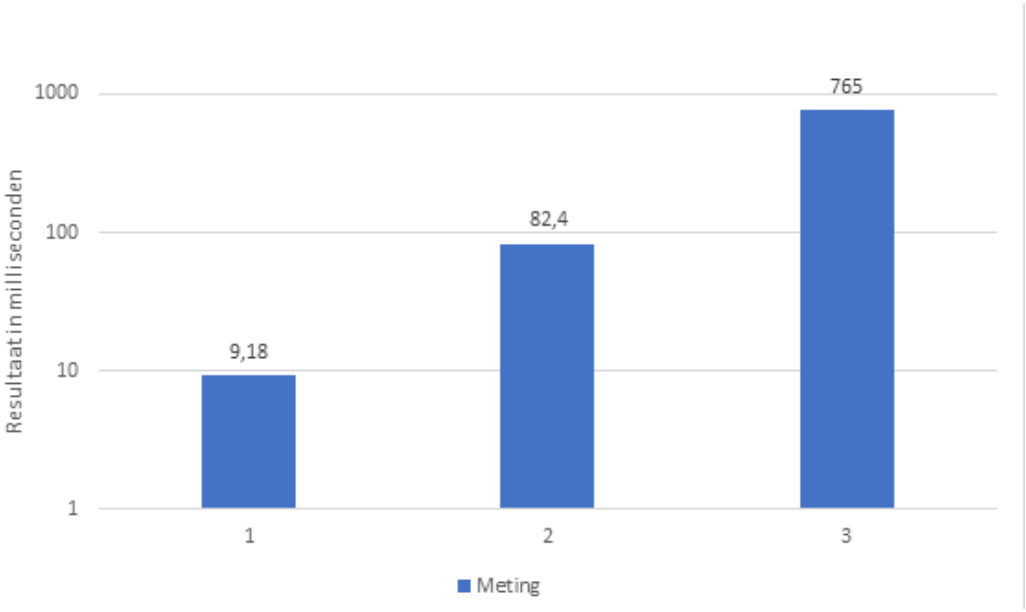
Figuur 5.13: Meting voor substitutie (kleinere shift)



Figuur 5.14: Meting voor substitutie (klein aantal binders)



Figuur 5.15: Meting voor substitutie (middelgroot aantal binders)



Figuur 5.16: Meting voor substitutie (groot aantal binders)

Overigens kunnen we ook opmerken dat we nooit synthesized contexts gebruikt hebben, maar gezien dat er in bijna geen enkele case study een binding meerdere variabelen introduceerde was dit ook niet nodig. Dit betekent niet dat synthesized constructs niet nuttig zijn. Ze zijn namelijk de enige manier waarop we meer dan 1 variabele tegelijkertijd kunnen introduceren. Zonder dit zouden we dus een hele hoop talen niet kunnen beschrijven. Onze inschatting van de computationele complexiteit van de operaties dat we voorzien lijkt overigens bevestigd te worden door onze metingen.

Hoofdstuk 6

Gerelateerd Werk

Dit hoofdstuk geeft een kort overzicht van gerelateerd werk dat in dit domein is gebeurd.

6.1 Unbound

Unbound [19][18] is een Haskell library geïmplementeerd met generalized algebraic datatypes. Het voorziet een manier om termen te substituteren zonder dat die code gegenereerd moet worden. Ze wordt namelijk op een zeer algemene manier geschreven. Het voordeel is dat men geen bijkomende specificatietaal moet gebruiken. De programmeur werkt enkel in Haskell.

6.2 Ott

Ott [17] is een tool die op basis van een specificatie zowel, LaTeX tekst en code kan genereren. Dit in een aantal talen, voornamelijk van proof assistants zoals coq [5]. Er is echter ook een mogelijkheid om ML code te genereren. De ML code is echter niet capture-avoiding.

6.3 Inbound

Inbound [14] is een voorafgaand werk dat ook werkt rond het mechaniseren van het binden van variabelen. Deze tool maakte gebruik van attribute grammmars en de UUAGC (Utrecht University Attribute Grammar Compilersystem [15]). Onze tool is hierop verder gebaseerd, aangezien de specificatietaal die we implementeerden hierop verderbouwde. Onze tool had echter wel extra mogelijkheden om zaken uit te drukken. Wij gebruikten ook geen attribute grammars of de UUAGC, maar werkten enkel in puur Haskell.

6.4 Samenvatting

Er is al vrij wat werk verricht in dit gebied om de specificatie van calculi makkelijker te laten verlopen. De aanpak waardoor dit werk zich onderscheidt van de andere werken, bevindt zich in het compleet naamloos maken van de tool. Hierdoor zijn zaken als alpha equivalentie en capture-binding substitution geen enkel probleem meer. Ook maakt de ontworpen tool gebruik van een specificatiefile zoals Ott [17] maar in plaats van de substituties te obfusceren zoals Unbound [19] genereren wij de nodige functies die de programmeur zelf kan gebruiken. Overigens, in tegenstelling tot Inbound [14], maakt deze tool geen gebruik van attribute grammars en is de oplossing in pure Haskell geschreven.

Hoofdstuk 7

Conclusie

Het doel van dit werk was om een tool te ontwikkelen die de code in verband met variabelebinding voor prototype-implementaties van calculi genereert.

Gebaseerd op Inbound, een specificatietaal al grotendeels geformuleerd in vorig onderzoek, hebbe we een uitgebreidere specificatietaal ontwikkeld, met heel wat nieuwe concepten om praktische en gevorderde calculi te modelleren.

De code in verband met variabelebinding wordt gegenereerd op basis van de specificatie die de scoping rules bevat. Deze code wordt modulair dan aan de programmeur beschikbaar gesteld. De programmeur kan dan de functionaliteit zoals substituties of het berekenen van de vrije variabelen van een term gebruiken.

Om het nut van de tool aan te tonen hebben we case studies uitgevoerd waarin we drie verschillende gevorderde calculi gemodelleerd hebben en prototype-implementaties gemaakt hebben van typecheckers en interpreters. Ook is de computationele complexiteit van de gegenereerde functies gemeten. De broncode van de verschillende calculi, de code voor metingen en de code van Aut'Bound zelf zijn te vinden op https://github.com/Belpaire/ASTTool.

7.1 Verder werk

Hier bespreken we enkele verdere mogelijke uitbreidingen van de Aut'Bound specificatietaal alsook enkele andere denkpistes die het bewandelen waard zijn.

7.1.1 Doorgedreven Haskell-integratie

In dit werk lezen we een Aut'Boundbestand uit en genereren we de Haskell code op basis hiervan. Dit heeft het nadeel dat pas tijdens runtime te zien valt dat een specificatie fout is. Het is mogelijk om in Haskell zelf de invoer te schrijven.

Een manier om dit te bewerkstelligen is door TemplateHaskell te gebruiken [8].

7.1.2 Mogelijke uitbreidingen

We geven enkele suggesties voor bijkomende ondersteuning bij het modelleren van calculi.

Rewrite

Een mogelijkheid zou zijn om ook een meer specifieke rewrite operatie toe te voegen die op enkel een specifieke constructor werkt. Momenteel werkt elke rewrite methode enkel op sort level.

Versoepeling van substitutie

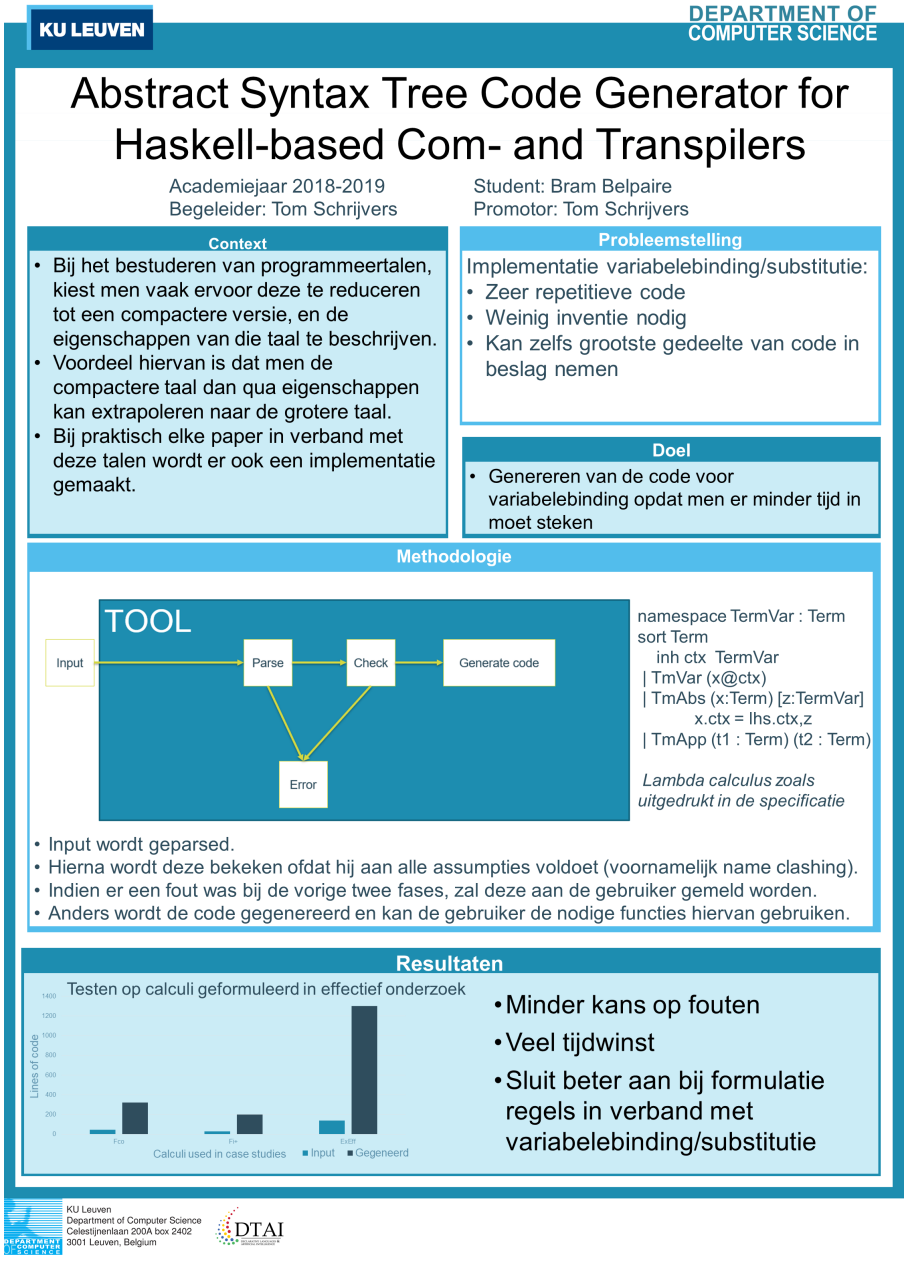
Momenteel verwacht de Aut'Boundtool dat bij een variabelesubstitutie altijd een variabele door een term van dezelfde soort wordt gesubstitueerd. Dit verhindert dat een variabele van een soort gesubstitueerd kan worden door een andere soort.In de meeste calculi vormt dit geen probleem. Termvariabelen substitueren in typevariabelen is immers niet wenselijk. Het zou echter handig zijn voor een aantal calculi om, met bijvoorbeeld een extra parameter, aan te duiden dat men dit gedrag anders wil. A-normal form bijvoorbeeld heeft expressies en waarden. Een waarde is in deze calculus een subset, van de expressies. Variabelen zijn dus waarden, maar het zou mogelijk kunnen zijn om expressies te substitueren voor deze variabelen. ANF in een grammaticavorm schrijven in Aut'Bound geeft geen substitutiefuncties die compleet correct zijn.

Bijlagen

Bijlage A

Poster

Figuur A.1: poster zoals ingediend op 26 april



Bijlage B

Gegenereerde code voor lambda calculus

```
module HaskellOutput
  ( Env(..)
  , HNat(..)
  , Term(..)
  , termtermSubstitute
  , freeVariablesTerm
  , termshiftplus
  , termshiftminus
  , generateHnatTermVar
  ) where

import Data.List

data Term
  = TmVar HNat
  | TmAbs Term
  | TmApp Term
          Term
  | TmTrue
  | TmFalse
  | TmIf Term
         Term
         Term
  deriving (Show, Eq)

data HNat
  = Z
  | STermVar HNat
  deriving (Show, Eq)
```

```
plus x1 (STermVar x2) = STermVar (plus x1 x2)
plus Z h = h
plus h Z = h

instance Ord HNat where
  compare (STermVar h1) (STermVar h2) = compare h1 h2
  compare Z Z = EQ
  compare Z _ = LT
  compare _ Z = GT

minus (STermVar h1) (STermVar h2) = minus h1 h2
minus Z Z = Z
minus Z _ = error " You cannot substract zero with a positive number
    "
minus result Z = result

data Env
  = Nil
  | ETermVar Env
  deriving (Show, Eq)

generateHnatTermVar 0 c = c
generateHnatTermVar n c = STermVar (generateHnatTermVar (n - 1) c)

termmap :: (HNat -> Term -> Term) -> HNat -> Term -> Term
termmap onTermVar c (TmVar hnat) = onTermVar c (TmVar hnat)
termmap onTermVar c (TmAbs x) = TmAbs (termmap onTermVar (STermVar c)
    x)
termmap onTermVar c (TmApp t1 t2) =
  TmApp (termmap onTermVar c t1) (termmap onTermVar c t2)
termmap onTermVar c (TmTrue) = TmTrue
termmap onTermVar c (TmFalse) = TmFalse
termmap onTermVar c (TmIf t t2 t3) =
  TmIf (termmap onTermVar c t) (termmap onTermVar c t2) (termmap
      onTermVar c t3)

termshiftHelpplus d c (TmVar hnat)
  | hnat >= c = TmVar (plus hnat d)
  | otherwise = TmVar hnat

termshiftplus :: HNat -> Term -> Term
termshiftplus d t = termmap (termshiftHelpplus d) Z t

termshiftHelpminus d c (TmVar hnat)
```

```
  | hnat >= c = TmVar (minus hnat d)
  | otherwise = TmVar hnat

termshiftminus :: HNat -> Term -> Term
termshiftminus d t = termmap (termshiftHelpminus d) Z t

termSubstituteHelp sub orig c (TmVar hnat)
  | hnat == plus orig c = termshiftplus c sub
  | otherwise = TmVar hnat

termtermSubstitute :: Term -> HNat -> Term -> Term
termtermSubstitute sub orig t = termmap (termSubstituteHelp sub orig)
    Z t

freeVariablesTerm :: HNat -> Term -> [HNat]
freeVariablesTerm c (TmVar hnat)
  | hnat >= c = [minus hnat c]
  | otherwise = []
freeVariablesTerm c (TmAbs x) = nub ((freeVariablesTerm (STermVar c)
   x))
freeVariablesTerm c (TmApp t1 t2) =
  nub ((freeVariablesTerm c t1) ++ (freeVariablesTerm c t2))
freeVariablesTerm c (TmTrue) = nub ([])
freeVariablesTerm c (TmFalse) = nub ([])
freeVariablesTerm c (TmIf t t2 t3) =
  nub
    ((freeVariablesTerm c t) ++
     (freeVariablesTerm c t2) ++ (freeVariablesTerm c t3))
```

Bijlage C

Complete specificatie van exeff

```
import (Operations)
namespace TermVar : Value , ValueType
namespace SkelTypeVar : SkeletonType
namespace DirtVar : Dirt
namespace TypeVar : ValueType , SkeletonType
namespace CoVar : Coercion , SimpleCoercionType

sort Value
    inh ctx TermVar
    inh tctx TypeVar
    inh skelctx SkelTypeVar
    inh dctx DirtVar
    inh coctx CoVar
 | TmVar (x@ctx)
 | TmFun (x:Computation) (t:ValueType) [z:TermVar]
           x.ctx = lhs.ctx,z
 | TmTSkellAbs (t1 : Value) [T:SkelTypeVar]
       t1.skelctx = lhs.skelctx, T
 | TmTSkelApp (t1 : Value) (ty : SkeletonType)
 | TmValueTypeAbs (t:Value) (ty:SkeletonType) [Z:TypeVar]
           t.tctx=lhs.tctx,Z
 | TmValueTypeApp (t:Value) (ty:ValueType)
 | TmDirtAbs (t:Value) [Z:DirtVar]
           t.dctx=lhs.dctx,Z
 | TmDirtApp (t:Value) (ty:Dirt)
 |TmCoAbs (t:Value) (coty:SimpleCoercionType) [Z:CoVar]
   t.coctx=lhs.coctx,Z
 | TmCoApp (t:Value) (Co:Coercion)
 | TmCast (val:Value) (Co:Coercion)
 |TmUnit
 |TmHandler (h:Handler)
```

```
sort OperationCompTuple
    inh ctx TermVar
    inh tctx TypeVar
    inh skelctx SkelTypeVar
    inh dctx DirtVar
    inh coctx CoVar
    |OpAndComp (comp:Computation) {Op} [k:TermVar]
           comp.ctx=lhs.ctx,k
sort Handler
    inh ctx TermVar
    inh tctx TypeVar
    inh skelctx SkelTypeVar
    inh dctx DirtVar
    inh coctx CoVar
    |ReturnHandler (ty:ValueType) (cr:Computation) (opsc:[
       OperationCompTuple]) [x:TermVar]
              cr.ctx=lhs.ctx,x
              opsc.ctx=lhs.ctx,x

sort Computation
    inh ctx TermVar
    inh tctx TypeVar
    inh skelctx SkelTypeVar
    inh dctx DirtVar
    inh coctx CoVar
    |ReturnComp (v:Value)
    |HandleComp (comp:Computation) (v:Value)
    |ComputationApp (t1 : Value) (t2 : Value)
    |LetComp (v:Value) (comp:Computation) [x:TermVar]
             comp.ctx=lhs.ctx,x
    |DoComp (c1:Computation) (c2:Computation) [x:TermVar]
              c2.ctx= lhs.ctx,x
    |CastComp (comp:Computation) (gamma:Coercion)
    |OpComp (v:Value) (ty:ValueType) (co:Computation) {Op}[y:TermVar]
           co.ctx=lhs.ctx,y

sort ValueType
    inh tctx TypeVar
    inh skelctx SkelTypeVar
    inh dctx DirtVar
    |ValTyVar (x@tctx)
    |ValTUnit
    |ValTyArr (ty:ValueType) (comp:ComputationType)
    |ValTyHandler (c1:ComputationType) (c2:ComputationType)
```

```
    |ValTyAllSkel (t:ValueType) [Z:SkelTypeVar]
           t.skelctx=lhs.skelctx,Z
    |ValTyAll (t:ValueType) (ty:SkeletonType) [Z:TypeVar]
           t.tctx=lhs.tctx,Z
    |ValTyAllDirt (t:ValueType) [Z:DirtVar]
       t.dctx=lhs.dctx,Z
    |ValTyCoArr (pi:SimpleCoercionType) (T:ValueType)
sort CoercionType
    inh tctx TypeVar
    inh skelctx SkelTypeVar
    inh dctx DirtVar
    | CoSimple (co: SimpleCoercionType)
    |CoComp (v1:ComputationType) (v2: ComputationType)

sort SimpleCoercionType
    inh tctx TypeVar
    inh skelctx SkelTypeVar
    inh dctx DirtVar
   | DirtCoTypes (d1:Dirt) (d2:Dirt)
   | ValTypes (d1:ValueType) (d2:ValueType)

sort SkeletonType
    inh skelctx SkelTypeVar
    |SkelTUnit
    |SkellAllType (t:SkeletonType) [T:SkelTypeVar]
           t.skelctx= lhs.skelctx,T
    |SkelVar (d@skelctx)
    |SkelArr (t1:SkeletonType) (t2:SkeletonType)
sort Dirt
    inh dctx DirtVar
    |DirtVariable (d@dctx)
    |EmptyDirt
    |UnionDirt (d:Dirt) {Op}

sort ComputationType
    inh tctx TypeVar
    inh skelctx SkelTypeVar
    inh dctx DirtVar
    | ComputationTy (t:ValueType) (d:Dirt)
sort Coercion rewrite
    inh coctx CoVar
    inh tctx TypeVar
    inh skelctx SkelTypeVar
    inh dctx DirtVar
    |CoercionVar(x@coctx)
```

```
    |CoUnit
    |CoTypeVar (valty:ValueType)
    |CoDirt (d:Dirt)
    |CoArrow (co1:Coercion) (co2:Coercion)
    |CoHandler (co1:Coercion) (co2:Coercion)
    |CoEmptyDirt
    |UnionOp (Co:Coercion) {Op}
    |CoskeletonAll (co:Coercion) [Z:SkelTypeVar]
           co.skelctx=lhs.skelctx,Z
    |CoTypeAll (co:Coercion) (t:SkeletonType) [Z:TypeVar]
              co.tctx=lhs.tctx,Z
    |CodirtAll (co: Coercion ) [Z:DirtVar]
           co.dctx=lhs.dctx,Z
    |CoCoArrow (pi: SimpleCoercionType) (Co:Coercion)
    |CoComputation (co1:Coercion) (co2:Coercion)


HaskellCode

rewriteTypeToCoercion:: ValueType ->Coercion
rewriteTypeToCoercion (ValTyVar hnat) = CoTypeVar (ValTyVar hnat)
rewriteTypeToCoercion (ValTUnit) = CoUnit
rewriteTypeToCoercion (ValTyArr ty1 (ComputationTy ty2 dirt))=
    CoArrow (rewriteTypeToCoercion ty1) (CoComputation (
    rewriteTypeToCoercion ty2) (CoDirt dirt))
rewriteTypeToCoercion (ValTyHandler (ComputationTy ty1 dirt1) (
    ComputationTy ty2 dirt2)) = CoHandler (CoComputation (
    rewriteTypeToCoercion ty1) (CoDirt dirt1)) (CoComputation (
    rewriteTypeToCoercion ty2) (CoDirt dirt2))
rewriteTypeToCoercion (ValTyAllSkel ty) = CoskeletonAll (
    rewriteTypeToCoercion ty)
rewriteTypeToCoercion (ValTyAll valty skel)= CoTypeAll (
    rewriteTypeToCoercion valty) skel
rewriteTypeToCoercion (ValTyAllDirt valty)=CodirtAll (
    rewriteTypeToCoercion valty )
rewriteTypeToCoercion (ValTyCoArr pi ty)= CoCoArrow pi (
    rewriteTypeToCoercion ty)


rewriteCoercion :: Coercion->Coercion
rewriteCoercion (CoTypeVar ty) = rewriteTypeToCoercion ty
rewriteCoercion (CoArrow co1 co2) = CoArrow (rewriteCoercion co1) (
    rewriteCoercion co2)
rewriteCoercion (CoHandler co1 co2) = CoHandler (rewriteCoercion co1)
     (rewriteCoercion co2)
```

```
rewriteCoercion (UnionOp co1 op) = UnionOp (rewriteCoercion co1) op
rewriteCoercion (CoskeletonAll co) = CoskeletonAll (rewriteCoercion
    co)
rewriteCoercion (CoTypeAll co skel) = CoTypeAll (rewriteCoercion co )
     skel
rewriteCoercion (CoCoArrow pi co) = CoCoArrow pi (rewriteCoercion co
    )
rewriteCoercion (CoComputation co1 co2) = CoComputation (
    rewriteCoercion co1) (rewriteCoercion co2)
rewriteCoercion co = co
```

Bibliografie

[1] URL: https://gitlab.haskell.org/ghc/ghc/wikis/commentary/compiler/core-syn-type, last checked on 2019-19-7.

[2] URL: https://www.haskell.org/, last checked on 2019-04-08.

[3] URL: https://gist.github.com/CMCDragonkai/9f5f75118dda10131764, last checked on 2019-30-5.

[4] URL: https://github.com/bixuanzju/ESOP2019-artifact, last checked on 2019-10-05.

[5] coq. URL: https://coq.inria.fr/, last checked on 2018-20-12.

[6] Hackage data.foldable. URL: http://hackage.haskell.org/package/base-4.12.0.0/docs/Data-Foldable.html, last checked on 2019-06-1.

[7] Parsec library. URL: http://hackage.haskell.org/package/parsec, last checked on 2019-29-04.

[8] Template haskell tutorial. URL: https://wiki.haskell.org/A_practical_Template_Haskell_Tutorial, last checked on 2019-06-1.

[9] P. Benjamin, C. *Types and Programming Languages*. The MIT Press, 2002. ISBN-13: 978-0-262-16209-8.

[10] X. Bi, N. Xie, B. C. d. S. Oliveira, and T. Schrijvers. Distributive disjoint polymorphism for compositional programming.

[11] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *ACM Sigplan Notices*, volume 28, pages 237–247. ACM, 1993.

[12] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of programming languages*. MIT press, 2001.

[13] R. Grigore. Java generics are turing complete. In *ACM SIGPLAN Notices*, volume 52, pages 73–85. ACM, 2017.

[14] S. Keuchel and T. Schrijvers. Inbound: Simple yet powerful specification of syntax with binders.

[15] P. R. A. A. S. D. Swierstra and J. a. Saraiva.

[16] A. H. Saleh, G. Karachalias, M. Pretnar, and T. Schrijvers. Explicit effect subtyping. In *European Symposium on Programming*, pages 327–354. Springer, Cham, 2018.

[17] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, et al. Ott: Effective tool support for the working semanticist. *Journal of functional programming*, 20(1):71–122, 2010.

[18] S. Weirich. Binders unbound tutorial. URL: https://github.com/sweirich/replib/blob/master/Unbound/tutorial/Tutorial.lhs, last checked on 2018-19-12.

[19] S. Weirich, B. A. Yorgey, and T. Sheard. Binders unbound. In *ACM SIGPLAN Notices*, volume 46, pages 333–345. ACM, 2011.

[20] B. Xuan, X. Ningning, T. Schrijvers, et al. Distributive disjoint polymorphism for compositional programming. *Lecture Notes in Computer Science*, 2019.

Fiche masterproef

Student: Bram Belpaire

Titel: Abstract Syntax Tree Code Generator for Haskell-based Com- and Transpilers

UDC: 681.3

Korte inhoud:
Bij het schrijven van calculi wordt vrij veel tijd verloren aan het schrijven van de code rond variabelebinding. Er zijn al tools hierrond maar de meeste gebruiken geen naamloze representatie. De tool voorzien in dit werk doet dit wel. We zien dat in dit onderzoeksdomein al verschillende manieren zijn gevonden voor het presenteren van de info nodig om de code te laten genereren. Wij werken verder gebaseerd op een van deze implementaties namelijk Inbound [14]. We nemen echter een andere aanpak bij het produceren van de effectieve code. Ook zorgen we voor meer mogelijkheden dan er waren in de originele taal. Deze aanpak werd geschreven in de functionele taal Haskell en is te vinden op https://github.com/Belpaire/ASTTool.

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

Promotor: Prof. dr. ir. Tom Schrijvers

Assessoren: Prof. dr. ir. Bart De Decker
Prof. dr. ir. Gerda Janssens

Begeleider: