# De lokale ⅄-calculus

Doctoraalscriptie Cognitieve Kunstmatige Intelligentie
Universiteit Utrecht

Marijn Zwitserlood

9 augustus 2007

$1^{ste}$ begeleider: Vincent van Oostrom

# Voorwoord

Het schrijven van deze scriptie is een langdurig project geworden. Ik wil daarom mijn beide begeleiders, Vincent van Oostrom en Doaitse Swierstra, heel hartelijk danken omdat ze steeds maar weer samen met mij, soms na tussenpozen van meer dan een half jaar, het project hebben willen oppakken en uiteindelijk mijn scriptie hebben willen nakijken. Zeker Vincent van Oostrom is erg belangrijk geweest. Zonder zijn enthousiasme voor het bereik van de $\lambda$-operator en zijn blinde vertrouwen in mijn afstudeercapaciteiten, was dit project waarschijnlijk niet tot een goed einde gekomen. Ik wil ook graag mijn derde beoordelaar, Dimitri Hendriks, bedanken voor het lezen van mijn scriptie en het geven van veel nuttige aanwijzingen ter verbetering daarvan. Natuurlijk wil ik ook mijn vriendin, Marije, bedanken, die me al die tijd veel liefde en steun heeft gegeven, en mijn vader, voor al zijn (financiële) steun. Tot slot wil ik nog mijn vrienden en mede-studiegenoten bedanken, met name Alain en Huib, die het volgen van colleges en werkgroepen altijd een stuk gezelliger maakten.

# Inhoudsopgave


**1 Inleiding** **5**

1.1 De $\lambda$-calculus . . . 7
1.2 Optimaliteit . . . 8
1.3 Efficiëntie . . . 10
1.4 De $\reflectbox{$\lambda$}$-operator . . . 11
1.5 Lokaliteit . . . 12
1.6 Notatie . . . 12

**2 De $\lambda$-calculus en de $\reflectbox{$\lambda$}$-calculus** **14**

2.1 De $\lambda$-calculus . . . 15
2.1.1 Syntax . . . 15
2.1.2 Semantiek . . . 16
2.1.3 $\alpha$-conversie . . . 17
2.1.4 $\beta$-reductie . . . 17
2.1.5 Substitutie . . . 18
2.1.6 Gesloten termen . . . 19
2.2 De $\reflectbox{$\lambda$}$-calculus . . . 20
2.2.1 Syntax . . . 20
2.2.2 De betekenis van de $\reflectbox{$\lambda$}$-operator - een voorbeeld . . . 21
2.2.3 Correspondentie tussen $\lambda$- en $\reflectbox{$\lambda$}$-termen . . . 22
2.2.4 Termen in balans . . . 23
2.2.5 De Bruijn-notatie . . . 25
2.2.6 $\alpha$-conversie . . . 26
2.2.7 $\beta$-reductie . . . 27

**3 Van termen naar grafen** **30**

3.1 Een voorbeeld - Naïeve vertaalfunctie . . . 30
3.1.1 De abstracte syntax-boom . . . 31
3.1.2 Lokaliteit . . . 32
3.1.3 Grafen in balans . . . 32
3.1.4 De vertaalfunctie aangepast . . . 33
3.2 De grafen . . . 34
3.3 Een recursieve definitie van de vertaalfunctie . . . 36
3.3.1 De variabele . . . 36
3.3.2 De variabele, ongebalanceerd . . . 37
3.3.3 De $\lambda$-stap . . . 39
3.3.4 De applicatie-stap . . . 40
3.3.5 De vertaalfunctie gedefinieerd . . . 41
3.4 De vertaling van twee $\alpha$-equivalente termen . . . 44

**4 Van grafen naar termen** **45**
4.1 Read-back aan de hand van een voorbeeld . . . . . . . . . . . . . 46
4.2 De machine in het algemeen . . . . . . . . . . . . . . . . . . . . . 49
4.2.1 Welgevormde- en onwelgevormde grafen . . . . . . . . . . 51
4.3 Heen en terug vertalen geeft $\alpha$-equivalentie . . . . . . . . . . . . 52

**5 De herschrijfregels** **61**
5.1 *Beta*-reductie . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
5.1.1 Geen gebonden variabelen . . . . . . . . . . . . . . . . . . 63
5.1.2 De read-back machine uitgebreid . . . . . . . . . . . . . . 64
5.2 Het pushen van een end-of-scopeknoop . . . . . . . . . . . . . . . 65
5.2.1 De read-back aangepast . . . . . . . . . . . . . . . . . . . 67
5.3 Het pushen van de (re-)open-scopeknoop . . . . . . . . . . . . . . 70
5.4 Annihilatie en wisseling van scopeknopen . . . . . . . . . . . . . 70

**6 Correctheid van de herschrijfregels** **72**
6.1 De $x$-regels . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
6.2 Gebalanceerde en transparante termen . . . . . . . . . . . . . . . 75
6.3 *Beta* en $\beta$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
6.3.1 Graaf in balans na *Beta*-stap . . . . . . . . . . . . . . . . 93

**7 Efficiëntie** **94**
7.1 $\lambda x$-calculus . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95
7.1.1 Kosten . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95
7.1.2 Wat is efficiënter . . . . . . . . . . . . . . . . . . . . . . . 100
7.2 Optimaliteit . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101
7.2.1 Combinatoren . . . . . . . . . . . . . . . . . . . . . . . . . 101
7.2.2 Supercombinatoren . . . . . . . . . . . . . . . . . . . . . . 103
7.2.3 Sharing . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104
7.2.4 Combinatoren zijn niet optimaal . . . . . . . . . . . . . . . 107
7.2.5 Wat is beter? . . . . . . . . . . . . . . . . . . . . . . . . . 111

**8 Conclusie** **112**
8.1 Gerelateerd werk . . . . . . . . . . . . . . . . . . . . . . . . . . . 113

# 1 Inleiding

In de Cognitieve Kunstmatige Intelligentie (CKI) [11] gaat het om de vraag hoe je kennis in de computer kunt stoppen en hoe je die computer met die kennis kunt laten omgaan. In de CKI bestudeert men:

- De structuur van kennis;
- Het verwerven, representeren en opslaan van kennis;
- Het bewerken van de kennis en het redeneren ermee en erover.

Daarom is CKI een interdisciplinaire studie. Wat kennis is en hoe de structuur daarvan er uit ziet wordt bestudeerd in de filosofie. Hoe we kennis verwerven is een onderwerp voor psychologen. Het redeneren en bewerken van kennis is een onderwerp voor de logica en de taalkunde. Hoe we de kennis zo representeren dat een computer er mee kan werken, is een onderwerp voor de informatica. Hierbij wordt gebruik gemaakt van hulpmiddelen als programmeertalen.

In deze scriptie bestuderen we de $\lambda$-calculus. Enige kennis van de $\lambda$-calculus bij de lezer is daardoor wel gewenst. Ik geef een introductie op $\lambda$-calculus in Paragraaf 2.1), maar die is noodgedwongen zeer beknopt en dient meer om het geheugen op te frissen. Meer over de $\lambda$-calculus is te lezen in bijvoorbeeld [3].

De $\lambda$-calculus is te beschouwen als de meest basale functionele programmeertaal. Door deze calculus te onderzoeken, onderzoeken we een aantal basisprincipes van alle functionele programmeertalen. De $\lambda$-calculus is een onderwerp dat wordt bestudeerd in de informatica. Door zijn eenvoud en elegantie is hij echter ook zeer geliefd bij de computationele taalkundigen, die het gebruiken bij het implementeren van formele semantiek. In de $\lambda$-calculus kunnen we kennis representeren als $\lambda$-termen. Deze kennis kunnen we vervolgens bewerken door herschrijfregels toe te passen op de $\lambda$-termen. De $\lambda$-calculus is als programmeertaal erg onpraktisch want de $\lambda$-termen worden snel erg groot en onoverzichtelijk. Het toepassen van de herschrijfregels kan erg inefficiënt zijn. Als ze in bepaalde volgordes worden toegepast kan er werk worden verdubbeld.

Er zijn al oplossingen gevonden om de $\lambda$-calculus efficiënter te maken. De supercombinatoren [6] zijn hier een goed voorbeeld van. Hierbij worden $\lambda$-termen anders gerepresenteerd zodat ze makkelijker kunnen worden bewerkt. Bij deze methode gaat de structuur van de $\lambda$-expressies grotendeels verloren en zodoende ook het inzicht in wat er precies gebeurt. We proberen in deze scriptie meer inzicht te verkrijgen in de structuur van de $\lambda$-calculus en hoe deze zich gedraagt tijdens het toepassen van herschrijfregels. We zullen bekijken of we de $\lambda$-calculus anders kunnen representeren zodat we wel efficiëntere herschrijfregels krijgen, maar op zo'n manier dat de structuur van de $\lambda$-calculus zichtbaar blijft. We zijn meer geïnteresseerd in de structuur van de $\lambda$-calculus en minder in een praktische implementatie van de $\lambda$-calculus. Hoewel we natuurlijk wel hopen dat een implementatie gebaseerd op de principes uit deze scriptie mogelijk is.

In het verband met CKI houdt deze scriptie zich dus bezig met het opslaan en representeren van kennis. Het gaat er in dit geval dus om hoe we de kennis zo

representeren dat hij makkelijker te bewerken valt. We zullen ons met name op het vakgebied informatica begeven, omdat we een aantal aspecten van functionele programmeertalen onderzoeken. De scriptie heeft echter een beschouwend karakter omdat ons doel eerder is meer inzicht te krijgen in deze aspecten dan een zo efficiënt mogelijke oplossing te vinden.

De focus van deze scriptie zal liggen op de volgende aspecten: het bereik (of de *scope*) van de $\lambda$-operator en op lokaliteit van de herschrijfregels. In $\lambda$-calculus wordt geen expliciete informatie bijgehouden over het einde van de scope van een $\lambda$-operator. Deze wordt impliciet altijd zo ver mogelijk naar beneden doorgetrokken. Soms is het echter nuttig om expliciete informatie over deze scopes te hebben.

**Voorbeeld 1 (Het nut van scope-informatie)**

$(\lambda y\ M)N$ *waarbij* $y$ *niet vrij voorkomt in* $M$

*In dit voorbeeld loopt de scope impliciet over* $M$*. Als we* $\beta$*-reductie uitvoeren, dan moeten we* $N$ *substitueren voor* $y$ *en om dat te doen moeten we heel* $M$ *afzoeken. In dit geval zitten er geen vrije variabelen met de naam* $y$ *in* $M$ *en we doen dus al het werk voor niets. We kunnen in de* $\lambda$*-calculus niet aangeven dat er geen vrije variabele met de naam* $y$ *in* $M$ *voorkomt.*

In het werk van V. van Oostrom en D. Hendriks [5] wordt de $\lambda$-calculus uitgebreid met een end-of-scope-operator, de ʎ-operator, die het einde van de scope van de $\lambda$-operator aangeeft. Deze uitgebreide variant van de $\lambda$-calculus noemen we de ʎ-calculus (ʎ spreken we uit als *admal*). Een nadeel van deze calculus is dat deze niet volledig lokaal is. Dat wil zeggen dat er geen bovengrens zit aan de grootte van het gedeelte van een term dat geïnspecteerd moet worden om te zien of het een redex vormt. Het doel van deze scriptie is deze calculus te verfijnen zodat de calculus wel lokaal wordt. Dit zullen we doen door een graafrepresentatie voor de calculus te geven. De herschrijfregels die we bij deze graafrepresentatie definiëren zijn lokaal, zodat we altijd snel kunnen bepalen of een regel toe te passen is of niet. Daarna bewijzen we dat ons systeem een correcte representatie van de $\lambda$-calculus is.

Deze graafcalculus met expliciete scope informatie kunnen we uitbreiden tot een optimaal herschrijfsysteem. In Asperti en Guerrinni [2] wordt aangetoond dat een optimaal herschrijfsysteem kan worden verkregen door het toevoegen van een sharing operator aan een lokale graafrepresentatie van $\lambda$-calculus . In [9] wordt aangetoond dat het systeem dat in deze scriptie staat door de aanwezigheid van een scope-operator kan worden uitgebreid tot een optimaal herschrijfsysteem. Deze scriptie houdt zich enkel bezig met de toevoeging van een scope operator aan een graafherschrijfsysteem en kan dus ook worden gezien als een voorstudie voor een optimaal graafherschrijfsysteem waar ook de sharing operator aan is toegevoegd (zoals beschreven in [9] en [10]).

De scriptie ziet er verder als volgt uit:

- In de rest van dit hoofdstuk leg ik uit wat het nut is van $\lambda$-calculus in het algemeen en van een optimale, lokale, grafische representatie van $\lambda$-calculus met expliciete scope informatie. Ik ga dieper in op de begrippen optimaliteit, efficiëntie en lokaliteit. Tot slot zal ik enkele notatievormen bespreken die in deze scriptie voorkomen.

- In Hoofdstuk 2 introduceer ik in het kort de $\lambda$-calculus en geef ik een samenvatting van [5], waarin $\lambda$-calculus naar $\reflectbox{$\lambda$}$-calculus wordt uitgebreid.

- In Hoofdstuk 3 definiëer ik een lokale graafrepresentatie van de $\reflectbox{$\lambda$}$-calculus en een vertaalfunctie die normale $\lambda$-termen naar deze grafische representatie vertaald.

- In Hoofdstuk 4 leg ik uit hoe we met behulp van context semantiek de grafen uit Hoofdstuk 3 weer kunnen terugvertalen naar termen. Door deze twee vertaaloperaties hebben we de correspondentie tussen termen en grafen gedefinieerd.

- In Hoofdstuk 5 zal ik een verzameling herschrijfregels definiëren, waarmee de grafen kunnen worden herschreven. Deze herschrijfregels zullen lokaal zijn. Zij zullen op het niveau van de grafische representatie hetzelfde doen als de $\beta$-reductie op het niveau van de termen. Dus:

$$\begin{array}{ccc} t_1 & \xrightarrow[\beta]{} & t_2 \\ \big\downarrow & & \big\downarrow \\ g_1 & \twoheadrightarrow & g_2 \\ & \text{Graafherschrijfregel} & \end{array}$$

- In Hoofdstuk 6 zal ik bewijzen dat de herschrijfregels uit Hoofdstuk 5 correct werken.

- In Hoofdstuk 7 zal ik ingaan op de verschillen tussen optimaliteit en efficiëntie. Ik zal het het systeem qua efficiëntie vergelijken met andere systemen. Ook zal ik een globale beschrijving geven van het herschrijfsysteem, uitgebreid met sharing en dit vergelijken met andere herschrijfsystemen.

## 1.1 De $\lambda$-calculus

De $\lambda$-calculus [3] bestaat uit termen en regels om de termen te herschrijven. De grammatica waarmee de termen worden opgebouwd is vrij eenvoudig. Toch kunnen we in de $\lambda$-calculus alle berekenbare functies uitdrukken. Dat wil niet zeggen dat we de $\lambda$-calculus als programmeertaal moeten gebruiken. Omdat de grammatica zo simpel is, heb je vaak grote termen nodig om simpele dingen uit te drukken. Dit maakt het minder geschikt als programmeertaal. Toch is het nuttig om de $\lambda$-calculus te bestuderen, want we komen een aantal problemen van programmeertalen in de meest pure vorm tegen in de $\lambda$-calculus. Bijvoorbeeld problemen rond variabelenbinding en het bereik van bepaalde operatoren. Door

het bestuderen van de $\lambda$-calculus kunnen we dus meer te weten komen over die problemen en de oplossingen die we vinden, gebruiken voor programmeertalen.

## 1.2 Optimaliteit

In de $\lambda$-calculus zijn er vaak meerdere reductiestrategieën die een term naar zijn normaalvorm kunnen reduceren. Het is moeilijk te bepalen welke strategie de beste is. We kunnen er bijvoorbeeld voor zorgen dat we altijd de *leftmost* redex eerst reduceren en zo de term van links naar rechts herschrijven. Deze strategie wordt de *normal order reduction strategy* genoemd. Als we deze strategie toepassen, blijkt dat we vaak dubbel werk doen. Zie bijvoorbeeld de volgende term:

**Voorbeeld 2 (Normal order reduction)**
*Met* $I = \lambda x\ x$*:*

$$(\lambda x.xx)(Iz) \ \to_\beta \ Iz(Iz) \ \to_\beta \ z(Iz) \ \to_\beta \ zz$$

Op deze manier wordt de subterm $Iz$ twee keer berekend. Als we de redexen in een andere volgorde uitrekenen, krijgen we het volgende:

**Voorbeeld 3 (Een optimale reductie)**

$$(\lambda x.xx)(Iz) \ \to_\beta \ (\lambda x.xx)z \ \to_\beta \ zz$$

We zien hier dat de *normal order reduction strategy* niet het minste aantal stappen vergt. Dit komt doordat subtermen met redexen erin gedupliceerd kunnen worden. Hierdoor moeten we deze subtermen twee maal uitrekenen. Als we echter de volgorde altijd omdraaien, dan kan het zo zijn dat we subtermen evalueren die we vervolgens weggooien, oftewel, dat we overbodig werk doen. In een herschrijfsysteem dat optimaal is in het aantal $\beta$-stappen, wordt geen dubbel of overbodig werk gedaan.

**Definitie 1 (Optimale reducties in de $\lambda$-calculus)** *Een reductiepad is optimaal als het in een zo klein mogelijk aantal reductiestappen tot de normaalvorm van een term komt.*

In [8] wordt voorgesteld dat in een optimaal herschrijfsysteem de eventuele kopieën die tijdens de reductie ontstaan, tegelijk moeten worden uitgerekend. $\beta$-stappen zouden dus moeten worden vervangen door familiestappen:

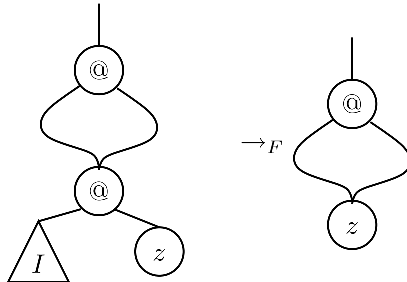
**Definitie 2 (Familiestap)** *Als er één of meerdere subtermen een term voorkomen die kopieën (zie [8] voor de formele definitie van kopie) van elkaar zijn en deze subtermen bevatten een redex, dan is een familiestap een stap waarin al deze redexen tegelijk worden geëvalueerd. We zullen een familiestap aangeven met* $\to_F$*.*

Voorbeeld 2 zou er dan zo uitzien:

**Voorbeeld 4 (Optimale reductie)**

$$(\lambda x.xx)(Iz) \to_F Iz(Iz) \to_F zz$$

Een graafherschrijfsysteem waar gebruik wordt gemaakt van *sharing* (zie [12]), lijkt een goede vorm voor de implementatie van deze stappen. In zo'n systeem kunnen we er voor zorgen dat redexen helemaal niet worden gekopieerd. De laatste stap uit Voorbeeld 2 zou er in zo'n systeem zo uit kunnen zien:



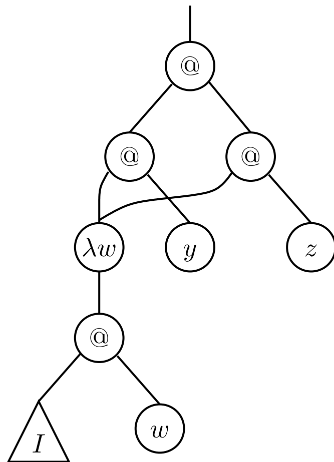

Als we in een herschrijfsysteem gebruik maken van familiestappen en van *normal order reduction strategy* dan weten we dat we geen dubbel werk doen en geen overbodig werk. Zo'n systeem is dan optimaal in het aantal reductiestappen (zie onder andere [2] voor een bewijs hiervan).

Helaas blijkt de simpele manier van *sharing* zoals boven beschreven, niet altijd voldoende om familiestappen te garanderen. Dit kunnen we zien aan het volgende voorbeeld (uit [2]):

**Voorbeeld 5 (Niet optimale sharing)** *Bekijk de volgende term:*

$$(\lambda x\ xy(xz))\lambda w\ Iw$$

*Deze zou er na de eerste $\beta$-stap in graafrepresentatie met* sharing *zo uit zien:*



De $\lambda$-knoop wordt nu geshared in twee redexen. De redexen kunnen zo echter niet worden gereduceerd. Als we er één zouden uitvoeren, zou de ander niet meer kunnen worden uitgevoerd. Om een redex uit te voeren moeten we de $\lambda$-knoop en de hele subgraaf eronder kopieëren. Daardoor wordt echter ook de redex in de subgraaf gekopieerd en zo ontstaat weer dubbel werk.

In [2] wordt een graafherschrijfsysteem beschreven waaraan een *sharing*-operator, de *fan*, is toegevoegd. Deze operator zorgt ervoor dat een subgraaf dynamisch *geshared* kan worden. Dat wil zeggen dat de subgraaf gedeeltelijk gekopieerd kan worden. Zo kan het systeem het gedeelte van de subgraaf dat nodig is voor de redex, namelijk de $\lambda$-knoop, kopiëren, terwijl de rest van de subterm *geshared* blijft. In dit systeem worden familiestappen gezet in plaats van $\beta$-stappen, het maakt gebruik van een *normal order reduction strategy* en daardoor is het optimaal in het aantal reductiestappen.

## 1.3 Efficiëntie

Ik maak in deze scriptie onderscheid tussen optimaliteit en efficiëntie. Met efficiëntie bedoel ik het volgende:

**Definitie 3 (Efficiëntie)** *Een herschrijfsysteem is efficiënter dan een ander herschrijfsysteem, als de kosten die het maakt om termen op de voor hem goedkoopste manier tot hun normaalvormen te reduceren, minder zijn dan de kosten die het andere systeem maakt.*

De kosten die een systeem maakt, worden bepaald door het aantal stappen dat het nodig heeft en de grote van deze stappen (oftewel, de kosten per stap).

Asperti en Guerrini hebben een optimaal herschrijfsysteem ontwikkeld. Ze werken met familiestappen en ze gebruiken een *normal order reduction strategy* . Ze hebben dus het minst mogelijk aantal reductiestappen nodig om een term tot zijn normaalvorm te reduceren. Om het systeem correct te laten werken hebben ze echter nog twee operatoren geïntroduceerd, die ze de boekhoud-operatoren noemen. Deze zorgen ervoor dat er tussen het zetten van een reductiestap, een variabel aantal extra herschrijfstappen moeten worden gezet. Deze herschrijfstappen hebben niets met de reductiestappen te maken, maar met de dynamische sharing van het systeem. Dit betekent echter wel dat de kosten per reductiestap variabel zijn (er zijn namelijk een variabel aantal herschrijfstappen nodig voor we een reductiestap kunnen zetten). Om een term naar zijn normaalvorm te reduceren, heeft het systeem dus het minste aantal reductiestappen nodig, maar de kosten per stap kunnen soms hoog zijn.

Er zijn herschrijfsystemen die niet optimaal zijn in het aantal reductiestappen, maar in de praktijk vaak efficiënter presteren dan het systeem van Asperti en Guerrini, bijvoorbeeld systemen die gebruik maken van supercombinatoren. Bij een systeem gebaseerd op supercombinatoren worden de reductiestappen onderverdeeld in meerdere goedkopere stappen. De kosten per reductiestap zijn daardoor vrij laag. Er kunnen echter wel nog steeds redexen worden gekopieerd en daardoor kan het zijn dat er dubbel werk wordt gedaan. Reducties waarin

veel subtermen met redexen worden gekopieerd, zijn daardoor een groter probleem voor deze systemen dan voor het systeem van Asperti en Guerrini, omdat ze niet optimaal zijn in het aantal reductiestappen.

Het systeem zoals in [9] en [10] beschreven staat, is optimaal en gebruikt maar één boekhoudoperator, namelijk de $\reflectbox{$\lambda$}$-operator. Dit scheelt in het aantal soorten boekhoudoperatoren ten opzichte van het systeem van Asperti en Guerrini. Bovendien heeft onze boekhoudoperator nog een duidelijke functie, namelijk het bijhouden van het einde van de scope. De kosten per reductiestap zijn in dit systeem echter ook variabel.

## 1.4 De $\reflectbox{$\lambda$}$-operator

In Paragraaf 1.3 heb ik er al op gewezen dat de $\reflectbox{$\lambda$}$-operator de functie van boekhoudoperator kan vervullen in een optimaal herschrijfsysteem. Maar het toevoegen van expliciete scope-informatie kan nog meer voordelen hebben. Je kunt nu in de objecttaal aangeven dat een bepaalde variabele ergens niet in voorkomt. Een heel aantal regels uit de logica en $\lambda$-calculus zijn nu in de objecttaal te definiëren, waar dat zonder de expliciete scope informatie niet het geval is. Enkele voorbeelden:

- De $\eta$-regel uit de $\lambda$-calculus luidt: $\lambda x\ Mx \to M$ als $x \notin VV(M)$. In de $\reflectbox{$\lambda$}$-calculus kan deze regel als volgt worden uitgedrukt: $\lambda x(\reflectbox{$\lambda$}x\ M)x \to M$.

- Als we termen naar combinatoren vertalen (zie [3, Definitie 7.1.5]) komen we de volgende regel tegen:

  $\lambda^* xM = KM$ als $x$ niet vrij voorkomt in $M$

  In de $\reflectbox{$\lambda$}$-operator kan deze regel als volgt worden weergegeven:

  $\lambda^* x \reflectbox{$\lambda$} xM = KM$

- Er zijn regels in de logica die alleen mogen worden toegepast als bepaalde variabelen niet vrij voorkomen in een term. Bijvoorbeeld het axioma om $\forall$ de distribueren over $\Rightarrow$ in een Hilbert-stijl systeem: $(\forall x(F \Rightarrow G)) \Rightarrow (F \Rightarrow \forall xG)$ als $x \notin VV(F)$. Een end-of-scope operator zou ook hier meer duidelijkheid kunnen verschaffen over welke variabelen vrij in een subterm voorkomen. Dan kunnen ook hier vrije-variabeleconstraints in de objecttaal worden uitgedrukt.

Twee andere eigenschappen van de $\reflectbox{$\lambda$}$-calculus die voordelig kunnen zijn, zijn de volgende:

- De $\lambda$-calculus uitgebreid met de $\reflectbox{$\lambda$}$-operator is confluent zonder gebruik te maken van alpha-gelijkheid en dus wordt confluentie modulo alpha van gewone lambda calculus geïmpliceerd (zie [5] voor een bewijs hiervan).

- Als we in de ʎ-calculus termen reduceren hoeven we niet meer bang te zijn voor verwarring tussen vrije en gebonden variabelen. We hoeven tijdens het substitueren dus niet meer te herbenoemen naar een $\alpha$-equivalente term. Als we willen kunnen we naar een naamloze variant van de $\lambda$-calculus (zie Paragraaf 2.2.5 voor een vergelijking van de ʎ-calculus en de $\lambda$-calculus met de Bruijn indices).

## 1.5 Lokaliteit

**Definitie 4 (Lokaliteit)**

- *Een herschrijfregel is* lokaal *als het gedeelte van de term dat we moeten onderzoeken om te weten of de regel kan worden toegepast, begrensd is. Gegeven de regel moet deze grens van tevoren bekend zijn.*
- *Een herschrijfsysteem is* lokaal *als alle herschrijfregels van dat systeem lokaal zijn.*

Een voorbeeld van een niet lokale regel is bijvoorbeeld de $\eta$-regel uit de $\lambda$-calculus ($\lambda x\ Mx \rightarrow_\eta M$ als $x$ niet vrij in $M$ voor komt). We moeten namelijk heel $M$ afzoeken om te kijken of $x$ er niet vrij in voorkomt, voordat we de regel kunnen toepassen en er zit geen vooraf bepaalde bovengrens aan de grootte van $M$.

De versie van de ʎ-calculus die in [5] beschreven staat is niet lokaal. Dit komt doordat er tijdens het herschrijven een willekeurig aantal ʎ-operatoren tussen een applicatie en $\lambda$-operator kunnen ontstaan. Hierdoor is de $\beta$-regel niet meer lokaal. We moeten dus een willekeurig groot gedeelte van de graaf onderzoeken voor we weten of we de regel kunnen toepassen en dit is soms veel werk. In een lokale versie van deze calculus wordt de $\beta$-regel onderverdeeld in een aantal lokale herschrijfregels. Per definitie van lokaliteit zit er een bovengrens aan het gedeelte van de graaf dat we moeten onderzoeken voordat we weten of we een van deze regels mogen toepassen. De ʎ-operatoren die zich tussen een applicatie en een $\lambda$-operator bevinden, worden nu om de beurt weggehaald. Een nadeel dat deze lokaliteit met zich meebrengt is dat we het overzicht verliezen. Het kan gebeuren dat we een heleboel lokale stapjes zetten, zonder dat we daarna een $\beta$-stap kunnen zetten (zie ook Hoofdstuk 7).

## 1.6 Notatie

In deze scriptie gebruik ik de volgende conventies met betrekking tot notatie:

**Algemeen**

- Met ..., gebruikt zoals in 1...5, bedoel ik de buitenste elementen, aangevuld met daartussen de meest voor de hand liggende elementen, in dit geval dus 234. Deze notatie is ook gebruikelijk in bijvoorbeeld Haskell.

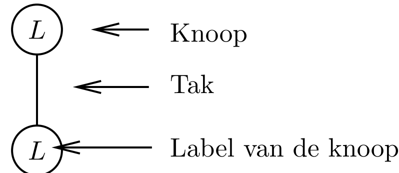
- Voor natuurlijke getallen gebruik ik de volgend variabelenamen: $n$, $m$, $i$ en $j$.

**Termen**

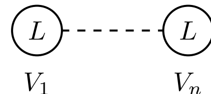
- $p...z$ of $p_1...z_n$ zijn termvariabelen uit de $\lambda$-calculus.
- $M$, $N$, $M_1...M_n$, etc. staan voor willekeurige termen uit de $\lambda$- of $\curlywedge$-calculus.
- De substitutie wordt aangegeven door $M[x := N]$, wat betekent: vervang alle vrije variabelen met de naam $x$ in $M$ door $N$.
- Als term $M$ zich in een omgeving $C$ bevindt, dan noteren we dat als volgt: $C[M]$
  Met een omgeving bedoelen we hier een superterm, oftewel een term met een gat erin, waarbij $M$ in het gat terechtkomt.

**Grafen**

- Grafen zien er als volgt uit:

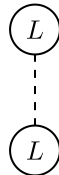
  De grafen in deze scriptie zijn over het algemeen ongericht.
- Met

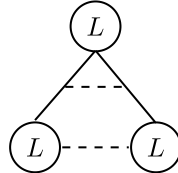
  bedoel ik een rij knopen met de namen $V_1$ tot en met $V_n$.
- Een willekeurige graaf of graafvariabele, met de naam $A$, wordt als volgt genoteerd:

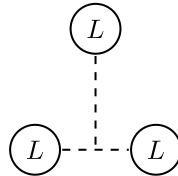
  $A$, $B$ en $C$ gebruik ik voor graafvariabelen.
- Soms weten we niet zeker of er zich een tak tussen twee knopen bevindt. In dat geval gebruiken we de volgende notatie:

- Als we een variabel aantal takken naar even zoveel knopen willen noteren, dan doen we dat zo:




- Als we één knoop hebben en een willekeurig aantal andere knopen, met eventueel een tak tussen de ene knoop en elk van de andere knopen, dan noteren we dat zo:




**Stacks**

In het algemeen geld dat ik stacks links/rechts noteer. Daarmee bedoel ik boven/beneden. Het meeste linkse element is dus eigenlijk het bovenste element. Verder geldt:

- Een stack noteer ik op de volgende manier: $[x, y, z, p]$.
- Als $C$ een stack is, dan is $xC$ de stack $C$ met het element $x$ erboven op gezet.
- Met $[..., n : x, ...]$ bedoel ik een willekeurige stack met als $n$-de element $x$. Hierbij beginnen we met tellen vanaf 1.
- $\square$ is de notatie voor een lege stack.
- Ik gebruik ook de vector notatie $\overrightarrow{x}$ om stacks of gedeeltes van stacks af te korten.

**Herschrijfregel** Met $\rightarrow_x^n$, bedoelen we $n$ maal de herschrijfregel van de soort $x$.

# 2 De $\lambda$-calculus en de $\reflectbox{$\lambda$}$-calculus

In [5] wordt de $\lambda$-calculus uitgebreid met een nieuwe operator, de end-of-scope operator of $\reflectbox{$\lambda$}$-operator. De calculus die zo ontstaat noemen we de $\reflectbox{$\lambda$}$-calculus. In Paragraaf 2.2 geef ik een beschrijving van de $\reflectbox{$\lambda$}$-calculus. Omdat deze calculus een uitbreiding is van de $\lambda$-calculus zal ik in Paragraaf 2.1 beginnen met een korte, informele beschrijving van de $\lambda$-calculus. Voor een uitgebreide beschrijving van de $\lambda$-calculus, zie [3].

## 2.1 De $\lambda$-calculus

De $\lambda$-calculus is een rekensysteem. Dat betekent dat de $\lambda$-calculus bestaat uit een verzameling objecten, de termen, en uit herschrijfregels die een term naar een andere term kunnen herschrijven. Ik zal kort de volgende onderwerpen bespreken: syntax (de grammatica waarmee termen kunnen worden opgebouwd), semantiek (betekenis van de termen), herschrijfregels ($\alpha$-conversie en $\beta$-reductie) en de afsluiting van termen.

### 2.1.1 Syntax

**Definitie 5 ($\lambda$-termen)**
*De termen worden gemaakt met het volgende alfabet:*

**Variabelen** $x_0, x_1, \ldots.$

**Abstractor** $\lambda$

**Haakjes** ( , )

*De termen worden met de volgende grammatica opgebouwd:*

**Variabelen** *Als $x$ een variabele is, dan is $x$ een term.*

**Abstractie** *Als $M$ een term is en $x$ een variabele, dan is $(\lambda x\ M)$ een term.*

**Applicatie** *Als $M$ en $N$ termen zijn, dan is $(MN)$ een term.*

We maken voor de notatie van termen gebruik van de volgende conventies:

- De buitenste haakjes worden weggelaten.
- Het bereik van de $\lambda$-operator loopt zover mogelijk door naar rechts, dus:

$$M\lambda x\ NP = M(\lambda x\ NP)$$

- Applicatie associeert naar links, dus:

$$M_1M_2M_3 = ((M_1M_2)M_3)$$

- Meerdere $\lambda$-operatoren achter elkaar korten we af, dus:

$$\lambda xyz.M = \lambda x\lambda y\lambda z\ M$$

Als we deze grammatica gebruiken om termen te genereren, kunnen we bijvoorbeeld het volgende verkrijgen, respectievelijk zonder en met het toepassen van de conventies:

$$x = x$$

$$(\lambda x\ x) = \lambda x\ x$$

$$(\lambda x\ (yz)) = \lambda x\ yz$$

$$((\lambda x(\lambda y(x(yp))))y) = (\lambda xy.x(yp))y$$

### 2.1.2 Semantiek

In de $\lambda$-calculus staan termen voor functies. De formele (ongetypeerde) $\lambda$-calculus bestudeert functies en hun *applicatieve* gedrag.

**Applicatie** Applicatie is daarom een primitieve operatie van de $\lambda$-calculus. De functie $f$ toegepast op argument $a$ wordt genoteerd als $fa$. $MN$ is dus term $M$ toegepast op de term $N$. Als een term uit één of meerdere termen is opgebouwd, dan noemen we de termen waaruit hij is opgebouwd subtermen.

**Abstractie** Naast de applicatie-operatie hebben we abstractie in de $\lambda$- calculus. Laat $t(x)$ een term zijn waar $x$ mogelijk vrij in voorkomt. Dan is $\lambda x\ t(x)$ de functie die aan een argument $a$ de waarde $t(a)$ toekent:

$$(\lambda x\ t(x))a = t(a)$$

**Definitie 6 (Bereik)** *Als een term de vorm $\lambda x\ M$ heeft, dan ligt $M$ precies in het bereik van de $\lambda x$-operator. Alle variabelen met de naam $x$ die* vrij *voorkomen in $M$, worden gebonden door de $\lambda x$-operator. De term $M$ noemen we de body van de $\lambda x$-operator.*

**Variabelen** We zien dat het basiselement van termen de variabele is. We moeten onderscheid maken tussen gebonden en vrije variabelen. Een variabele $x$ is vrij als hij zich niet binnen het *bereik* van een $\lambda$ $x$-operator bevindt. Anders is de variabele gebonden. Bijvoorbeeld in $x(\lambda y\ xy)$ komt $x$ twee maal vrij voor en is $y$ gebonden. Een vrije variabele staat voor een willekeurig object, een gebonden variabele is een syntactische categorie.

**Definitie 7 (Vrije variabelen)** *De verzameling $VV(M)$ bestaat uit de vrije variabelen van $M$ en kan als volgt inductief worden gedefiniëerd:*

$$VV(x) = \{x\},$$

$$VV(\lambda x\ M) = VV(M) - \{x\},$$

$$VV(MN) = VV(M) \cup VV(N)$$

- $M$ is gesloten als $VV(M) = \emptyset$.
- Als $M$ gesloten is, noemen we $M$ een combinator.
- Als $VV(M) = X$ dan is $\lambda\overrightarrow{x}.M$ de afsluiting van $M$, waarbij $\overrightarrow{x}$ een geordende versie is van $X$.

### 2.1.3 $\alpha$-conversie

In de $\lambda$-calculus zijn er meerdere termen die hetzelfde gedrag vertonen. Deze termen zijn representaties van dezelfde functie. Bijvoorbeeld: $\lambda x\ x$ en $\lambda y\ y$ zijn allebei instanties van de identiteitsfunctie, de functie die aan een argument het argument als waarde toekent. (Vergelijking met wiskunde: F(x) = x en F(y) = y). Het enige verschil tussen deze termen is de naam van de gebonden variabelen en hun binders. De structuur van beide termen is hetzelfde (de structuur van een term is te zien aan hoe hij grammaticaal is opgebouwd). We noemen termen die instanties zijn van dezelfde functie $\alpha$-equivalent. Twee $\alpha$-equivalente termen zijn naar elkaar toe te schrijven met $\alpha$-conversie.

Om $\alpha$-conversie te kunnen formuleren hebben we de substitutie operator nodig. $M[x := N]$ staat voor het resultaat van dat je krijgt als in $M$ alle vrije voorkomens van $x$ worden vervangen (gesubstitueerd) door $N$.

**Definitie 8 ($\alpha$-conversie)**

$$\lambda x\ M \ \rightarrow_\alpha\ \lambda y\ M[x := y]$$

*Met $x$ en $y$ variabelen, $M$ een term en $y$ een variabele die vers is, dus in het geheel niet in $M$ voor komt.*

De eis $y$ niet in $M$ voor komt, is er niet voor niets. Als we substitutie gebruiken moeten we altijd zorgen dat er geen verwarring ontstaat tussen gebonden en vrije variabelen. Hierover meer in Paragraaf 2.1.5.

**Voorbeeld 6 ($\alpha$-equivalentie)** *$\lambda xy.yx$ en $\lambda yx.xy$ zijn $\alpha$-equivalent. Ze zijn naar elkaar toe te schrijven met behulp van $\alpha$-conversie:*

$$\lambda xy.yx \ \rightarrow_\alpha\ \lambda zy.yx[x := z] = \lambda zy.yz \ \rightarrow_\alpha\ \lambda zx.xz \ \rightarrow_\alpha\ \lambda yx.xy$$

### 2.1.4 $\beta$-reductie

**Definitie 9 ($\beta$-redex)** *Een $\beta$-redex (of kortweg, redex) is een term of subterm van de volgende vorm:*

$$(\lambda x\ M)N$$

*Met $x$ een willekeurige variabele en $M$ en $N$ willekeurige termen.*

We zien dat een redex bestaat uit een functie die wordt toegepast op een argument. Een redex kunnen we herschrijven door middel van $\beta$-reductie. $\beta$-reductie ziet er als volgt uit:

**Definitie 10 ($\beta$-reductie)**

$$(\lambda x\ M)N \ \rightarrow_\beta\ M[x := N]$$

*Met $x$ een willekeurige variabele en $M$ en $N$ willekeurige termen.*

Enkele voorbeelden van het herschrijven (reduceren) van termen:

**Voorbeeld 7 ($\beta$-reductie)**

$$(\lambda y\ xyz)a\ \rightarrow_\beta\ xyz[y := a] = xaz$$

$$(\lambda x\ xxz)y\ \rightarrow_\beta\ yyz$$

$$(\lambda xz.xz)yv\ \rightarrow_\beta\ (\lambda z\ yz)v\ \rightarrow_\beta\ yv$$

$$(\lambda x\ xy)\lambda z\ zvz\ \rightarrow_\beta\ (\lambda z\ zvz)y\ \rightarrow_\beta\ yvy$$

$$(\lambda x\ xx)\lambda x\ xx\ \rightarrow_\beta\ (\lambda x\ xx)\lambda x\ xx\ \rightarrow_\beta\ (\lambda x\ xx)\lambda x\ xx\ \rightarrow_\beta\ ...$$

We zien in de voorbeelden dat sommige termen na één of meerdere stappen niet meer verder kunnen worden gereduceerd. Het laatste voorbeeld laat echter zien dat dit niet altijd het geval is. Als we het laatste voorbeeld reduceren krijgen we dezelfde term terug. Deze term kunnen we dus tot in het oneindige blijven reduceren.

**Definitie 11 ($\beta$-normaalvorm)** *Een $\lambda$-term waar geen redexen meer in voor komen, is in $\beta$-normaalvorm.*

Zoals we in Voorbeeld 7 kunnen zien, hebben sommige termen geen normaalvorm. Dit is vergelijkbaar met een programma dat niet termineert, iets dat we bij programmeren willen voorkomen. Er is een variant van de $\lambda$-calculus, de getypeerde $\lambda$-calculus, waar een domein-bereik idee wordt geïncorporeerd voor termen. Een term krijgt een type $(A \rightarrow B)$ dat aangeeft dat het een functie is van het type dat alleen kan worden toegepast op termen met het juiste type $(A)$. In deze variant van de $\lambda$-calculus heeft iedere term een $\beta$-normaalvorm. Bij deze variant zullen we verder echter niet stil staan. Wel zullen we de voorbeelden meestal zo kiezen dat ze een normaalvorm hebben.

De $\lambda$-calculus is confluent (zie voor een bewijs hiervan bijvoorbeeld [3, Paragraaf 11.1]). Dat wil zeggen dat als we vanuit een term twee resultaten kunnen bereiken, dan zijn deze resultaten via $\beta$-reductie naar elkaar toe te schrijven. Dit houdt gelijk in dat als een term een $\beta$-normaalvorm heeft, deze uniek is voor die term. Als een term namelijk twee $\beta$-normaalvormen zou hebben, zouden deze naar elkaar toe moeten kunnen worden geschreven, maar omdat ze allebei niet verder kunnen worden gereduceerd (het zijn immers normaalvormen) is dit niet mogelijk. Een term kan er dus maar één hebben.

### 2.1.5 Substitutie

We zullen nu laten zien dat enige voorzichtigheid is geboden bij het gebruik van substitutie om verwarring tussen gebonden en vrije variabelen te voorkomen:

**Voorbeeld 8 (Verwarring tussen gebonden en vrije variabelen)**
*In dit voorbeeld geven we een incorrecte afleiding om te illustreren wat er fout kan gaan bij het toepassen van een naïeve notie van substitutie:*
*Neem* $F = \lambda xy.yx$*. Dan geldt voor alle* $M$*,* $N$*:*

$$FMN = ((\lambda xy.yx)M)N \ \rightarrow_\beta \ (\lambda y \ yM)N \ \rightarrow_\beta \ NM$$

*In het bijzonder:* $Fyx =_\beta xy$*. Maar:*

$$Fyx = ((\lambda xy.yx)y)x \ \rightarrow_\beta \ (\lambda y \ yy)x \ \rightarrow_\beta \ xx$$

*Dus:* $yx =_\beta xx$*.*

Met dit resultaat kunnen we alles afleiden, dus alle termen worden $\beta$-equivalent. Dit maakt ons systeem inconsistent. Het gaat mis in deze stap:

$$(\lambda xy.yx)y \ \rightarrow_\beta \ \lambda y \ yy$$

De vrije variabele $y$, wordt nadat hij voor $x$ is gesubstitueerd, gebonden door de $\lambda y$-operator. We hebben gezien dat vrije variabelen en gebonden variabelen een andere betekenis hebben. We moeten dus voorkomen dat een vrije variabele na substitutie gebonden wordt. We kunnen zorgen dat dit niet gebeurt door voor de substitutie de term met behulp van $\alpha$-conversie te herbenoemen, zodat alle gebonden variabelen anders heten dan de variabelen die voorkomen in de term die wordt gesubstitueerd. In ons voorbeeld krijgen we dan:

$$(\lambda xy.yx)y \ \rightarrow_\alpha \ (\lambda xz.zx)y \ \rightarrow_\beta \ \lambda z \ zy$$

Nu is de vrije variabele na substitutie nog steeds vrij. Dit is ook de reden dat we bij $\alpha$-conversie eisen dat de variabelenaam die we kiezen vers is (niet al in de term voorkomt). Anders zouden we het volgende kunnen krijgen:

$$\lambda x \ xy \ \rightarrow_\alpha \ \lambda y \ yy$$

We zullen zien dat de verwarring tussen gebonden en vrije variabelen ook te voorkomen is door het introduceren van de $\curlywedge$-operator (zie Paragraaf 2.2).

Merk overigens op dat deze problemen ontstaan door een linguïstisch aspect van de termen. Een aspect waar we eigenlijk niet in geïnteresseerd zijn. We zullen in deze scriptie over het algemeen de termen beschouwen modulo $\alpha$-conversie. Een term beschouwen we als een instantie van de hele $\alpha$-equivalentieklasse: als we het hebben over een $\lambda$-term, dan hebben we het over de klasse van $\alpha$-equivalente termen. De grafen die we in Hoofdstuk 3 introduceren, representeren een hele $\alpha$-equivalentieklasse van termen.

### 2.1.6 Gesloten termen

Een term zonder vrije variabelen noemen we een gesloten term. We zullen bij het vertalen van een term naar zijn graafrepresentatie er altijd van uitgaan dat de term die we vertalen een gesloten term is. Van een niet gesloten term kunnen we altijd de afsluiting nemen. We kunnen de afsluiting van de term herschrijven en daarna weer terugvertalen naar de onafgesloten term. Dit gaat als volgt:

**Voorbeeld 9 (Afsluiting van een term)**
*Voor* $M_1 \rightarrow_\beta M_2$ *en* $VV(M_1) = \overrightarrow{x}$*:*

$$M_1 \rightarrow_{afsl} \lambda\overrightarrow{x}.M_1 \rightarrow_\beta \lambda\overrightarrow{x}.M_2 \rightarrow_{open} M_2$$

*Met* $\rightarrow_{afsl}$ *de operatie die van een term zijn afsluiting geeft en* $\rightarrow_{open}$ *de operatie die dit weer ongedaan maakt.*

Het bereik van de $\lambda$-operatoren $\lambda\overrightarrow{x}$ ligt helemaal over $M_1$ heen. Daardoor zullen ze geen onderdeel vormen van een redex en dus niet worden herschreven. Na de herschrijfregels kunnen we ze dus weer weghalen. We zullen dit aannemelijk maken aan de hand van het volgende voorbeeld met de term $((\lambda zv.vz)x)y$:

**Voorbeeld 10 (Afsluiting van een term 2)**
*Met* $VV(((\lambda zv.vz)x)y) = \{xy\}$*:*

$$((\lambda zv.vz)x)y \rightarrow_{afsl} \lambda xy.((\lambda zv.vz)x)y \rightarrow_\beta \lambda xy.(\lambda v\ vx)y \rightarrow_\beta$$

$$\lambda xy.yx \rightarrow_{open} yx$$

## 2.2 De ʎ-calculus

In [5] wordt de $\lambda$-calculus uitgebreid met een nieuwe operator tot de ʎ-calculus. Deze operator is de end-of-scope operator of de ʎ-operator (*admal*-operator). Het idee is dat ʎ$x$ het bereik afsluit van de bijbehorende $\lambda x$-operator boven hem in de term (gezien als boom). In deze paragraaf zal ik de syntax van de ʎ-calculus geven en bespreken hoe noties van $\alpha$-equivalentie, $\beta$-reductie en substitutie kunnen worden uitgebreid van de $\lambda$-calculus naar de ʎ-calculus. Daarna zal ik ingaan op overeenkomsten tussen de ʎ-calculus en de $\lambda$-calculus in de zogenaamde De Bruijn notatie ([4]).

### 2.2.1 Syntax

**Definitie 12 (De ʎ-termen)**
*De ʎ-termen worden gemaakt met het volgende alfabet:*

**Variabelen** $x_0, x_1, ....$

**Abstractor** $\lambda$

**End of Scope** ʎ

**Haakjes** ( , )

*De ʎ-termen worden met de volgende grammatica opgebouwd:*

**Variabelen** *Als* $x$ *een variabele is, dan is* $x$ *een ʎ-term.*

**Abstractie** *Als* $M$ *een ʎ-term is en* $x$ *een variabele, dan is* $(\lambda x\ M)$ *een ʎ-term.*

**Applicatie** *Als* $M$ *en* $N$ *ʎ-termen zijn, dan is* $(MN)$ *een ʎ-term.*

**End of scope** *Als $M$ een ⅄-term is en $x$ een variabele, dan is ⅄$x$ $M$ een ⅄-term.*

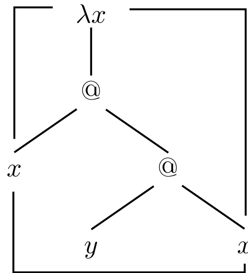
*Waarbij $x$ een willekeurige variabele en $M$ en $N$ willekeurige ⅄-termen.*

We zullen in de notatie van ⅄-termen dezelfde conventies hanteren als bij $\lambda$-termen, waarbij de ⅄-operator wordt behandeld als de $\lambda$-operator.

### 2.2.2 De betekenis van de ⅄-operator - een voorbeeld

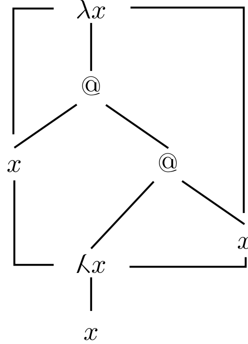
De termen uit de ⅄-calculus staan, net als in de $\lambda$-calculus, voor functies. Applicatie en abstractie hebben in de ⅄-calculus dezelfde betekenis. Alleen de ⅄-operator is nieuw en daarom ga ik nu wat dieper in op de werking van deze operator.

Abstractie in de $\lambda$-calculus kunnen we zien als samengesteld uit twee delen: een deel is de duale operatie van applicatie, het andere gedeelte zorgt voor het openen van het bereik van de $\lambda$-operator, waarin variabelen gebonden kunnen worden. Het idee is om een operator te introduceren, de ⅄-operator, die dit bereik weer afsluit. ⅄$x$ $M$ drukt uit dat het bereik van de laatste $\lambda$-operator, die $x$ bindt, is afgesloten in $M$. De ⅄-operator lijkt veel op het 'haakje sluiten' in de $\lambda$-calculus, maar moet daar toch niet mee worden verward. Het haakje sluit het bereik af naar rechts in een lineaire representatie, terwijl ⅄ het bereik naar beneden in de boom afsluit. Bijvoorbeeld in $M = \lambda x\ x(($⅄$x\ x)x)$ is de $x$ in ⅄$x$ $x$ niet gebonden door de $\lambda x$-operator en dus een vrije variabele. Deze term is equivalent met bijvoorbeeld de $\lambda$-term $\lambda x\ x(yx)$. De vrije variabele kan weer worden gebonden door een $\lambda$-operator, als in $\lambda x\ M\ = \lambda xx.x(($⅄$x\ x)x)$. Deze term is dan equivalent met $\lambda yx.x(yx)$. (Zie [5] voor een formele definitie van deze equivalentierelatie). We zien in de $\lambda$-calculus dat het bereik van de $\lambda$-operator over de hele body van de $\lambda$-operator heen loopt. In ons voorbeeld ook over de vrije variabele $y$. De term $\lambda x\ x(yx)$ ziet er in boomrepresentatie zo uit:

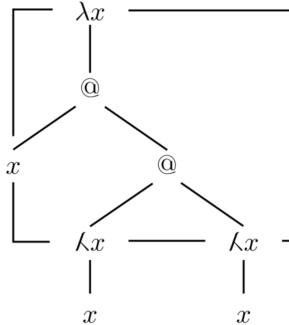



Met de ⅄-operator sluiten we het bereik in de diepte af. We kunnen dus een subterm uit de body buiten het bereik van de $\lambda$-operator houden. De term

$\lambda x\ x((\reflectbox{\lambda} x\ x)x)$ ziet er in boomrepresentatie als volgt uit:




Merk op dat er door de vertakkende structuur van termen, meerdere $\reflectbox{\lambda}$-operatoren kunnen zijn die een $\lambda$-operator kunnen afsluiten. Bijvoorbeeld in de term $\lambda x\ x((\reflectbox{\lambda} x\ x)(\reflectbox{\lambda} x\ x))$. In boomrepresentatie ziet dat er zo uit:




### 2.2.3 Correspondentie tussen $\lambda$- en $\reflectbox{\lambda}$-termen

De syntax van $\reflectbox{\lambda}$-termen is een uitbreiding van de syntax van $\lambda$-termen. Dat betekent dat de verzameling van $\lambda$-termen een deelverzameling is van de verzameling van $\reflectbox{\lambda}$-termen. Omdat we de eigenschappen van $\reflectbox{\lambda}$-termen willen vergelijken met de eigenschappen van $\lambda$-termen moeten we weten hoe we een $\reflectbox{\lambda}$-term kunnen vertalen naar een equivalente $\lambda$-term. Om dit te bereiken moeten we zorgen dat alle $\reflectbox{\lambda}$-operatoren uit de term verdwijnen. We kunnen niet zomaar alle $\reflectbox{\lambda}$-operatoren weghalen want dan kan er verwarring ontstaan tussen gebonden en vrije variabelen zoals in het volgende voorbeeld:

**Voorbeeld 11 (Op de verkeerde manier van $\reflectbox{\lambda}$- naar $\lambda$-term)**

$$\lambda x\ x(\reflectbox{\lambda} x\ x)x \Rightarrow \lambda x\ xxx$$

In dit voorbeeld is de variabele $x$ in $\reflectbox{\lambda} x\ x$ vrij. Als we de $\reflectbox{\lambda}$-operator zouden weghalen, wordt het een gebonden variabele omdat hij in het bereik van de $\lambda$-operator terechtkomt. We kunnen dit probleem oplossen door voordat we een

$\reflectbox{$\lambda$}$-operator weghalen de term te vertalen naar een $\alpha$-equivalente term door de $\lambda$-operator waarvan de scope door de $\reflectbox{$\lambda$}$-operator wordt afgesloten, alle variabelen die door de $\lambda$-operator gebonden worden en de $\reflectbox{$\lambda$}$-operator, te herbenoemen met een verse variabelenaam. Ons voorbeeld ziet er dan als volgt uit:

$$\lambda x\ x(\reflectbox{$\lambda$}x\ x)x\ \rightarrow_\alpha\ \lambda y\ y(\reflectbox{$\lambda$}y\ x)y \Rightarrow \lambda y\ yxy$$

We zien dat nu de vrije variabele vrij blijft en dat dit de vertaling is die we ook al in Paragraaf 2.2.2 gaven. We kunnen via deze procedure een $\reflectbox{$\lambda$}$-term altijd vertalen naar een normale term. De term die we dan krijgen noemen we de corresponderende $\lambda$-term.

### 2.2.4 Termen in balans

Met de $\reflectbox{$\lambda$}$-operator kunnen we de diepte van de scope van een $\lambda$-operator beperken. We kunnen dus een subterm in de body van een $\lambda$-operator scope-vrij maken. Nu we het einde van de scopes van $\lambda$-operatoren expliciet bijhouden, moeten we nadenken over de volgorde waarin ze worden afgesloten. Omdat $\reflectbox{$\lambda$}$-operatoren veel op haakjes zouden we ervoor kunnen kiezen dat een $\reflectbox{$\lambda$}$-operator altijd de $\lambda$-operator afsluit die het laagst boven hem in de boomrepresentatie voorkomt. Maar omdat $\reflectbox{$\lambda$}$-operatoren namen met zich meedragen (en in die zin verschillen van haakjes), is dit niet altijd het geval. In het volgende voorbeeld liggen de scopes over elkaar heen:

$$P = \rlap{$\underbrace{\lambda x\lambda y\reflectbox{$\lambda$}x}$}\lambda x\overbrace{\lambda y\reflectbox{$\lambda$}x\reflectbox{$\lambda$}y}Q$$

Dit geeft echter duidelijk semantische problemen (probeer substitutie te definiëren). Daarom hanteren we een *jump*-semantiek. Een voorkomen van $\reflectbox{$\lambda$}x\ M$ sluit impliciet ook alle scopes af van alle $\lambda$-operatoren die tussen het voorkomen en zijn bijbehorende $\lambda$-operator voorkomen. $P$ is dus equivalent met $\lambda xy.\reflectbox{$\lambda$}yxy.Q$. $\reflectbox{$\lambda$}$-termen komen voor in verschillende gradaties van balans. Naast de ongebalanceerde *jump*-termen, definiëren we twee subklassen:

**Definitie 13 (Scope-balans)** *Een term is in* scope-balans *als geldt dat als een scope wordt afgesloten door een $\reflectbox{$\lambda$}$-operator, deze scope behoort tot de $\lambda$-operator die het laagst boven deze $\reflectbox{$\lambda$}$-operator in de boom voorkomt en die nog niet is afgesloten.*

Termen in scope-balans kunnen met de volgende afleidingsregels worden afgeleid:

$$\frac{x \in \overrightarrow{s}}{\overrightarrow{s} \vdash x}\,Var \qquad \frac{x\overrightarrow{s} \vdash M}{\overrightarrow{s} \vdash \lambda x\ M}\,\lambda \qquad \frac{\overrightarrow{s} \vdash M}{x\overrightarrow{s} \vdash \reflectbox{$\lambda$}x\ M}\,\reflectbox{$\lambda$} \qquad \frac{\overrightarrow{s} \vdash M \quad \overrightarrow{s} \vdash N}{\overrightarrow{s} \vdash (MN)}\,@$$

Uit deze definitie volgt dat in een term die in scope-balans is, de scopes *genest* zitten. De termen die we in deze scriptie bekijken zullen altijd in scope-balans zijn.

We kunnen nog verder gaan dan scope-balans. We kunnen ook zorgen dat de variabelen in balans zijn. Hiermee willen we zeggen dat een variabele altijd

gebonden is door de $\lambda$-operator die het laagst boven de variabele in de boom zit en nog niet is afgesloten.

**Definitie 14 (Balans)** *Een term is in* balans *als de* ʎ*-operatoren zo geplaatst zijn dat alle variabelen in de term gebonden worden door de* $\lambda$*-operator die het laagst boven de variabele in de boom voorkomt en nog niet is afgesloten. Daarnaast moet de term ook in scope-balans zijn.*

De afleidingsregels voor ʎ-termen in balans, zijn hetzelfde als die voor ʎ-termen in scope-balans. Alleen de eerste is anders:

$$\frac{}{x\overrightarrow{s} \vdash x}\ \mathit{Var}$$

**Voorbeeld 12 (Termen in balans)**

**Termen in balans** *De volgende termen zijn volledig in balans:*

$$\begin{array}{l} \vdash \lambda x\ x \\ \vdash \lambda xy.\text{ʎ}y\ x \\ h \vdash \lambda xyz.z\text{ʎ}zy.x\lambda pq.\text{ʎ}qpx.h \\ x \vdash \lambda xxxxx.x\text{ʎ}xxx.x\text{ʎ}xx.x \end{array}$$

*Deze termen zijn af te leiden met de afleidingsregels voor termen in balans. Ik zal dat laten zien aan de hand van het tweede voorbeeld:*

$$\dfrac{\dfrac{\dfrac{\dfrac{x \in \{x\}}{x \vdash x}\ \mathit{Var}}{yx \vdash \text{ʎ}y\ x}\ \text{ʎ}}{x \vdash \lambda y\text{ʎ}y\ x}\ \lambda}{\vdash \lambda x\lambda y\text{ʎ}y\ x}\ \lambda$$

**Termen in scope-balans:** *De volgende termen zijn niet in balans, maar wel in scope balans:*

$$\begin{array}{l} \lambda xyz.y \\ \lambda xyz.\text{ʎ}z\ x \\ \lambda xyz.y\text{ʎ}zy.x\lambda pq.ph \end{array}$$

*Deze termen zijn af te leiden met de afleidingsregels voor termen in scope-balans. Dit laat ik zien aan de hand van het tweede voorbeeld:*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{yx \vdash x}\ \mathit{Var}}{zyx \vdash \text{ʎ}z\ x}\ \text{ʎ}}{yx \vdash \lambda z\text{ʎ}z\ x}\ \lambda}{x \vdash \lambda y\lambda z\text{ʎ}z\ x}\ \lambda}{\vdash \lambda x\lambda y\lambda z\text{ʎ}z\ x}\ \lambda$$

*De termen zijn niet in balans omdat sommige variabelen worden gebonden door een $\lambda$-operator die niet de onderste is die nog niet is afgesloten. In het tweede voorbeeld wordt $x$ door $\lambda x$ gebonden, terwijl $\lambda y$ nog niet is afgesloten.*

**Termen uit balans:** *De volgende termen zijn niet in balans of in scope-balans. We moeten de jump-semantiek hanteren voor de $\text{ʎ}$-operatoren:*

$$\lambda xyz.y\text{ʎ}y\ x$$
$$\lambda xyz.\text{ʎ}x\ x$$
$$\lambda xy.x\text{ʎ}x\ y$$

*Deze termen zijn niet af te leiden met de afleidingsregels voor termen in scope-balans. We zien ook dat de $\text{ʎ}$-operatoren in de termen steeds een $\lambda$-operator afsluiten die niet het laagst boven hen in de term zit. Voor deze termen moeten we de* jump-semantiek *gebruiken.*

Uit de definities van balans en scope-balans volgt:

- Alle termen die in balans zijn, zijn ook in scope-balans.
- Alle normale $\lambda$-termen zijn in scope-balans, omdat er geen $\text{ʎ}$-operatoren in voor komen.

### 2.2.5 De Bruijn-notatie

In [4] wordt een notatie voor $\lambda$-termen beschreven waarbij geen variabelenamen meer nodig zijn. De $\lambda$-operatoren komen voor zonder variabelenaam en de variabelen worden weergegeven door een index. Deze index staat voor het aantal $\lambda$-operatoren dat je moet overslaan voordat je bij de operator komt die deze variabele bindt. Bijvoorbeeld:

$$\lambda xyz.xz(yz) \rightarrow_{DB} \lambda\lambda\lambda 20(10)$$

(Met $\rightarrow_{DB}$ de functie die $\lambda$-termen naar hun De Bruijn representatie vertaalt) Het voordeel van deze notatie is dat alle termen die $\alpha$-equivalent zijn, hetzelfde worden genoteerd. Eén De Bruijn term correspondeert dus met een klasse van $\alpha$-equivalente termen. Als we een term substitueren dan hoeven we deze niet meer te herbenoemen. Na de substitutie is het eventueel wel nodig om enkele indices te updaten.

$$\lambda((\lambda\lambda\lambda\mathbf{2}0(10))(\lambda 01)) \ \rightarrow_{\beta}\ \lambda\lambda\lambda(\lambda 0\mathbf{3})0(10)$$

We moeten de gesubstitueerde 1 ophogen tot 3 om te zorgen dat hij door de juiste $\lambda$-operator wordt gebonden.

We kunnen indices in de De Bruijn-termen ook zien als het aantal scopes dat we zouden moeten afsluiten voordat we de variabele mogen plaatsen in een gebalanceerde term. Deze correspondentie wordt nog duidelijker als we de getallen weergeven met behulp van 0 en de successor-operator:

$$\lambda xyz.((\text{ʎ}zy.x)z)((\text{ʎ}z\ y)z) \mapsto \lambda\lambda\lambda(s(s(0))0)(s(0)0)$$

Het aantal $s$-en en het aantal $ʎ$ operatoren is gelijk. De overeenkomst tussen het aantal successor- en $ʎ$-operatoren kan echter verdwijnen als we een $\beta$-stap zetten. Bijvoorbeeld:

$$\lambda x\ ((\lambda xyz.((ʎzy.x)z)((ʎz\ y)z))(\lambda x\ x(ʎx\ x))) \rightarrow_\beta$$

$$\lambda xyz.((ʎzy.(\lambda x\ x(ʎx\ x)))z)((ʎz\ y)z))$$

In tegenstelling tot de successor-operator, die enkel op indices (variabelen) kan worden toegepast, kan de $ʎ$-operator worden toegepast op een subterm. We hoeven de term niet up te daten na de substitutie. De $ʎ$-operatoren blijven dus hoger in de boom zitten dan de successor-operatoren, die steeds naar de bladeren, de variabelen, worden geduwd. Dit is een voordeel, want daardoor hebben we al een vorm van sharing (zie voor meer hierover [5]). Er is echter wel een ander probleem waar we mee te maken krijgen, namelijk dat van de verborgen redexen. Zie voor de behandeling van dit probleem Paragraaf 2.2.7.

### 2.2.6 $\alpha$-conversie

Ook in de $ʎ$-calculus zijn twee termen $\alpha$-equivalent als ze dezelfde functie uitdrukken. Dat wil zeggen dat beide termen in vorm gelijk zijn. De binders en bijbehorende gebonden variabelen staan in beide termen op dezelfde plaats. Ze hebben alleen een andere naam. Maar door het herbenoemen van de binders en de bijbehorende variabelen, kunnen we de termen gelijk maken. Als we willen weten of twee termen $\alpha$-equivalent zijn moeten we onderzoeken of de verschillen enkel zitten tussen de namen van de paren van binders en hun bijbehorende variabelen. Hier zijn een paar verschillende methoden voor. Wij bekijken hier de methode van Kahrs, omdat die vrij eenvoudig uit te breiden is naar de $ʎ$-calculus.

Kahrs' manier om dit te onderzoeken is gebaseerd op inductie. In zijn methode gebruikt hij een stack om bij te houden welke binders er in een term zitten, de *scopestack*. Zijn methode gaat als volgt:

**Definitie 15 ($\alpha$-gelijkheid volgens Kahrs)** *We beginnen met het plaatsen van een lege scopestack voor beide termen, dus:*

$$M =_\alpha N \Rightarrow \Box M =_\alpha \Box N$$

*Daarna worden beide termen op recursieve wijze afgebroken tot bewezen is dat beide termen gelijk zijn. De recursieve regels die hierbij gebruikt worden, zijn de volgende:*

*Applicatie*

$$\overrightarrow{x}(M_1 M_2) =_\alpha \overrightarrow{y}(N_1 N_2) \Rightarrow \overrightarrow{x} M_1 =_\alpha \overrightarrow{y} N_1 \wedge \overrightarrow{x} M_2 =_\alpha \overrightarrow{y} N_2$$

*Abstractie*

$$\overrightarrow{x}(\lambda x\ M) =_\alpha \overrightarrow{y}(\lambda y\ N) \Rightarrow x\overrightarrow{x}M =_\alpha y\overrightarrow{y}N$$

*Variabelen*

$$\overrightarrow{x}x =_\alpha \overrightarrow{y}y \Rightarrow \overrightarrow{x} = x\overrightarrow{x}' \wedge \overrightarrow{y} = y\overrightarrow{y}'$$

$$\overrightarrow{x}x =_\alpha \overrightarrow{y}y \Rightarrow \overrightarrow{x} = x'\overrightarrow{x}' \wedge \overrightarrow{y} = y'\overrightarrow{y}' \wedge x \neq x' \wedge y \neq y' \wedge \overrightarrow{x}'x =_\alpha \overrightarrow{y}'y$$

$$\overrightarrow{x}x =_\alpha \overrightarrow{y}y \Rightarrow \overrightarrow{x} = \Box \wedge \overrightarrow{y} = \Box \wedge x = y$$

In de regels wordt bij het variabelegeval rekening gehouden met vrije variabelen die eventueel in de termen kunnen voorkomen. Deze variabelen moeten in beide termen dezelfde naam hebben.

Om Kahrs' methode uit te breiden naar de $\reflectbox{$\lambda$}$-calculus hoeven we alleen een recursieve regel te geven voor de end-of-scope operator. Dus:

**Definitie 16 ($\alpha$-gelijkheid voor $\reflectbox{$\lambda$}$-termen volgens Kahrs)**

*End of Scope*

$$\overrightarrow{x}(\reflectbox{$\lambda$} x\ M) =_\alpha \overrightarrow{y}(\reflectbox{$\lambda$} y\ N) \Rightarrow \overrightarrow{x} = x\overrightarrow{x}' \wedge \overrightarrow{y} = y\overrightarrow{y}' \wedge \overrightarrow{x}'M =_\alpha \overrightarrow{y}'N$$

Deze definitie werkt alleen voor termen die in balans of in scope-balans zijn.

### 2.2.7 $\beta$-reductie

De toevoeging van de end-of-scope operator kan in verband met $\beta$-reductie een probleem opleveren. Laten we, om dit toe te lichten, eerst nog eens kijken hoe een $\beta$-redex in zijn werk gaat:

$$(\lambda x\ M)N \ \rightarrow_\beta \ M[x := N]$$

We hebben in Paragraaf 2.2.1 gezien dat de $\reflectbox{$\lambda$}$-operator kan worden toegepast op een subterm. Dit heeft echter als gevolg dat deze operator een redex in de weg kan zitten. We spreken dan van een verborgen redex:

**Definitie 17 (Verborgen redexen)** *Een verborgen redex is van de volgende vorm:*

$$(\reflectbox{$\lambda$}\overrightarrow{y}\lambda x\ M)N$$

*Waarbij* $\overrightarrow{y} \neq \Box$

In de $\lambda$-term die met deze $\curlywedge$-term correspondeert, staan deze $\curlywedge$-operatoren er niet en zonder de $\curlywedge$-operatoren is er een $\beta$-redex. We willen dat in een $\curlywedge$-term dezelfde stappen kunnen worden gedaan als in de corresponderende $\lambda$-term. We zullen de definitie van $\beta$-reductie dus moeten aanpassen voor de $\curlywedge$-calculus. Het idee is dat we de informatie van de $\curlywedge$-operatoren die in de weg zitten tijdens de substitutie meenemen.

**Definitie 18 ($\beta$-reductie voor $\curlywedge$-termen)**

$$(\curlywedge\overrightarrow{y}.\lambda x M)N \ \to_\beta \ M[\overrightarrow{y}, x := N, \Box]$$

Merk op dat er in de substitutie aan de rechterkant een derde argument is (in eerste instantie een lege stack). Het zorgt ervoor dat we kunnen bepalen dat een voorkomen van $x$ in $M$ overeenkomt met de variabele waarvoor we moeten substitueren. Om precies te zijn, er wordt een variabelenaam op de stack gezet als we een abstractie tegenkomen en er wordt een element verwijderd als we een end-of-scope tegenkomen. Voor substitutie gebruiken we de volgende inductieve definitie:

**Definitie 19 (Substitutie in $\curlywedge$-calculus)** **Applicatie**

$$(M_1 M_2)[\overrightarrow{y}, x := N, \overrightarrow{z}] \Rightarrow (M_1[\overrightarrow{y}, x := N, \overrightarrow{z}] M_2[\overrightarrow{y}, x := N, \overrightarrow{z}])$$

*We zien hier dat de substitutie onveranderd de beide subtermen in wordt geduwd.*

**Abstractie**

$$(\lambda z \ M)[\overrightarrow{y}, x := N, \overrightarrow{z}] \Rightarrow (\lambda z \ M[\overrightarrow{y}, x := N, z\overrightarrow{z}])$$

*Als de substitutie over $\lambda$-operator gaat, komt hij een nieuwe scope binnen. De naam van deze scope slaan we op in de scopestack.*

**Variabelen**

*Met $x \neq z$:*

$$\begin{aligned} x[\overrightarrow{y}, x := N, \overrightarrow{z}] &= x \ \textit{als} \ x \in \overrightarrow{z} \\ x[\overrightarrow{y}, x := N, \overrightarrow{z}] &= \curlywedge\overrightarrow{z} N \ \textit{als} \ x \notin \overrightarrow{z} \\ z[\overrightarrow{y}, x := N, \overrightarrow{z}] &= z \ \textit{als} \ z \in \overrightarrow{z} \\ z[\overrightarrow{y}, x := N, \overrightarrow{z}] &= \curlywedge\overrightarrow{z}\curlywedge\overrightarrow{y} z \ \textit{als} \ z \notin \overrightarrow{z} \end{aligned}$$

*Het variabelegeval is het meest complex. We bespreken per geval waarom de regel er zo uitziet.*

- *In het eerste geval heeft de variabele wel de goede naam, maar is er later een scope met dezelfde naam geopend. De variabele behoort tot deze later geopende scope. We moeten niet substitueren en de variabele blijft dus staan.*
- *In het tweede geval komen we een variabele tegen met de juiste naam en er is geen latere scope geopend met dezelfde naam. We moeten dus substitueren. We weten dat de geopende scopes moeten worden afgesloten, omdat deze geen invloed mogen hebben op de subterm die wordt gesubstitueerd (dit scheelt herbenoemen). Op deze manier worden er dus nieuwe* $\reflectbox{\ensuremath{\lambda}}$*-operatoren geproduceerd. De stack met de originele* $\reflectbox{\ensuremath{\lambda}}$*-operatoren zetten we hier niet neer, omdat er in de subterm die wordt gesubstitueerd variabelen kunnen voorkomen die behoren tot een scope die door een van deze operatoren wordt afgesloten.*
- *In het derde geval komen we een variabele tegen met een andere naam. We moeten hier dus niet substitueren. De variabele behoort tot een scope die geopend wordt in de subterm waarin we substitueren. We kunnen de* $\reflectbox{\ensuremath{\lambda}}$*-operatoren hier niet neerzetten, omdat we dan niet meer in scope balans zijn.*
- *In het vierde geval hebben we ook te maken met een andere naam en moeten we ook niet substitueren. Deze variabele behoort echter tot een scope die buiten de subterm ligt waarin we substitueren. We kunnen dus alle scope die geopend worden in de subterm sluiten en bovendien de lijst met* $\reflectbox{\ensuremath{\lambda}}$*-operatoren neerzetten. Ook hier worden nieuwe* $\reflectbox{\ensuremath{\lambda}}$*-operatoren gecreëerd.*

**End-of-scope**

$$
\begin{aligned}
&(\reflectbox{\ensuremath{\lambda}} zM)[\overrightarrow{y}, x := N, z\overrightarrow{z}] = \reflectbox{\ensuremath{\lambda}} zM[\overrightarrow{y}, x := N, \overrightarrow{z}] \\
&(\reflectbox{\ensuremath{\lambda}} xM)[\overrightarrow{y}, x := N, \Box] = \reflectbox{\ensuremath{\lambda}} \overrightarrow{y} M
\end{aligned}
$$

- *In het eerste geval komen we het einde van een scope tegen die in de subterm waarin we substitueren is geopend. De naam van deze scope is de laatste die op de scopestack staat (wegens scope-balans). Deze naam halen we van de stack af en we substitueren verder in de body van de* $\reflectbox{\ensuremath{\lambda}}$*-operator.*
- *In het tweede geval komen we het einde van de scope tegen van de substituerende* $\lambda$*-operator. Er zullen dus geen variabelen meer in de subterm voorkomen die gebonden zijn door deze* $\lambda$*. We stoppen de substitutie en we zetten alle* $\reflectbox{\ensuremath{\lambda}}$*-operatoren neer. Merk wel op dat we de* $\reflectbox{\ensuremath{\lambda}}$*-operator die we net tegenkwamen,* $\reflectbox{\ensuremath{\lambda}} x$*, niet in het resultaat neerzetten. De* $\lambda$*-operator die de scope opende die deze* $\reflectbox{\ensuremath{\lambda}}$*-operator afsloot, is namelijk verdwenen bij het zetten van de* $\beta$*-stap.*

**Voorbeeld 13 ($\beta$-reductie van $\reflectbox{$\lambda$}$-termen)**

$$\lambda xyz.(\reflectbox{$\lambda$} zy.\lambda vw.(wv\reflectbox{$\lambda$} wv.x))z \ \to_\beta \ \lambda xyz.(\lambda w \ (wv\reflectbox{$\lambda$} wv.x))[[zy], v := z, \square]$$

$$\Rightarrow \lambda xyzw.(wv\reflectbox{$\lambda$} wv.x)[[zy], v := z, [w]]$$

$$\Rightarrow \lambda xyzw.(wv[[zy], v := z, [w]]\reflectbox{$\lambda$} wv.x[[zy], v := z, [w]])$$

$$\Rightarrow_2 \lambda xyzw.(w[[zy], v := z, [w]]v[[zy], v := z, [w]]\reflectbox{$\lambda$} w \ (\reflectbox{$\lambda$} v \ x)[[zy], v := z, \square])$$

$$\Rightarrow_3 \lambda xyzw.((w\reflectbox{$\lambda$} w.z)\reflectbox{$\lambda$} wzy.x)$$

We zien in dit voorbeeld dat de scope-balans bewaard blijft. Omdat er meer $\reflectbox{$\lambda$}$-operatoren ontstaan gaan we van scope-balans steeds meer richting complete balans (het resultaat is hier zelfs in balans).

Een nadeel van $\beta$-reductie, zoals deze nu is gedefinieerd, is dat hij niet lokaal is. Er kunnen een willekeurig aantal $\reflectbox{$\lambda$}$-operatoren tussen de $\lambda$-operator en de applicatie zitten. In de volgende hoofdstukken geven we een graafrepresentatie van $\reflectbox{$\lambda$}$-termen, waarmee we een lokale versie van $\beta$-reductie kunnen definiëren.

# 3 Van termen naar grafen

Het doel van deze scriptie is om een lokale, grafische representatie te geven van de $\lambda$-calculus, uitgebreid met expliciete scope-operatoren. Het grafisch representeren van de $\lambda$-calculus werd voor het eerst gedaan door Wadsworth in 1971 [12]. Een aantal van zijn principes zullen uiteindelijk in onze graafrepresentatie terugkomen.

We zullen eerst de correspondentie tussen de objecten van beide calculi moeten vaststellen. Vandaar dat we hier beginnen met de definitie van een vertaalfunctie die termen naar grafen vertaalt. Met termen bedoelen we hier gesloten $\lambda$-termen. Deze bevatten dus geen vrije variabelen en geen $\reflectbox{$\lambda$}$-operatoren. Tijdens de vertaling zullen we ervoor zorgen dat de term wordt vertaald naar een equivalente graaf die volledig in balans is. Door de verschillende hoofdstukken we maken we gebruik van een lopend voorbeeld, een $\lambda$-term aan de hand waarvan we verschillende dingen kunnen demonstreren. De $\lambda$-term die we hiervoor gebruiken is:

$$T = \lambda x \ \lambda y \ xy$$

## 3.1 Een voorbeeld - Naïeve vertaalfunctie

In deze paragraaf schets ik hoe ik aan het idee ben gekomen van hoe de vertaalfunctie eruit moet zien. Het heeft dan ook een informeel en beschrijvend karakter. In Paragraaf 3.2 zal ik een formele definitie geven van de grafen die we gebruiken om de termen te representeren en in Paragraaf 3.3 geef ik de formele definitie van de vertaalfunctie.

### 3.1.1 De abstracte syntax-boom

We kunnen de termen van de $\lambda$-calculus ook als een boom noteren. De operatoren en de variabelen vormen de knopen in de boom. In een boom-notatie is de structuur van de $\lambda$-term veel duidelijker. In zo'n boom onderscheiden we drie soorten knopen, de applicatie-, abstractie- en variabeleknoop. De knoopsoorten zijn te onderscheiden door hun labels. De applicatieknoop heeft @ als label, de abstractieknoop heeft $\lambda$ als label en de variabeleknoop heeft een variabelenaam als label. Iedere knoopsoort heeft een vast aantal takken. De applicatieknoop heeft er twee, omdat er twee subbomen aan gekoppeld moeten worden. Deze subbomen zijn representaties van de subtermen die op elkaar worden toegepast. De abstractieknoop heeft ook twee takken, één voor de body en één voor een variabeleknoop, die de representatie is van de variabelenaam die bij de $\lambda$-operator hoort. De variabeleknoop heeft geen takken. De variabeleknopen zijn daardoor de bladeren van de boom. De knoopsoorten in de bomen die we gebruiken hebben een vaste ariteit. Dat wil zeggen dat de soort van de knoop bepaalt hoeveel takken eraan verbonden kunnen worden. Hierdoor vallen ze binnen de definitie van abstracte syntax-bomen.

We kunnen een vertaalfunctie tussen termen en abstracte syntax-bomen inductief over de syntax van termen definiëren:

**Definitie 20 (Vertaling van termen naar abstracte syntax-bomen)**

$$F : T \rightarrow AST$$

*Waarbij $T$ een term en $AST$ een abstracte syntax-boom*

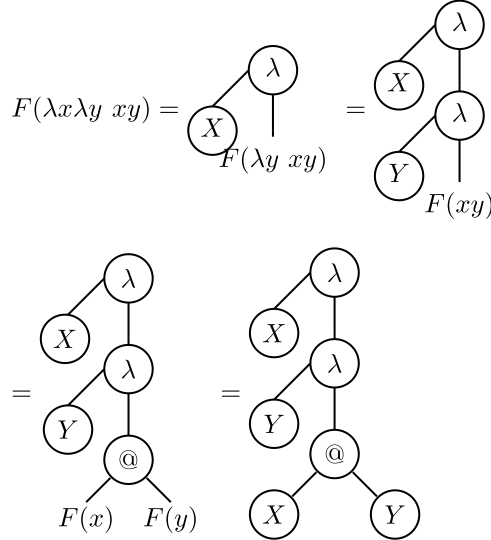
$F(x) = $ (X)        $F(\lambda x\ M) = $ ($\lambda$) met takken (X) en $F(M)$

$F(MN) = $ (@) met takken $F(M)$ en $F(N)$

*Waarbij $x$ staat voor een willekeurige variabele en $M$ en $N$ staan voor willekeurige termen.*

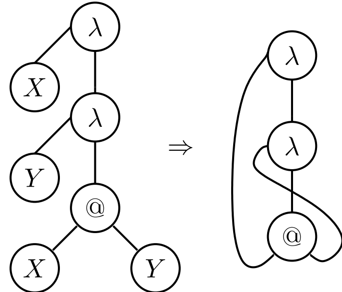
Als we deze functie op ons lopende voorbeeld toepassen, dan krijgen we het volgende resultaat:

**Voorbeeld 14 (Lopend voorbeeld vertaald naar abstracte syntax-boom)**




### 3.1.2 Lokaliteit

Er is echter een probleem met de abstracte syntax-boom. Als we een $\beta$-stap willen zetten, moeten we de hele vertaling van de body afzoeken naar de substitutievariabelen. De representatie van de $\beta$-reductie is geen lokale herschrijfregel. Om te zorgen dat deze regel lokaal wordt, zullen we de variabelen vertalen als een tak van de plaats waar de variabeleknoop zou hebben gezeten naar de $\lambda$-knoop waar ze bij horen. De binding wordt zo expliciet weergegeven. Zo'n tak zullen we een *backpointer* noemen. Onze graaf moet als volgt worden aangepast:

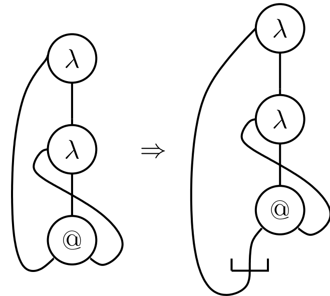



We zien dat we nu de variabelenamen niet meer nodig hebben. Daarom vallen ook de variabeleknopen bij de $\lambda$-knopen weg. Onze grafen representeren dus een klasse van $\alpha$-equivalente termen (zie voor een bewijs Paragraaf 3.4).

### 3.1.3 Grafen in balans

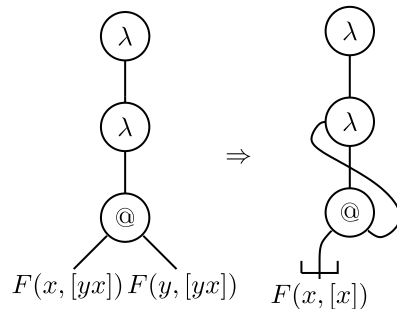
We hebben al genoemd dat we termen willen vertalen naar gebalanceerde grafen. Om dit voor elkaar te krijgen moeten alle scopes die later zijn geopend dan de

scope die een variabele bindt, afgesloten zijn voordat deze variabele vertaald kan worden. De vertaling van ons lopende voorbeeld is nog niet in balans. De vertaling van $x$ wordt door de bovenste $\lambda$-operator gebonden, terwijl de scope van de onderste $\lambda$-operator nog niet afgesloten is. Om de scope van de onderste $\lambda$ af te sluiten, maken we gebruik van de ⅄-knoop. Deze zullen we in de graaf representeren als een vierkante haak. Het resultaat van ons lopende voorbeeld gaat er als volgt uitzien:




### 3.1.4 De vertaalfunctie aangepast

Als we deze veranderingen in ons resultaat willen terugzien, dan zullen we de vertaalfunctie moeten aanpassen. Het in balans brengen van de graaf is niet zo moeilijk. We maken de functie tweeplaatsig, zodat we op de tweede plaats een scopestack kunnen plaatsen. In deze stack kunnen we opslaan welke scopes er geopend zijn en in welke volgorde. Steeds als we een $\lambda$-operator vertalen, zetten we de variabelenaam die erbij hoort bovenop de stack. Als we een variabele willen vertalen, dan kan dat alleen als de naam van deze variabele boven op de stack staat. Als dit niet het geval is, zijn er scopes geopend na de scope waar deze variabele bij hoort. We plaatsen dan een ⅄-knoop en halen de bovenste naam van de stack. We plaatsen net zo lang ⅄-knopen tot we de juiste variabelenaam tegenkomen. Pas dan vertalen we de variabele. Zo krijgen we een resultaat dat in balans is. Om dit mechanisme te illustreren, nemen we een momentopname uit ons lopende voorbeeld:




In dit voorbeeld gaan we er impliciet van uit dat we weten aan welke $\lambda$-knoop een backpointer verbonden moet worden. We zullen dit verder in Paragraaf 3.3 behandelen.

## 3.2 De grafen

Voordat we de vertaalfunctie definiëren, zullen we eerst een formele definitie moeten geven van de grafen die we als resultaat van onze functie willen. De grafen die we gaan gebruiken bestaan uit een aantal verschillende soorten knopen met takken ertussen. Deze takken zijn ongericht. We definiëren per operator een knoopsoort. We krijgen dus te maken met een applicatieknoop, een $\lambda$-knoop en een ⅄-knoop. De knopen zijn te onderscheiden doordat iedere soort een eigen label krijgt. De knopen hebben allemaal een aantal poorten. De takken tussen de knopen zijn verbonden aan deze poorten. Iedere knoop heeft een interactiepoort. Als twee knopen met hun interactiepoort aan elkaar verbonden zijn, kan er interactie plaatsvinden tussen beide knopen. Deze interactie wordt vastgelegd in herschrijfregels die we definiëren in Hoofdstuk 5. Het aantal poorten per knoop en welke van deze poorten de interactiepoort is, wordt door het soort van de knoop bepaalt. Door dit gedrag lijken onze grafen veel op interactienetten (zie [7]).

**Definitie 21 (De applicatieknoop)** *De applicatieknoop is te herkennen aan het label @. De applicatie-knoop heeft een bovenpoort en twee onderpoorten die we de linker- en de rechterpoort zullen noemen. De linkerpoort is de interactiepoort.*

**Definitie 22 (De $\lambda$-knoop)** *De $\lambda$-knoop is te herkennen aan het label $\lambda$. De $\lambda$-knoop heeft een bovenpoort, een body-poort en een variabelenpoort. De variabelenpoort heeft als eigenschap dat er een willekeurig aantal takken aan kunnen worden bevestigd. De bovenpoort is de interactiepoort.*

**Definitie 23 (De end-of-scopeknoop)** *De end-of-scopeknoop, of ⅄-knoop, heeft de vorm van een vierkante haak. De end-of-scopeknoop heeft een bovenpoort en een onderpoort, die we ook wel de body-poort noemen. De body-poort is de interactiepoort.*

Om een graaf te krijgen moeten we de knopen aan elkaar kunnen verbinden. Dit doen we door middel van takken. Deze definiëren we als volgt:

**Definitie 24 (Tak)** *Een tak is een verbinding tussen twee knopen. De uiteinden van een tak moeten allebei aan een poort van een knoop verbonden zijn.*

Aan deze definitie zien we dat we de takken zo simpel mogelijk houden. Ze zijn ongericht en ongelabeld. Minimale eisen waar de graaf aan moet voldoen zijn de volgende:

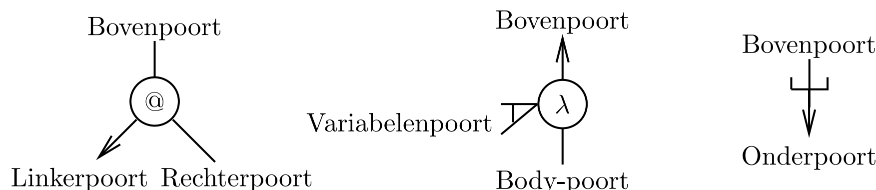
**Definitie 25 (Graaf)** *Een graaf is een verzameling knopen die verbonden zijn door takken. De knoopsoorten die mogen worden gebruikt liggen vast in een signatuur. Verder moet een graaf aan de volgende voorwaarden voldoen:*

- *Aan elke poort van elke knoop zit precies één tak. Uitzondering hierop is de variabelenpoort van de $\lambda$-knoop, waar $i$ takken aan mogen zitten, met $0 \leq i < n$, met $n$ een willekeurig natuurlijk getal.*

- *Een tak loopt altijd van een niet-bovenpoort naar een bovenpoort, of van een niet-bovenpoort naar een variabelenpoort. Als we de backpointers uit een graaf weghalen, hou je een boomstructuur over.*

- *Iedere graaf heeft één knoop, te herkennen aan het label R (van* root*), die we de wortelwijzer noemen. Deze knoop heeft alleen een onderpoort en is verbonden aan de bovenpoort van een andere knoop. De knoop die aan deze knoop verbonden is noemen we de wortel van de graaf.*

Onze signatuur bestaat (voorlopig) uit de volgende knopen:




Waarbij de pijlen de interactiepoorten van de knopen weergeven.

**Definitie 26 (Een pad door een graaf)** *Een pad door een graaf gaat van knoop naar knoop over de takken tussen de knopen. Voor een pad gelden de volgende regels:*

- *Een pad verlaat een knoop nooit via de poort waarlangs hij de knoop binnen kwam.*

- *Een $\lambda$-knoop wordt niet verlaten via een variabelenpoort.*

- *Een pad stopt in ieder geval als een $\lambda$-knoop wordt bereikt via een variabelenpoort.*

Uit deze definitie volgt dat een pad altijd stopt als de wortelwijzer wordt bereikt. Deze knoop heeft slechts ëën poort en het pad kan de knoop niet meer verlaten. Onze grafen zijn cyclisch, maar in de pad-definitie wordt dit ondervangen doordat een pad stopt als we een $\lambda$-knoop middels een variabelenpoort bereiken. Dit is namelijk precies de plek waar de circulariteit van onze grafen ontstaat. Door daar te stoppen zal een pad nooit twee keer over dezelfde tak gaan en zijn paden dus niet cyclisch. Met behulp van deze definitie kunnen we de volgende twee richtingen in een graaf definiëren:

**Definitie 27 (Een pad in de graaf naar boven)** *Voor een pad in de graaf naar boven geldt ook de volgende voorwaarde:*

- *Het pad moet elke knoop die hij tegenkomt verlaten via de bovenpoort.*

Omdat elke knoop maar één bovenpoort heeft, is er vanaf een bepaalde knoop altijd maar één pad naar boven mogelijk.

**Definitie 28 (Een pad naar beneden)** *Voor een pad naar beneden gelden de volgende voorwaarden:*

- *Een knoop mag niet worden verlaten via een bovenpoort of een variabelenpoort.*

Bij de applicatieknoop moet het pad een keuze maken. Het kan de poort via de linker- en via de rechterpoort verlaten.

**Definitie 29 (Volledig pad)** *Een pad naar boven vanuit een knoop is volledig als het pad doorloopt tot we de wortelwijzer bereikt hebben. Een pad naar beneden vanuit een knoop is volledig als het doorloopt tot we de variabelenpoort van een $\lambda$-knoop tegenkomen.*

**Definitie 30 (Paden in balans)** *Een pad is in balans als geldt:*

- *Elke scope die op het pad wordt geopend, wordt op datzelfde pad weer gesloten.*

Dit heeft als consequentie dat als er een $\lambda$-knoop voorkomt op een pad in balans, er ëën van de volgende twee dingen moet gelden:

- Het pad komt terug bij de $\lambda$-knoop middels de variabelenpoort.
- Op het pad bevindt zich een $\reflectbox{$\lambda$}$-knoop die de scope van de $\lambda$-knoop afsluit.

We zullen in Hoofdstuk 6 zien dat het begrip balans een belangrijke rol speelt in de correctheid van de read-back-functie.

**Definitie 31 (Grafen in balans)** *Een graaf is in balans als elk pad naar beneden dat van een $\lambda$-knoop naar zichzelf loopt, in balans is.*

## 3.3 Een recursieve definitie van de vertaalfunctie

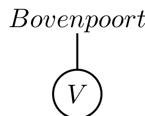
We geven nu een recursieve definitie van onze vertaalfunctie. Hiervoor gebruiken we de inductieve syntax van de $\lambda$-termen, uitgebreid met een scopestack. We zullen eerst per geval een intuïtieve bespreking geven over welke vorm de definitie zal aannemen, dus eerst voor variabelen, dan voor abstractie en dan voor applicatie. In Paragraaf 3.3.5 zullen we uiteindelijk de formele definitie geven van de vertaalfunctie.

### 3.3.1 De variabele

In Paragraaf 3.1.2 hebben we gezien dat we een variabele willen vertalen als een backpointer. Als we nu een vertaling willen geven van $F(x, x\overrightarrow{s})$ dan hebben we een probleem. De tak die we willen hebben kan op zichzelf niet bestaan. Een tak moet volgens Definitie 24 twee knopen aan elkaar verbinden. We moeten dus bij de vertaling ook gelijk twee knopen als resultaat hebben. De eerste van deze knopen ligt voor de hand. De plaats van de variabele is momenteel de hoogste in de graaf en wordt dus aangewezen door de wortelwijzer. De ene kant van de tak kunnen we dus aan de wortelwijzer verbinden. De andere kant zou verbonden moeten worden aan de vertaling van de $\lambda$-operator waar deze knoop

bij hoort. Maar we hebben nog geen vertaling van deze $\lambda$-operator. We zien echter aan het voorste element van de stack dat de variabele wel in een scope thuishoort. We zullen een grafische representatie van de stack maken. Daarvoor definiëren we een nieuwe knoopsoort, de stack-knoop, met label $V$. We kunnen de elementen van de stack dan representeren als een rij stack-knopen. Deze rij bevat evenveel elementen als de stack. Omdat de elementen in een stack geordend zijn, zullen we de knopen nummeren zodat de volgorde niet verloren gaat. De stack-knopen krijgen de namen $v_1$ tot en met $v_n$, waarbij $n$ de lengte is van de stack. Het bovenste element van de stack wordt gerepresenteerd door $v_n$. De variabele kan nu worden vertaald als een tak tussen de wortelwijzer en $v_n$, de stack-knoop met het hoogste nummer. Als we later meer variabelen tegenkomen die door dezelfde scope gebonden worden dan moeten we ervoor zorgen dat deze ook vertaald worden als een tak naar deze knoop. Hoe dit gebeurt zien we in de vertaling van de applicatie (Paragraaf 3.3.4). De stack-knopen zijn als volgt gedefinieerd:

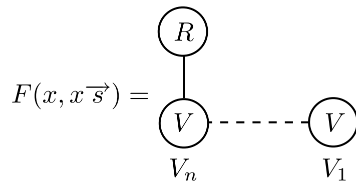
**Definitie 32 (De stack-knoop)** *De stack-knoop is te herkennen aan het label $V$. De stack-knoop heeft een bovenpoort en verder geen poorten. Ook heeft hij geen interactiepoort. De knoop ziet er als volgt uit:*




Deze knoopsoort heeft geen interactiepoort en zal dus niet deelnemen aan de interacties die plaats kunnen vinden in de graaf. Dit komt doordat de stack-knopen uiteindelijk allemaal weer uit de graaf zullen verdwijnen. Als we een $\lambda$-operator vertalen, halen we de stack-knoop die de scope van de $\lambda$-operator representeert, weg en zetten er een $\lambda$-knoop voor in de plaats (zie het $\lambda$-geval, paragraaf 3.3.3). Bij het vertalen van een gesloten term zullen alle scope-knopen in het uiteindelijke resultaat verdwenen zijn.

**Definitie 33 (Welgevormde grafen)** *Een graaf die geen scope-knopen bevat, noemen we een welgevormde graaf.*

De vertaling van het variabelegeval ziet er dus als volgt uit:




Waarbij *Lengte* $x\overrightarrow{s} = n$

### 3.3.2 De variabele, ongebalanceerd

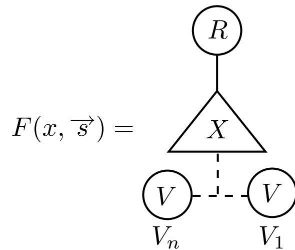
Vaak komt het voor dat een variabele vertaald moet worden waarvan de naam niet voorop de stack staat: $F(x, y\overrightarrow{s})$. We hebben in Paragraaf 3.1.4 al gezien

dat we de stack dan recursief afbreken tot de naam van de variabele wel voorop de stack staat. Terwijl dit gebeurt moeten er end-of-scope-knopen ontstaan. We moeten de vertaling van $F(x, \overrightarrow{s})$ zo manipuleren dat er een end-of-scopeknoop ontstaat en dat dit ook de nieuwe wortel wordt. Deze manipulatie noemen we de end-of-scopestap, $E : G \rightarrow G$. We krijgen dus:

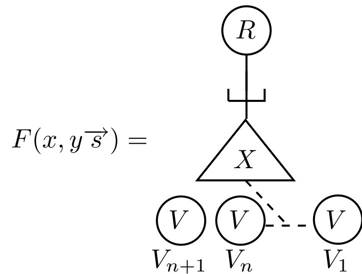
$$F(x, y\overrightarrow{s}) = E(F(x, \overrightarrow{s}))$$

Merk op dat de kleinere graaf minder informatie bevat dan de grote. De $y$ die eerst op de stack staat is in de volgende stap verdwenen. Deze informatie is echter niet relevant. We sluiten altijd de laatst geopende scope af met een ⅄-knoop en hoe die scope heet is niet van belang, want er zullen toch geen variabelen meer worden gebonden door deze scope.

Om te weten te komen wat de end-of-scopestap precies inhoudt, bekijken we de verschillen tussen de vertalingen van $F(x, y\overrightarrow{s})$ en $F(x, \overrightarrow{s})$. Van $F(x, \overrightarrow{s})$ weten we niet precies hoe deze eruit ziet. We nemen daarom een willekeurige, algemene graaf. We weten van willekeurige grafen dat er een knoop de wortel is, aangewezen door de wortelwijzer. Verder bestaat de graaf uit knopen met takken ertussen. We weten dat enkele van deze knopen de stack representeren. Als conventie tekenen we deze stack-knopen in een horizontale rij onder de rest van de graaf. De wortelwijzer tekenen we als bovenste knoop van de graaf, zodat de graaf overzichtelijker wordt. Het geheel ziet er dan als volgt uit:




We weten dan dat de vertaling van $F(x, y\overrightarrow{s})$ er onder dezelfde conventies als volgt uitziet:




De end-of-scopestap, $E : G \rightarrow G$, bestaat dus uit de volgende manipulaties:

- Er wordt een nieuwe end-of-scopeknoop gecreëerd, die tussen de oude wortel en de wortelwijzer wordt geplaatst. Deze knoop wordt de nieuwe wortel van de graaf.

- Er wordt een nieuwe $V$-knoop gecreëerd, met de naam $V_{n+1}$, waarbij $n$ het nummer is van de hoogst genummerde $V$-knoop.

De tweede manipulatie doen we zodat de rij van $V$-knopen een correcte representatie van de stack blijft.

### 3.3.3 De $\lambda$-stap

Laten we de $\lambda$-regel bekijken uit Definitie 5:

> Als $M$ een term is en $x$ een variabele, dan is $(\lambda x\ M)$ een term

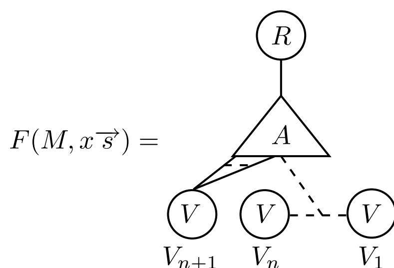
Als we dit willen omzetten naar grafen, dan krijgen we:

> $F(M, x\overrightarrow{s})$ is een graaf $\Rightarrow F(\lambda x\ M, \overrightarrow{s})$ is een graaf

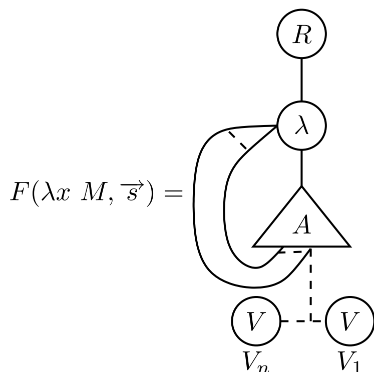
We zullen moeten kijken welke manipulaties nodig zijn om van $F(M, x\overrightarrow{s})$ naar $F(\lambda x\ M, \overrightarrow{s})$ te komen. Als we deze manipulaties samenvatten in de $\lambda$-stap, $\Lambda : G \rightarrow G$, dan kunnen we de functie als volgt definiëren:

$$F(\lambda x\ M, \overrightarrow{s}) = \Lambda(F(M, x\overrightarrow{s}))$$

Om te bekijken wat de $\lambda$-stap precies inhoudt bekijken we weer de verschillen tussen de resultaten van $F(M, x\overrightarrow{s})$ en $F(\lambda x\ M, \overrightarrow{s})$:




En:




De $\lambda$-stap bestaat dus uit de volgende graaf-manipulaties:

- Er wordt een nieuwe $\lambda$-knoop gecreëerd.
- Deze knoop wordt tussen de wortel en de wortelwijzer geplaatst.
- Alle takken die aan de $V$-knoop met het hoogste nummer zitten, worden aan de variabelenpoort van de nieuwe $\lambda$-knoop verbonden.
- De $V$-knoop met het hoogste nummer verdwijnt.

### 3.3.4 De applicatie-stap

De applicatie zullen we op analoge wijze behandelen als de $\lambda$. We bekijken de applicatie-regel uit Definitie 5:

> Als $M, N$ termen zijn, dan is $(MN)$ een term

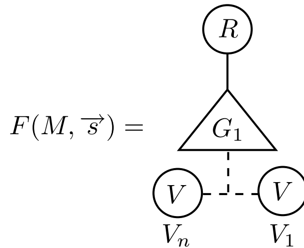
Dus voor grafen:

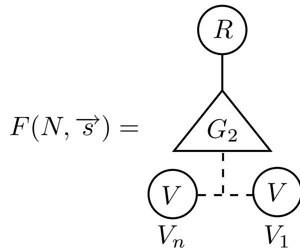
$$F(M, \overrightarrow{s}), F(N, \overrightarrow{s}) \text{ zijn grafen} \Rightarrow F(MN, \overrightarrow{s}) \text{ is een graaf}$$

Als we de graaf-manipulaties die nodig zijn om van $F(M, \overrightarrow{s})$ en $F(N, \overrightarrow{s})$, $F(MN, \overrightarrow{s})$ te maken, samenvatten in de applicatie-stap, $A : G \times G \rightarrow G$ dan kunnen we $F(MN, \overrightarrow{s})$ als volgt uitdrukken:

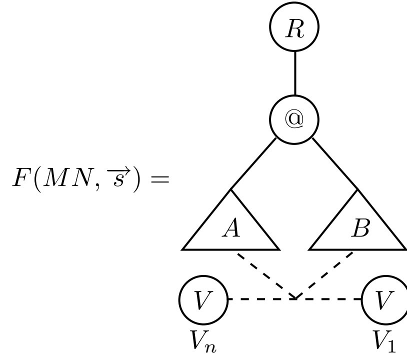
$$F(MN, \overrightarrow{s}) = A(F(M, \overrightarrow{s}), F(N, \overrightarrow{s}))$$

Om te weten wat $A : G \times G \rightarrow G$ inhoudt, bekijken we weer de resultaten van F(MN, $\overrightarrow{s}$), F(M, $\overrightarrow{s}$) en F(N, $\overrightarrow{s}$):

$F(M, \overrightarrow{s}) =$ 


$F(N, \overrightarrow{s}) =$ 

en




Het resultaat dat we willen hebben voor $F(MN, \overrightarrow{s})$ verdient nog enige uitleg. Dat we een applicatieknoop hebben met daaraan de twee subgrafen, ligt wel voor de hand. Dat we één rij $V$-knopen hebben, komt omdat we in de functie ook maar één stack hebben. En zoals gezegd is de rij $V$-knopen een representatie van die stack. De rij $V$-knopen wordt gevormd uit de beide rijen van beide subgrafen. We versmelten de $V$-knopen met dezelfde nummers. Dat betekent dat we van beide knopen één knoop maken waar de takken van beide knopen aan verbonden zijn.

De applicatiestap, $A : G \times G \rightarrow G$, ziet er als volgt uit:

- Er wordt een nieuwe applicatie-knoop gecreëerd.
- De applicatieknoop wordt verbonden aan de wortelwijzer met de bovenpoort.
- De wortel van de eerste subgraaf wordt aan de linkerpoort van de applicatieknoop verbonden.
- De wortel van de tweede subgraaf wordt aan de rechterpoort van de applicatieknoop verbonden.
- De $V$-knopen van beide subgrafen worden met elkaar versmolten.

### 3.3.5 De vertaalfunctie gedefinieerd

De vertaalfunctie maakt gebruik van een aantal functies. Deze functies hebben we in de vorige paragrafen informeel besproken. We vatten ze nog even samen:

**Definitie 34 (De end-of-scope-stap)** *De functie $E : G \rightarrow G$ voert de volgende manipulaties op een graaf uit:*

- *Er wordt een nieuwe $\lambda$-knoop gecreëerd.*
- *Deze knoop komt tussen de oude wortel en de wortelwijzer in met de bovenpoort aan de wortelwijzer en de onderpoort aan de oude wortel. Deze knoop wordt de nieuwe wortel van de graag.*

- *Er wordt een nieuwe V-knoop gecreëerd, met een nieuw nummer. We nemen hiervoor het nummer van de hoogst genummerde V-knoop plus 1. Deze knoop is niet aan de graaf verbonden.*

**Definitie 35 (De lambda-stap)** *De functie $\Lambda : G \to G$ voert de volgende manipulaties op een graaf uit:*

- *Er wordt een nieuwe $\lambda$-knoop gecreëerd.*
- *Deze knoop wordt tussen de wortel en de wortelwijzer geplaatst met de bovenpoort aan de wortelwijzer en met de body-poort aan de oude wortel.*
- *Alle takken die aan de $V$-knoop met het hoogste nummer zitten, worden aan de variabelenpoort van de nieuwe $\lambda$-knoop verbonden.*
- *De $V$-knoop met het hoogste nummer verdwijnt.*

**Definitie 36 (De applicatie-stap)** *De functie $A : G \times G \to G$, voert de volgende manipulaties uit op twee grafen:*

- *Er wordt een nieuwe applicatie-knoop gecreëerd.*
- *De applicatieknoop wordt verbonden aan de wortelwijzer met de bovenpoort.*
- *De wortel van de eerste subgraaf wordt aan de linkerpoort van de applicatieknoop verbonden.*
- *De wortel van de tweede subgraaf wordt aan de rechterpoort van de applicatieknoop verbonden.*
- *De $V$-knopen van beide subgrafen worden met elkaar versmolten.*

*Met versmelten bedoelen we hier dat twee $V$-knopen uit beide stacks met hetzelfde nummer, worden vervangen door één $V$-knoop met datzelfde nummer en dat de takken van beide $V$-knopen aan deze $V$-knoop verbonden worden.*

Nu kan de vertaalfunctie als volgt worden gedefinieerd:

**Definitie 37 (De vertaalfunctie)** *De vertaalfunctie $F : T \times S \to G$ (waarbij $T$ staat voor de verzameling termen, $S$ de verzameling stacks en $G$ de verzameling van grafen) is als volgt gedefinieerd:*

**Applicatie** $F(MN, \overrightarrow{s}) = A(F(M, \overrightarrow{s}), F(N, \overrightarrow{s}))$

**Abstractie** $F(\lambda x\ M, \overrightarrow{s}) = \Lambda(F(M, x\overrightarrow{s}))$

**Variabele niet in balans** $F(x, y\overrightarrow{s}) = E(F(x, \overrightarrow{s}))$, *waarbij* $x \neq y$

**Variabele in balans** $F(x, x\overrightarrow{s}) =$



*waarbij n gelijk is aan de lengte van $x\overrightarrow{s}$.*

We zullen hier nog een voorbeeld geven waarin de werking van de vertaalfunctie wordt gedemonstreerd. De term die we vertalen is: $\lambda x\ \lambda y\ (xx)y$. Dit is een variant op het lopend voorbeeld.

**Voorbeeld 15 (Het vertalen van een term)**

$$F(\lambda x\ \lambda y\ (xx)y, \square) = \Lambda(F(\lambda y\ (xx)y, x)) = \Lambda(\Lambda(F((xx)y, yx))) =$$

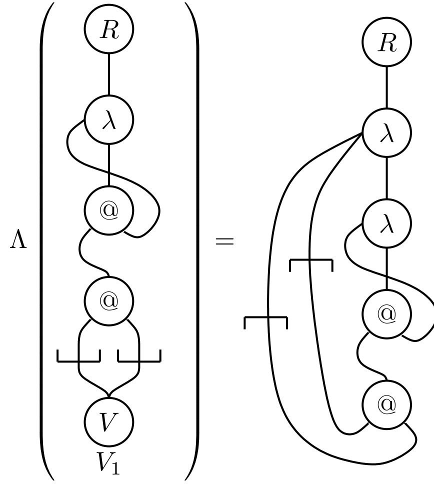
$$\Lambda(\Lambda(A(F(xx, yx), F(y, yx)))) = \Lambda(\Lambda(A(A(F(x, yx), F(x, yx)), F(y, yx)))) =$$

$$\Lambda(\Lambda(A(A(E(F(x, x)), E(F(x, x))), F(y, yx)))) =$$

$\Lambda(\Lambda(A(A(E$ ( R — V, $V_1$ ) $, E$ ( R — V, $V_1$ ) $), F(y, yx)))) =$

$\Lambda(\Lambda(A(A$ ( V $V_2$, R — V $V_1$ , V $V_2$, R — V $V_1$ ) $, F(y, yx)))) =$

$\Lambda(\Lambda(A$ ( R — @ — V $V_1$, V $V_2$ , R — V $V_2$, V $V_1$ ) $)) = \Lambda(\Lambda$ ( R — @ — @ — V $V_1$, V $V_2$ ) $) =$




## 3.4 De vertaling van twee $\alpha$-equivalente termen

We bewijzen in deze paragraaf dat als twee termen $\alpha$-equivalent zijn, ze dezelfde graafrepresentatie hebben.

**Stelling 1**
*Twee $\alpha$-equivalente termen hebben dezelfde graafrepresentatie:*

$$\overrightarrow{x} M =_\alpha \overrightarrow{y} N \Rightarrow F(M, \overrightarrow{x}) = F(N, \overrightarrow{y})$$

**Bewijs:**

*We maken gebruik van inductie over de opbouw van termen:*

**Variabelen - Basisgeval**

*Bij de variabelen zullen we onderscheid moeten maken tussen gebalanceerde en ongebalanceerde variabelen, waarbij we bij het ongebalanceerde geval inductie over de lengte van de stack nodig hebben:*

- *Variabelen, gebalanceerd:*

$$F(x, x\overrightarrow{x}) = \quad R - V_n \text{ - - - - - } V_1 \quad = F(y, y\overrightarrow{y})$$

*Merk hierbij op dat dit alleen waar is als $\overrightarrow{x}$ en $\overrightarrow{y}$ even lang zijn (en gelijk zijn aan $n-1$), maar een simpele inductie over de Kahrs-methode laat zien dat de scopestacks behorende bij twee $\alpha$-equivalente termen, altijd even lang zijn.*

- *Gebonden variabelen, ongebalanceerd:*

$$F(x, v\overrightarrow{x}) = E(F(x, \overrightarrow{x})) =_1 E(F(y, \overrightarrow{y})) = F(y, w\overrightarrow{y})$$

*Equivalentie 1 volgt uit Kahrs:*

$$v\overrightarrow{x}\ x =_\alpha w\overrightarrow{y}\ y \Rightarrow \overrightarrow{x}\ x =_\alpha \overrightarrow{y}\ y$$

*en de inductie hypothese.*

**Abstractie**

$$F(\lambda xM, \overrightarrow{x}) = \Lambda(F(M, x\overrightarrow{x})) =_1 \Lambda(F(N, y\overrightarrow{y})) = F(\lambda yN, \overrightarrow{y})$$

*Equivalentie 1 volgt uit de inductie hypothese en:*

$$\overrightarrow{x}\ \lambda xM =_\alpha \overrightarrow{y}\ \lambda yN \Rightarrow x\overrightarrow{x}\ M =_\alpha y\overrightarrow{y}\ N$$

**Applicatie**

$$F(M_1M_2, \overrightarrow{x}) = A(F(M_1, \overrightarrow{x}), F(M_2, \overrightarrow{x})) =_1$$

$$A(F(N_1, \overrightarrow{y}), F(N_2, \overrightarrow{y})) = F(N_1N_2, \overrightarrow{y})$$

*Equivalentie 1 volgt weer uit de inductie hypothese en:*

$$\overrightarrow{x}(M_1M_2) =_\alpha \overrightarrow{y}(N_1N_2) \Rightarrow \overrightarrow{x}M_i =_\alpha \overrightarrow{y}N_i$$

# 4 Van grafen naar termen

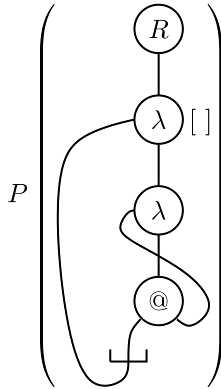
In dit hoofdstuk zullen we definiëren hoe we de grafische representatie van een term weer naar de term kunnen vertalen. Het terugvertalen gaat met context semantiek. We kiezen voor deze aanpak omdat die het makkelijkste is uit te breiden naar grafen waar ook de sharing-operator aan toe is gevoegd. We definiëren een deterministische, abstracte machine, die de grafen kan terugvertalen naar termen. Het resultaat van deze machine noemen we de read-back van de graaf en de machine noemen we de read-back-machine. De machine heeft een leeskop die bij de wortel van de graaf begint te lezen. Ook heeft de machine een stack waar informatie kan worden opgeslagen. Deze stack noemen we ook wel de context. De machine zal een aantal acties achter elkaar uitvoeren, die uiteindelijk een term als resultaat hebben. De graaf die de machine als input kreeg, is een grafische representatie van de term die de machine als output heeft. Als we een term eerst naar zijn grafische representatie vertalen en dan terugvertalen, dan moet het resultaat op z'n minst $\alpha$-equivalent zijn. Dat zullen we dan ook bewijzen in Stelling 2. De acties van de machine bestaan uit het manipuleren van de stack of het genereren van output, of allebei. Iedere actie van de machine wordt bepaald door de knoop die door de leeskop wordt gelezen en het element dat boven op de stack staat. Na iedere actie zal de leeskop een stap omlaag de graaf in gaan. De leeskop volgt dus een pad naar beneden door de graaf

(Definitie 28). Een pad naar beneden, splitst zich echter bij applicatieknopen. We zullen ervoor zorgen dat de machine zich ook 'splitst' zodat beide subgrafen worden vertaald.

In onze grafische weergave van de machine zullen we de inputgraaf tekenen en de stack. De positie van de leeskop wordt aangegeven door middel van de positie die de stack inneemt in de graaf. We laten in Paragraaf 4.1 aan de hand van een voorbeeld zien hoe het algoritme in zijn werk gaat. In Paragraaf 4.2 zullen we een definitie geven van de machine.

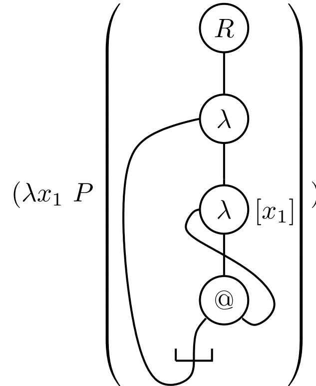
## 4.1 Read-back aan de hand van een voorbeeld

Als voorbeeld nemen we de grafische representatie van term T uit Hoofdstuk 3. We laten de machine beginnen bij de wortel met een lege stack. De wortel van de graaf wordt aangewezen door de wortelwijzer:

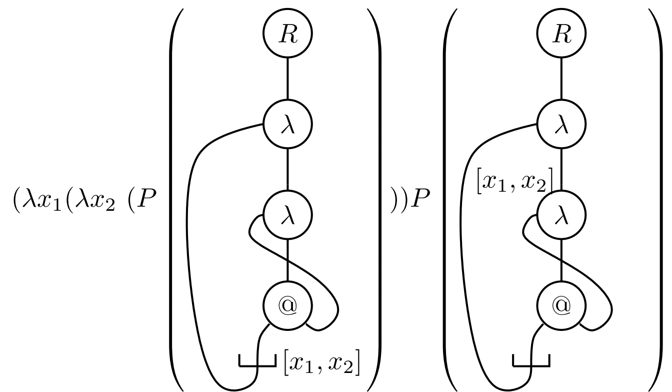



De leeskop leest als eerste een $\lambda$-knoop. Omdat we de graaf willen terug vertalen naar een term, zullen we de knoop moeten vertalen naar een $\lambda$-operator. De graaf die we vertalen is een naamloze representatie van een $\alpha$-equivalentieklasse van termen. We moeten nu vertalen naar één instantie van die klasse. Dit betekent dat we de variabelenaam die bij de $\lambda$-operator hoort, vrij kunnen kiezen, zolang de naam maar vers is. Met een verse variabelenaam bedoelen we in dit geval een naam die nog niet in de output of op de stack voor komt. In deze scriptie zullen we altijd een genummerde $x$ nemen, met als nummer een zo laag mogelijk natuurlijk getal. In dit geval dus $x_1$. Als we later een variabele vertalen die in de scope van deze $\lambda$-operator thuis hoort, moeten we weten welke variabelenaam we moeten gebruiken. We zullen de machine deze naam op de stack laten opslaan. Verder laten we de machine nog twee haakjes produceren. Eén voor de $\lambda$-operator en één na de output die nog door de machine geproduceerd zal worden. Het genereren van output (de $\lambda$-operator en haakjes) en deze stackmanipulatie (het opslaan van de variabelenaam op de stack) vormen samen de acties die de machine uitvoert als hij een $\lambda$-knoop leest. Na de actie gaat de leeskop, via de body-poort, een stap omlaag om de body

van de $\lambda$-knoop te vertalen. Deze stap ziet er dus als volgt uit:

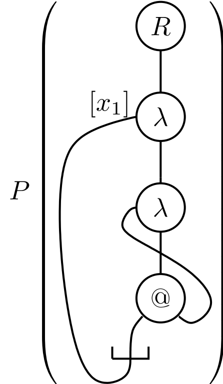



Vervolgens leest de leeskop weer een $\lambda$-knoop, dus de machine doet dezelfde actie nog een keer. Als variabelenaam kiezen we $x_2$. Na deze stap leest de leeskop een applicatie-knoop. In de term wordt de applicatie van twee subtermen uitgedrukt doordat de eerste voor de tweede staat. We moeten de machine dus de vertaling van de eerste subterm voor de tweede laten zetten. Om correct te blijven, genereren we twee haakjes die we om de hele subterm plaatsen. Als de machine een applicatieknoop leest, bestaat de actie van de machine dus uit het genereren van output. Er worden geen stackmanipulaties uitgevoerd. Na deze actie splitst de machine zich. We kopiëren de graaf. In de eerste versie van de graaf laten we de leeskop verder gaan via de linkerpoort en in de tweede graaf laten we de leeskop via de rechterpoort verdergaan. Het resultaat van de eerste graaf zetten we voor die van de tweede graaf. De stacks die we gebruiken zijn dezelfde. Dit alles samen geeft ons de volgende stap:

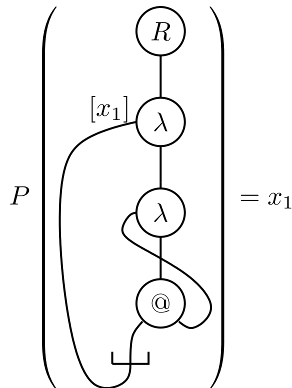



We richten ons nu eerst op de vertaling van de eerste subterm. In de eerste subterm komt de leeskop een $\reflectbox{$\lambda$}$-knoop tegen. We weten dat een $\reflectbox{$\lambda$}$-knoop de scope van de laatst geopende $\lambda$-operator afsluit. De machine moet dus het bovenste element van de stack verwijderen. We willen geen $\reflectbox{$\lambda$}$-operatoren in ons

resultaat hebben, dus op dit punt hoeft de machine geen output te generen:




Vervolgens komt de leeskop weer bij de $\lambda$-knoop maar nu vanaf de variabelenpoort. Dit is de grafische weergave van een variabele. De naam van de variabele die bij de binder van deze variabele hoort, staat nu bovenop de stack. Dat dit altijd het geval is, volgt uit het feit dat de graaf in balans is. Het algoritme werkt ook alleen maar als de grafen in balans zijn. Als output laten we de machine het bovenste element van de stack genereren. De machine stopt als de leeskop een variabelenpoort van een $\lambda$-knoop bereikt. Deze stap ziet er als volgt uit:




In de tweede subterm komt de leeskop direct bij een $\lambda$-knoop uit, via de variabelenpoort. Ook hier genereren we de naam die bovenop de stack staat ($x_2$). Het gehele resultaat wordt dan:

$$(\lambda x_1(\lambda x_2\ (x_1 x_2)))$$

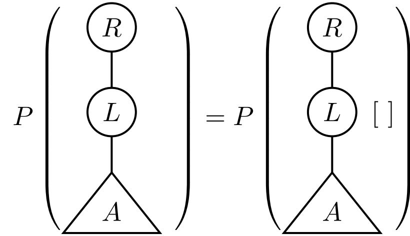
We zien dat dit resultaat $\alpha$-equivalent is met de term $\lambda x \lambda y\ xy$ waar we mee begonnen. Voor dit voorbeeld werkt onze machine. Laten we nu de algemene regels voor de machine opstellen. Daarna laten we zien dat de inputgraaf altijd een grafische representatie is van de output van de machine.

## 4.2 De machine in het algemeen

Per knoop die we kunnen tegenkomen in onze grafen zullen we nu één of meerdere algemene regels definiëren waarin staat beschreven welke output wordt gegenereerd en hoe de stack zal worden aangepast. We zullen zien dat we in de vorige paragraaf alle knopen die we (tot nu toe) kunnen tegenkomen al hebben behandeld en dat de algemene regels geen verrassing meer zullen opleveren:

**Definitie 38 (De vertaalmachine) Startstap**

*We laten de machine beginnen bij de knoop die door de wortelwijzer wordt aangewezen. De eerste stap bestaat dus uit het plaatsen van de leeskop bij de wortel van de inputgraaf. Dat ziet er als volgt uit:*

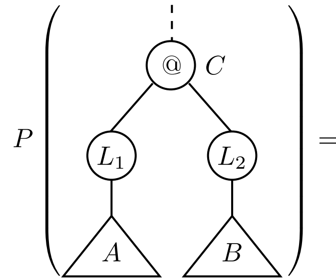


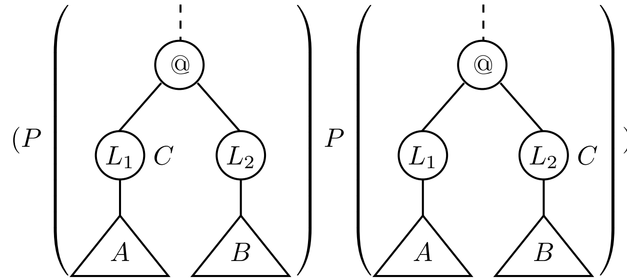

**Applicatieknoop**

*Als de leeskop een applicatieknoop leest dan worden de volgende acties uitgevoerd:*

- *Als output worden er twee haakjes gegenereerd, om de toekomstige output van de machine.*
- *Het algoritme splitst zich: De inputgraaf wordt gekopieerd. In de eerste versie gaat de leeskop verder via de linkerpoort van applicatieknoop. In de tweede versie gaat de leeskop via de rechterpoort verder. De stacks die we in beide versies gebruiken zijn gelijk.*
- *De output van de vertaling van de eerste versie van de inputgraaf wordt op de output van de tweede versie toegepast, oftewel de eerste wordt voor de tweede gezet en er worden haakjes omheen gezet.*

*Dit ziet er als volgt uit:*





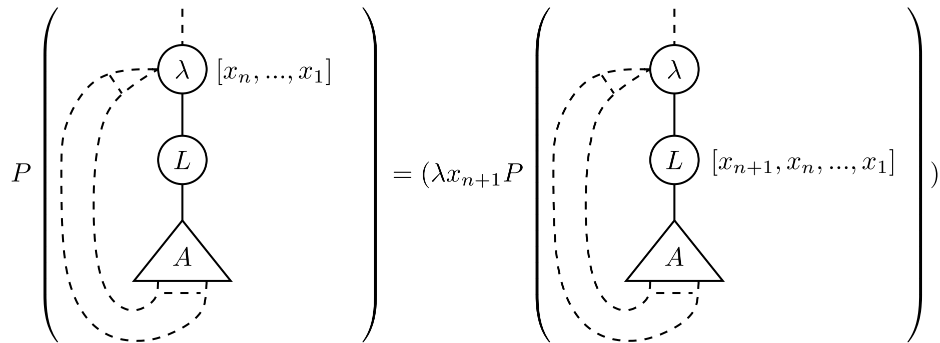

*Met $C$ een willekeurige stack, $L_1$ en $L_2$ willekeurige labels, $A$ en $B$ twee willekeurige grafen.*

**Abstractie**

*Als de leesknoop een $\lambda$-knoop bereikt via de bovenpoort, dan worden de volgende acties uitgevoerd:*

- *Er wordt een verse variabelenaam gegenereerd. Deze mag niet voorkomen op de stack of in de al gegenereerde output. De naam wordt bovenop de stack gezet.*
- *Er wordt een $\lambda$-operator als output gegenereerd, met als variabelenaam de zojuist gegenereerde variabelenaam.*
- *Als output worden er twee haakjes gegenereerd, om de $\lambda$-operator en de toekomstige output van de machine.*

*Dit ziet er als volgt uit:*



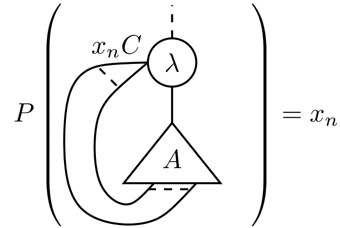

*Waarbij $x_{n+1}$ de verse variabelenaam is, $L$ een willekeurig label en $A$ een willekeurige graaf.*

**Variabele**

*Als de leeskop een $\lambda$-knoop bereikt via de variabelenpoort, dan worden de volgende acties uitgevoerd:*

- *Het bovenste element van de stack wordt als output gegenereerd.*
- *Het proces stopt, de leeskop gaat niet verder de graaf in.*

*Dit ziet er als volgt uit:*

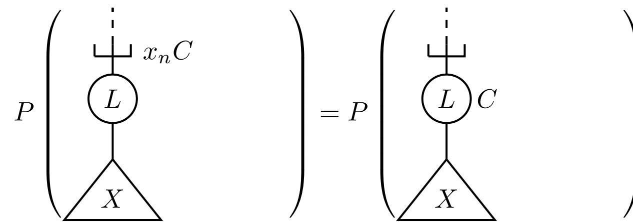



**End-of-scopeknoop**

*Als de leeskop een ʎ-knoop leest, wordt de volgende actie uitgevoerd:*

- *Het bovenste element wordt van de stack gehaald.*

*Dit ziet er als volgt uit:*




### 4.2.1 Welgevormde- en onwelgevormde grafen

Het vertaalalgoritme dat we nu hebben gegeven werkt alleen op welgevormde grafen (zie Definitie 33). In een welgevormde graaf worden alle variabelen gebonden door $\lambda$-knopen, er zijn immers geen $V$-knopen meer aanwezig. Een welgevormde graaf is dus een representatie van een gesloten term. Als een graaf onwelgevormd is, bevat deze wel $V$-knopen. Deze $V$-knopen zijn een representatie van een scopestack. Een onwelgevormde graaf moeten we vertalen naar een term met een scopestack ervoor.

Om te zorgen dat de machine ook onwelgevormde grafen kan vertalen, zullen we een regel moeten toevoegen voor de $V$-knoop. Als de leeskop bij een $V$-knoop komt, dan weten we dat we te maken hebben met een variabele en we moeten de machine dus hetzelfde laten doen als wanneer we een $\lambda$-knoop tegenkomen vanaf de variabelenpoort. Er volgt echter uit gebalanceerdheid van de graaf dat als we met een lege stack bij de wortel beginnen, de stack weer leeg zal zijn als we bij de $V$-knoop komen. Op de stack wordt informatie gezet steeds als de read-back-machine een $\lambda$-knoop tegenkomt. Uit de definitie van vertaalfunctie $F$ (Definitie 37) volgt dat de output van deze functie in balans is. Daaruit volgt dat als een variabele wordt vertaald alle scopes gesloten zijn die later werden geopend dan de $\lambda$-operator die de variabele bindt. Omdat de variabelen die aan een $V$-knoop bevestigd zijn, niet door een $\lambda$-knoop gebonden zijn, moeten alle scopes afgesloten zijn. Steeds als er een scope wordt afgesloten, wordt de

informatie die erbij hoort van de stack gehaald. Als we een $V$-knoop bereiken, is de stack dus leeg. De machine weet dan niet welke variabelenaam als output moet worden gegenereerd. Deze informatie wordt in het variabele geval van de stack gehaald. We moeten het algoritme dus al beginnen met informatie op de stack en wel zoveel informatie als dat er scopeknopen zijn. Dus een stack met daarop $x_1$ tot en met $x_n$, waarbij $n$ de lengte is van de rij $V$-knopen. Deze stack met informatie moet bovendien direct als output worden gegenereerd, zodat deze als scopestack voor de term komt te staan. Naast het toevoegen van een $V$-knoop regel zullen we dus ook de startstap moeten aanpassen.

**Definitie 39 (De vertaalmachine aangepast)**

**Startstap**

$P$ ( $R$, $A$, $V$ $V$, $V_n$ $V_1$ ) $= [x_n, ..., x_1]P$ ( $R$, $[x_n, ..., x_1]$, $A$, $V$ $V$, $V_n$ $V_1$ )

**Scopeknoop**

$P$ ( $V$ $[x_m, ..., x_1]$ ) $= x_m$

Merk op dat de startstap voor welgevormde grafen een speciaal geval is van de startstap die hier staat. In dat geval is de stack bij de leeskop leeg en er wordt geen output gegenereerd (eigenlijk wordt er een lege stack als output gegenereerd, maar die kunnen we net zo goed weg laten).

## 4.3 Heen en terug vertalen geeft $\alpha$-equivalentie

We hebben gedefinieerd hoe we een term kunnen vertalen naar een graaf en we hebben gedefinieerd hoe een graaf kan worden terugvertaald naar een term. Nu moeten we bewijzen dat als we een term vertalen naar een graaf en weer terugvertalen, dat het resultaat $\alpha$-equivalent is met de term waar we mee begonnen. Om dit te bewijzen zullen we het volgende lemma nodig hebben:

**Lemma 1**
*Als we de read-back-machine op een subgraaf toepassen, dan geeft dit hetzelfde resultaat als wanneer de subgraaf een graaf zou zijn. Dus:*

$P$ ( R — C — A ) $= P$ ( R — B — C — A )

***Bewijs met inductie naar opbouw van de omgeving:***
*De omgeving moet zijn opgebouwd met behulp van de graafconstructie regels $E : G \to G$, $\Lambda : G \to G$ en $A : G \times G \to G$ (zie paragraaf 3.3.5).*

**Basis:**

*Als de omgeving bestaat uit één laag, dan onderscheiden we de volgende drie gevallen:*

$E : G \to G$

*De vraag is of:*

$P$ ( R — C — A; $V_n$ … $V_1$ ) $= P(E$ ( R — C — A; $V_n$ … $V_1$ ) $) =$

$P$ ( R — C — A; $V_{n+1}$, $V_n$ … $V_1$ )

*Om te bewijzen dat het resultaat van de read-back-machine niet anders is, moeten we de nieuwe elementen in de graaf bekijken en laten zien dat ze geen invloed op de read-back hebben.*

***Nieuwe elementen:***

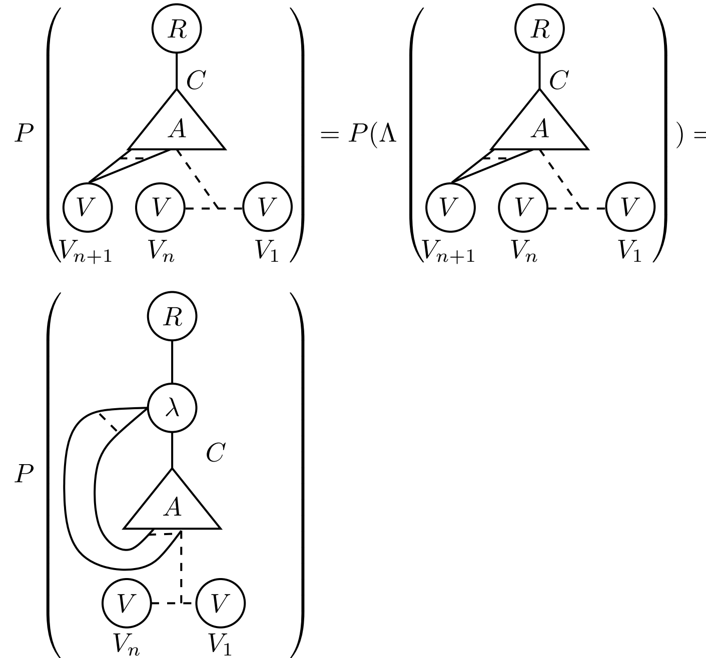
**De ʎ-knoop** *De ʎ-knoop heeft geen invloed op de read-back. De leeskop is al langs deze knoop en omdat we weten dat het read-back-algoritme een pad door de graaf naar beneden volgt, weten we ook dat de ʎ-knoop niet nog eens wordt gelezen.*

**De $V$-knoop** *De $V$-knoop is volgens de definitie van $E(..)$ niet verbonden met subgraaf $A$ (zie Definitie 34). Knopen die niet bereikt kunnen worden via een tak zullen niet door de leeskop van de read-back-machine gelezen worden. Dat betekent dat de $V$-knoop geen invloed heeft op de read-back.*

*De enige verschillen tussen beide grafen zijn de bovengenoemde knopen en deze knopen hebben geen invloed op de read-back.*

$\Rightarrow$ *Een omgeving van één laag, opgebouwd met $E : G \to G$ heeft geen invloed op de vertaling van de subterm.*

$\Lambda : G \to G$ *De vraag is of:*




***Nieuwe elementen:***

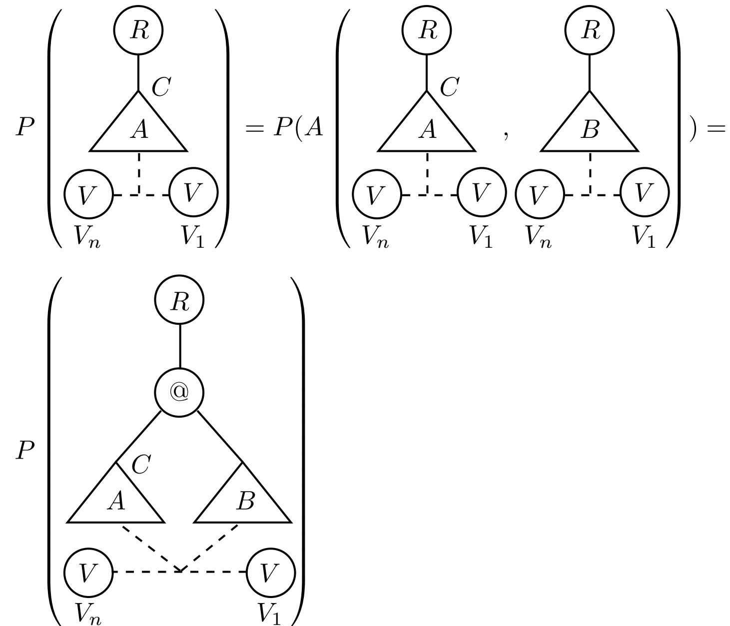
**Een $V$-knoop is verdwenen** *Als de $V$-knoop niet verbonden was aan graaf $A$ dan heeft de verdwijning van die knoop ook geen invloed op de read-back. Als de $V$-knoop wel een tak had aan $A$ dan zal de leeskop nu in plaats van de $V$-knoop een $\lambda$-knoop via de variabelenpoort bereiken. De resultaten voor beide gevallen zijn echter, volgens de definitie van het read-back-algoritme hetzelfde. De verdwijning van de $V$-knoop heeft dus geen invloed op de read-back.*

**De $\lambda$-knoop** *De $\lambda$-knoop heeft ook geen invloed op de read-back. De leeskop is er al voorbij en als de leeskop terugkeert bij deze knoop, dan hebben we te maken met het geval van de verdwenen $V$-knoop en is er ook geen verschil in de read-back.*

*De enige verschillen tussen beide grafen zijn de bovengenoemde knopen en deze knopen hebben geen invloed op de read-back.*

$\Rightarrow$ *Een omgeving van één laag, opgebouwd met $\Lambda : G \to G$ heeft geen invloed op de vertaling van de subterm.*

$A : G \times G \to G$ *De vraag is of:*




***Nieuwe elementen:***

**De @-knoop** *De @-knoop heeft geen invloed op de read-back. De leeskop is al langs deze knoop en omdat het read-back-algoritme een pad naar beneden volgt, komt de leeskop niet nog eens bij de A-knoop.*

**Subgraaf** $B$ *Grafen $A$ en $B$ hebben onderling geen takken. De leeskop kan dus niet van $A$ naar $B$ komen. Beide subgrafen kunnen takken hebben naar dezelfde $V$-knoop. Als de leeskop bij een $V$-knoop komt waar ook een tak uit $B$ aan verbonden is, dan stopt het algoritme en wordt er niets gedaan met de andere tak. De aanwezigheid van $B$ heeft dus geen invloed op de read-back van $A$.*

*De enige verschillen tussen beide grafen zijn de bovengenoemde knoop en bovengenoemde graaf en deze hebben allebei geen invloed op de read-back.*

*Als de graaf aan de rechterpoort van een applicatieknoop verbonden is, is het bewijs analoog aan het bewijs van een graaf die aan de linkerpoort van een applicatieknoop verbonden is. Dit geval wordt hier verder achterwege gelaten.*

$\Rightarrow$ *Een omgeving van één laag, opgebouwd met* $A : G \to G$ *heeft geen invloed op de vertaling van de subterm.*

*Een omgeving van één laag kan alleen worden opgebouwd met de functies* $E : G \to G$, $\Lambda : G \to G$ *en* $A : G \times G \to G$. *Geen van deze functies heeft invloed op de read-back van de subterm.*

$\Rightarrow$ *Een omgeving van één laag heeft geen invloed op de vertaling van de subterm.*

**Inductie**

*De inductie hypothese is als volgt:*

P ( R – C – A ) = P ( R – B – C – A )

*Waarbij* $B$ *is opgebouwd met* $n$ *maal het toepassen van* $A : G \times G \to G$, $E : G \to G$ *of* $\Lambda : G \to G$.

***Geval*** $n+1$***:*** *Als* $B$ *is opgebouwd door* $n+1$ *keer een graafconstructie regel toe te passen, dan weten we dat het bewijs geldt voor* $B'$, *die met* $n$ *graafconstructie regels is opgebouwd. Om van* $B'$ *naar* $B$ *te komen moeten we één maal een graafconstructie regel toepassen. Het laten zien dat dit geen invloed op de read-back heeft, is zo weinig verschillend van het basisgeval dat ik dat hier achterwege laat.*

**Stelling 2**

*Heen en terugvertalen geeft een* $\alpha$*-equivalente term als resultaat:*

$$P(F(M, \overrightarrow{s})) =_\alpha \overrightarrow{s}\ M$$

*Met* $M$ *een willekeurige term en* $\overrightarrow{s}$ *de eventueel lege scopestack die daarbij hoort.*

**Bewijs:**

*We zullen bij dit bewijs gebruik maken van inductie over de opbouw van termen en van* $\alpha$*-equivalentie, zoals gedefinieerd door Kahrs (Definitie 15). In dit bewijs gebruik ik* $\overrightarrow{x}$ *en* $\overrightarrow{y}$ *als afkortingen voor respectievelijk* $[x_n, ...., x_1]$ *en* $[y_n, ...., y_1]$.

**Variabelen** *In het basisgeval, het vertalen van een variabele, moeten we onderscheid maken tussen het gebalanceerde en het ongebalanceerde geval. Een variabele kan alleen worden vertaald in combinatie met een scopestack. Als de variabelenaam boven op de stack staat is het een gebalanceerd geval. Als de variabelenaam niet boven op de stack staat is het een ongebalanceerd geval.*

- *De variabele, gebalanceerd:*

$$P(F(y, y\overrightarrow{y})) = P\left( \text{[diagram: } R \text{ — } V_n \text{ - - - } V_1 \text{]} \right) =$$

$$[x_n, ..., x_1]P\left( \text{[diagram: } R \text{ — } V_n \text{ - - - } V_1 \text{, } [x_n, ..., x_1] \text{]} \right) = [x_n, ..., x_1]\ x_n =_\alpha y\overrightarrow{y}\ y$$

- *De variabele, ongebalanceerd:*
  *Met* $z \neq y$

$$P(F(y, z\overrightarrow{y})) = P(E(F(y, \overrightarrow{y}))) = P(E\left( \text{[diagram: } R \text{ — } X \text{; } V_n \text{ - - - } V_1 \text{]} \right)) =$$

$$P\left( \text{[diagram: } R \text{ — } X \text{; } V_{n+1}, V_n \text{ - - - } V_1 \text{]} \right) = [x_{n+1}, x_n, ..., x_1]P\left( \text{[diagram: } R \text{ — } X \text{, } [x_{n+1}, x_n, ..., x_1] \text{; } V_{n+1}, V_n \text{ - - - } V_1 \text{]} \right) =$$

$$x_{n+1}\overrightarrow{x}\ P\left(\begin{array}{c} R \\ [x_n,...,x_1] \\ X \\ V\ V\ V \\ V_{n+1}\ V_n\ V_1 \end{array}\right) = x_{n+1}\overrightarrow{x}\ x_i$$

*We weten dat graaf $X$ bestaat uit een rij end-of-scopeknopen en dan een $V$-knoop. Dus dit vertaalt uiteindelijk naar een variabelenaam die op de stack $\overrightarrow{x}$ voorkomt. We moeten dus bewijzen:*

$$x_{n+1}\overrightarrow{x}\ x_i =_\alpha z\overrightarrow{y}\ y$$

*Dit is waar, volgens de Kahrs-methode als:*

$$\overrightarrow{x}\ x_i =_\alpha \overrightarrow{y}\ y$$

*De inductiehypothese is als volgt:*

$$P(F(y,\overrightarrow{y})) =_\alpha \overrightarrow{y}\ y$$

*En:*

$$P(F(y,\overrightarrow{y})) = P\left(\begin{array}{c} R \\ X \\ V\ V \\ V_n\ V_1 \end{array}\right) =$$

$$[x_n,...,x_1]P\left(\begin{array}{c} R \\ [x_n,...,x_1] \\ X \\ V\ V \\ V_n\ V_1 \end{array}\right) =_{Lemma\ 1} \overrightarrow{x}\ P\left(\begin{array}{c} R \\ [x_n,...,x_1] \\ X \\ V\ V\ V \\ V_{n+1}\ V_n\ V_1 \end{array}\right) =$$

$$\overrightarrow{x}\ x_i =_\alpha \overrightarrow{y}\ y$$

**abstractie**

$P(F(\lambda y M, \overrightarrow{y})) = P(\Lambda(F(M, y\overrightarrow{y}))) = P(\Lambda$

R
A
V V V
$V_{n+1}$ $V_n$ $V_1$

$) =$

$P$

R
$\lambda$
A
V V
$V_n$ $V_1$

$= \overrightarrow{x} P$

R
$[\overrightarrow{x}]$
$\lambda$
A
V V
$V_n$ $V_1$

$=$

$\overrightarrow{x}\ \lambda x_{n+1} P$

R
$\lambda$
$[x_{n+1}, \overrightarrow{x}]$
A
V V
$V_n$ $V_1$

$=_{Lemma\ 1}$

$\overrightarrow{x}\ \lambda x_{n+1} P$

R
$[x_{n+1}, \overrightarrow{x}]$
A
V V V
$V_{n+1}$ $V_n$ $V_1$

$= \overrightarrow{x}\ \lambda x_{n+1} N$

*Uit de inductiehypothese volgt nu:*

$$P(F(M, y\overrightarrow{y})) = x_{n+1}\overrightarrow{x}\ N =_\alpha y\overrightarrow{y}\ M$$

*en uit de definitie van $\alpha$-equivalentie volgt dan:*

$$x_{n+1}\overrightarrow{x}\ N =_\alpha y\overrightarrow{y}\ M \Rightarrow \overrightarrow{x}\ \lambda x_{n+1}N =_\alpha \overrightarrow{y}\ \lambda yM$$

**applicatie**

$$P(F(M_1M_2, \overrightarrow{y})) = P(A(F(M_1, \overrightarrow{y}), F(M_2, \overrightarrow{y}))) =$$

$P(A$ ( … , … ) $) =$

$P$ ( … ) $= [x_n, ..., x_1]P$ ( … ) $=$

$[x_n, ..., x_1](P$ ( … ) $P$ ( … ) $) =_{2\times Lemma\ 1}$

$$\overrightarrow{x}(P\left(\begin{array}{c} R \\ | \, C \\ A \\ V_n \quad V_1 \end{array}\right) P\left(\begin{array}{c} R \\ | \, C \\ B \\ V_n \quad V_1 \end{array}\right)) =$$

$$\overrightarrow{x}(N_1 N_2)$$

*Uit de inductie hypothese volgt:*

$$P(F(M_1, \overrightarrow{y})) = \overrightarrow{x}\ N_1 =_\alpha \overrightarrow{y}\ M_1$$

*en:*

$$P(F(M_2, \overrightarrow{y})) = \overrightarrow{x}\ N_2 =_\alpha \overrightarrow{y}\ M_2$$

*Uit de definitie van $\alpha$-equivalentie volgt nu:*

$$\overrightarrow{x}\ N_1 N_2 =_\alpha \overrightarrow{y}\ M_1 M_2$$

# 5 De herschrijfregels

In de vorige twee hoofdstukken hebben we gezien hoe we termen naar grafen kunnen vertalen en hoe we ze weer terug kunnen vertalen. De termen kunnen we herschrijven door middel van $\beta$-reductie. Om een correcte semantische representatie te krijgen, moeten we ook herschrijfregels voor grafen definiëren, zodanig dat ze $\beta$-reductie representeren. Dus:

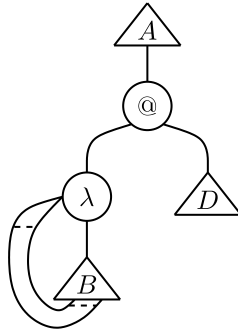
$$\begin{array}{ccc} t_1 & \xrightarrow{\beta} & t_2 \\ \downarrow & & \downarrow \\ g_1 & \xrightarrow[\text{Herschrijfregel}]{} & g_2 \end{array}$$

## 5.1 *Beta*-reductie

Bij het toepassen van een $\beta$-regel, wordt een $\beta$-redex herschreven. Om een correcte representatie van de regel te vinden zullen we dus eerst moeten kijken hoe de representatie van de $\beta$-redex er uitziet. Een $\beta$-redex ziet er als volgt uit:

$$C[(\lambda x\ M)N]$$

Met $C[...]$ een willekeurige context, $x$ een willekeurige variabele en $M$ en $N$ willekeurige termen. Als we deze vertalen krijgen we de volgende graaf:



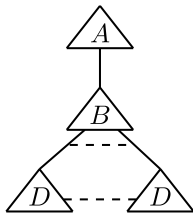

Met $A$ de vertaling van $C$, $B$ de vertaling van $M$ en $D$ de vertaling van $N$. We zien dat een redex in de graaf wordt gerepresenteerd door een applicatie-knoop die via zijn linkerpoort aan een $\lambda$-knoop verbonden is. Deze combinatie van knopen zullen we vanaf nu een *Beta*-redex noemen.

**Definitie 40 (*Beta*-redex)** *Een Beta-redex is een grafische representatie van een $\beta$-redex en bestaat uit een @-knoop die via zijn linkerpoort aan een $\lambda$-knoop verbonden is.*

We moeten een (lokale) herschrijfregel definiëren die deze combinatie van knopen als linkerkant heeft. Om te zien wat er aan de rechterkant moet komen zullen we eerst bekijken hoe een willekeurige redex in een term eruitziet, nadat deze gereduceerd is:

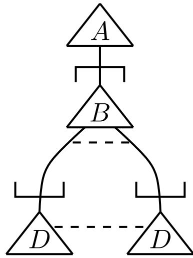
$$C[(\lambda\ xM)N] \ \rightarrow_\beta \ C[M[x := N]]$$

Als we dit naar de graaf representatie zouden vertalen, krijgen we het volgende:



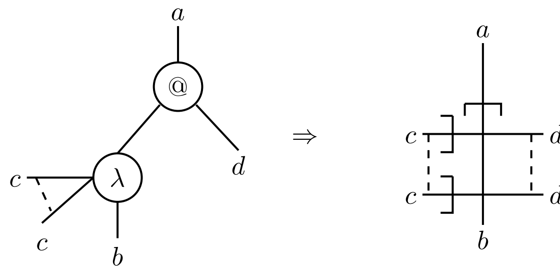

Deze vertaling is echter niet correct, want het zou zo kunnen zijn dat de graaf niet meer in balans is. Er kunnen ʎ-knopen voorkomen in $B$ die de scope van de verdwenen $\lambda$-knoop afsluiten. Na het toepassen van de herschrijfregel sluiten deze dus een scope af die er niet meer is. We moeten zorgen dat de graaf in balans blijft. Dit kan op twee manieren. We kunnen alle ʎ-knopen die bij de $\lambda$-knoop horen verwijderen. Dit is echter niet lokaal, want we moeten de hele subgraaf afzoeken naar die knopen. Daarnaast zouden we een probleem krijgen met het updaten van indices (die ontstaan bij andere herschrijfregels, zie bijvoorbeeld Paragraaf 5.2). Een andere mogelijkheid is om een knoop te

plaatsen op de plaats van de $\lambda$-knoop, die een scope opent. Deze scope is een lege scope omdat alle variabelen die tot deze scope behoren al zijn gesubstitueerd. We zullen deze scope nog moeten afsluiten voordat we graaf $D$ op de plek van de variabele plaatsen. De vertaling gaat er dan als volgt uitzien:




De herschrijfregel is als volgt:

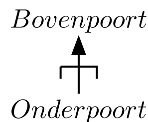
**Definitie 41 (*Beta*-stap)**




*Merk op dat de subgraaf die aan d gekoppeld is, na de stap gedupliceerd is.*

De nieuwe knoopsoort die in de graaf ontstaat noemen we de open-scopeknoop:

**Definitie 42 (De open-scopeknoop)** *De open-scopeknoop is te herkennen doordat we deze de vorm van een vierkante haak geven. De open-scopeknoop heeft een bovenpoort en een onderpoort die we ook wel de body-poort noemen. De bovenpoort is de interactiepoort. De knoop ziet er als volgt uit:*

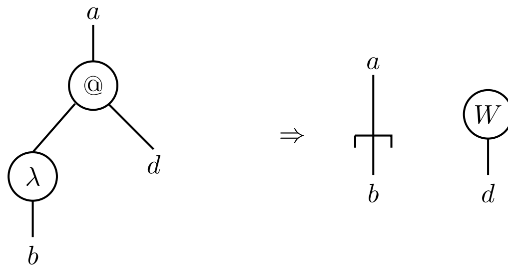



Eigenlijk zijn de open-scopeknoop en de ⅄-knoop dezelfde knoop. Ze hebben alleen een andere oriëntatie in de graaf.

### 5.1.1 Geen gebonden variabelen

Het kan voorkomen dat de $\lambda$-knoop die in de $\beta$-stap betrokken is, geen variabelen bindt. In de $\lambda$-calculus wordt in dat geval het argument weggegooid. Dat moet

ook in de graafrepresentatie gebeuren. We kunnen dit in onze graafrepresentatie doen door het argument los te koppelen van de hoofdgraaf. De read-back-machine zal dan niet in het argument komen en deze niet vertalen. We definiëren voor dit geval de volgende regel:




We introduceren hier een nieuwe knoop, de $W$-knoop (weggooi-knoop). Dit doen we vooral omdat we anders te maken zouden krijgen met een tak vanuit $d$ die slechts aan één knoop bevestigd is. De $W$-knoop heeft verder geen functie, behalve dat het aangeeft dat de subgraaf die er aan hangt is weggegooid. In een implementatie zullen we normaal gesproken de graaf ook echt moeten weggooien, omdat we ruimte moeten vrij maken. Dit zou kunnen door wel herschrijfregels te maken voor de $W$-knoop. Zodoende zou de $W$-knoop de graaf knoop voor knoop kunnen weggooien. In deze scriptie laten we deze *garbage collection* verder echter achterwege.

### 5.1.2 De read-back machine uitgebreid

In deze herschrijfregels ontstaat een nieuwe knoop, de open-scopeknoop. Als er nieuwe knopen voorkomen in de graaf, dan zullen we de regels voor de read-back machine ook moeten uitbreiden, zodat deze de nieuwe knopen ook kan vertalen. De betekenis van de open-scopeknoop is dat er een scope wordt geopend. Er zullen zich echter nooit variabelen bevinden in de scope van een open-scopeknoop. Deze zijn immers allemaal gesubstitueerd tijdens de *Beta*-stap. De enige reden dat deze scope wordt geopend is dat de graaf in balans blijft en dat de $\curlywedge$-knopen in de graaf de juiste scope blijven afsluiten. In het read-back-algoritme moet er een element op de stack worden gezet, zodat de $\curlywedge$-knopen dit element weer kunnen verwijderen. Omdat er echter geen variabelen in de scope gebonden worden, maakt het niet uit wat dat element is. We nemen voor alle open-scopeknopen hetzelfde element, namelijk $*$. De read-back regel voor open-scopeknopen ziet er als volgt uit:

**Definitie 43 (Vertaalmachine uitgebreid voor open-scopeknopen)**

**De open-scopeknoop** *Als de leeskop een open-scopeknoop leest, worden de volgende acties uitgevoerd:*

- *Het element $*$ wordt op de stack gezet.*

- *De leeskop verlaat de knoop via de onderpoort en leest de knoop aan de andere kant van de tak.*
- *Er wordt geen output gegenereerd.*

*Dit ziet er als volgt uit:*

$$P\left( \begin{array}{c} [x_n, ..., x_1] \\ L \\ A \end{array} \right) = P\left( \begin{array}{c} L\ [*, x_n, ..., x_1] \\ A \end{array} \right)$$

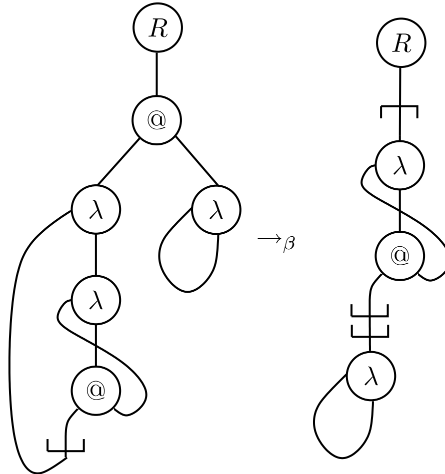
## 5.2 Het pushen van een end-of-scopeknoop

Zoals al eerder gezegd zijn onze grafen een verfijning van λ-termen. Ze bevatten meer informatie dan de λ-termen, namelijk informatie over het einde van de scope van de λ-operatoren. Deze informatie zorgt echter ook voor problemen, net als in de ʎ-calculus. Een voorbeeld:

**Voorbeeld 16 (Het ontstaan van verborgen redexen)**

$$(\lambda xy.xy)\lambda z\ z\ \rightarrow_\beta\ \lambda y\ (\lambda z\ z)y\ \rightarrow_\beta\ \lambda y\ y$$

*In de grafische representatie ziet dit er als volgt uit:*

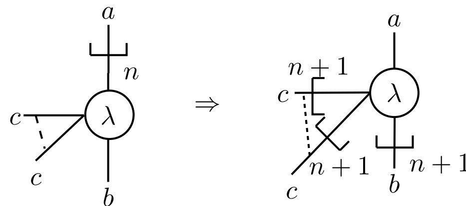



We zien dat na de eerste *Beta*-stap twee ʎ-knopen tussen de applicatie- en de λ-knoop van de tweede *Beta*-redex zijn terecht gekomen. We hebben hier te maken met een verborgen redex.

**Definitie 44 (Verborgen redexen voor grafen)** *Als er aan de linkerpoort van een applicatie-knoop een willekeurig aantal scope-informatie-knopen, gevolgd door een λ-knoop bevestigd zijn, dan spreken we van een verborgen Beta-redex.*

Om dit probleem op te lossen zullen we een tweede herschrijfregel definiëren waarin de ⅄-knoop over de λ-knoop heen wordt geduwd, zodat hij niet meer in de weg zit:

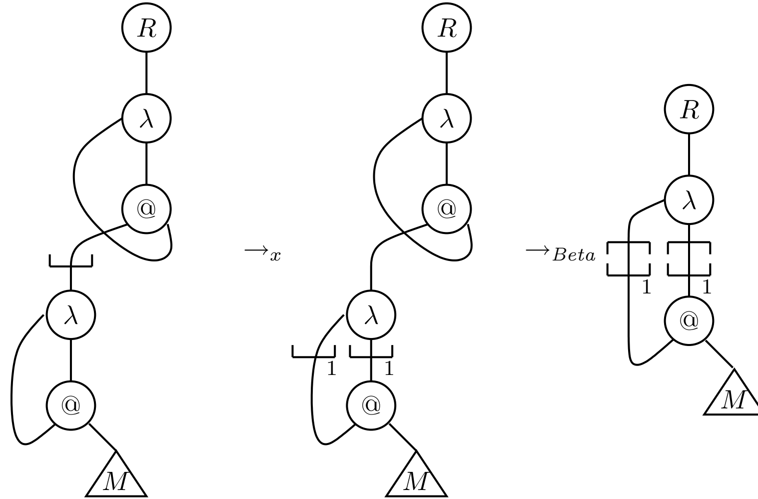
**Definitie 45 (De ⅄-push)**




De knopen die we in deze regel gebruiken zijn voorzien van een index. De reden hiervoor is als volgt:
De betekenis van een gewone ⅄-knoop is dat hij de laatst geopende scope afsluit. Als we de ⅄-push-regel toepassen, dan wordt de ⅄-knoop een nieuwe scope ingeduwd. Dit betekent dat de ⅄-knoop niet de laatst geopende scope afsluit (want dat is degene die wordt geopend door de λ-knoop waar hij net overheen geduwd is), maar de op één na laatste scope. Met de index geven we aan over hoeveel λ-knopen de ⅄-knoop is gepushed, dus hoeveel scopes er later zijn geopend dan de scope die de ⅄-knoop moet afsluiten. De conventie is dat als de index 0 is, we hem niet noteren. Dus een ⅄-knoop die niet over een λ-knoop heen gepushed is, heeft geen index. De betekenis van een geïndiceerde ⅄-knoop is dat de scope tijdelijk wordt afgesloten. De geïndiceerde open-scopeknoop, die we re-open-scopeknoop zullen noemen, kan deze scope weer heropenen. We geven een voorbeeld om dit verder toe te lichten:

**Voorbeeld 17 (De werking van de ⅄-push)** *We zullen de representatie van*

*$\lambda x($ʎ$x \lambda y\ yM)x$ herschrijven:*




*We zien dat de ʎ-knoop uit de weg wordt gehaald en dat de redex wordt uitgevoerd. De ʎ-knoop is nu gesplitst in een geïndiceerde ʎ-knoop en een re-open-scopeknoop. De geïndiceerde ʎ-knoop sluit nog steeds de scope van de laagste $\lambda$-knoop boven hem af en zorgt er zo voor dat subgraaf $M$ buiten de scope van deze knoop blijft. De re-open-scopeknoop opent de scope van de $\lambda$-operator weer zodat de representatie van variabele $x$ weer binnen de scope valt.*

### 5.2.1 De read-back aangepast

Door de herschrijfregels ontstaan er een paar nieuwe knopen:

- De geïndiceerde ʎ-knoop of temporarily-close-scopeknoop
- De geïndiceerde open-scopeknoop, of re-open-scopeknoop.

We zullen nu regels definiëren waarmee de read-back machine deze knopen kan vertalen.

De betekenis van een ʎ-knoop met index $n$ is dat de op $n$ na laatste scope tijdelijk wordt gesloten. Dit betekent dat als de vertaalmachine een ʎ-knoop met index $n$ leest, er op de stack van bovenaf geteld $n$ elementen moeten worden overgeslagen en het element daaronder moet worden verwijderd. Dit element kan echter niet worden weggegooid, want de scope moet slechts tijdelijk worden gesloten. Als de leeskop een re-open-scopeknoop tegenkomt met index $n$, moet de scope weer worden heropend en daar is de informatie die dit element bevat bij nodig. We moeten deze informatie ergens opslaan waar het tijdelijk geen invloed heeft op het read-back algoritme van de graaf. De structuur van de context, de stack, zoals we die nu hebben, is hiervoor niet toereikend. We zullen de context dan ook aanpassen:

**Definitie 46 (De Context)** *Een context wordt met de volgende grammatica opgebouwd:*

**Context** $C = \Box \mid lC$

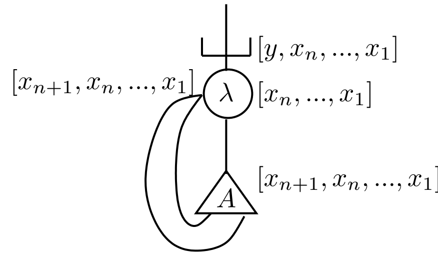
**Niveau** $l = i \mid (l_1, l_2)$

**Informatie-eenheid** $i = v \mid *$

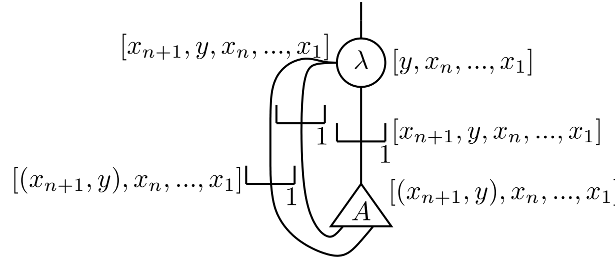
*Waarbij $C$ een context is, $l_1$ en $l_2$ niveaus (*levels*) zijn, $v$ een variabelenaam is en $i$ een informatie-eenheid is.*

Het volgende voorbeeld laat zien hoe de geïndiceerde scopeknopen hun werk doen, ten opzichte van de gewone $\lambda$-knoop. Het element dat we tijdelijk moeten weggooien, wordt nu even opzij gezet. Het bewuste element en het element erboven worden samen tot een paar gevormd met het bovenste element voorop:

- De werking van de normale $\lambda$-knoop is als volgt:




- De werking van de geïndiceerde $\lambda$-knoop is als volgt:




**Definitie 47 (Primaire informatie)**
*De functie $\Pi : C \to C$ waarbij $C$ is een context, is inductief als volgt gedefinieerd, naar opbouw van context:*

$$\Pi(i) = i$$

$$\Pi((l_1, l_2)) = \Pi(l_1)$$

$$\Pi(lC) = \Pi(l) : \Pi(C)$$

*Het resultaat van deze functie is een context waar alleen de primaire informatie op staat van de context waar de functie op werd toegepast.*

**Definitie 48 (Secundaire informatie)** *Alle informatie op een context die geen primaire informatie is, is secundaire informatie.*

Uit de grammatica van de context volgt dat ieder level van een context precies één element als primaire informatie heeft. De primaire informatie van een context is het gedeelte van de context dat invloed heeft op de read-back van een graaf. De read-back regels voor de geïndiceerde ʎ-knoop en re-open-scopeknoop veranderen primaire informatie naar secundaire informatie en andersom. De regels voor het terugvertalen van een variabele moeten iets worden aangepast. De read-back-machine neemt niet meer het bovenste element van de context, maar de primaire informatie van het bovenste level en produceert dat als output. De algemene regel voor geïndiceerde ʎ-knopen (met index groter dan 0) ziet er als volgt uit:

**Definitie 49 (De vertaalmachine uitgebreid voor scopeknopen)**
*De volgende acties worden aan de read-back-machine toegevoegd:*

**De geïndiceerde ʎ-knoop** *Als de leeskop een ʎ-knoop met index n leest, worden de volgende acties uitgevoerd:*

- *Het n-de element van de context (van bovenaf geteld) en het n+1-de element van de context worden samengevoegd tot een paar met het n-de element als eerste element.*
- *Er wordt geen output gegenereerd.*

*Dit ziet er als volgt uit:*

$$P\left( \begin{array}{c} \sqcup_n \; [...,n:x,n+1:y,...] \\ (L) \end{array} \right) = P\left( \begin{array}{c} \sqcup_n \\ (L) \; [...,n:(x,y),...] \end{array} \right)$$

De read-backregel voor de re-open-scopeknoop (met index groter dan 0) is precies het tegenovergestelde van de regel voor de geïndiceerde ʎ-koop. Dus:

**Definitie 50 (Read-back voor re-open-scopeknopen)**

**De re-open-scopeknoop** *Als de leeskop een re-open-scopeknoop leest, worden de volgende acties uitgevoerd:*

- *Het n-de element van de context (van bovenaf geteld) wordt gesplitst. Het eerste element blijft staan en het tweede element wordt op de context onder het eerste gezet.*
- *Er wordt geen output gegenereerd.*

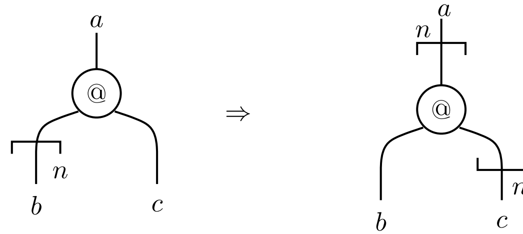
*Dit ziet er als volgt uit:*

$$P\left( n \quad [...,n:(x,y),...] \quad L \right) = P\left( n \quad L \quad [...,n:x,n+1:y,...] \right)$$

## 5.3 Het pushen van de (re-)open-scopeknoop

Met het ontstaan van nieuwe knopen ontstaan soms nieuwe problemen. Het kan nu namelijk zo zijn dat er re-open-scopeknopen of open-scopeknopen tussen de linkerpoort van een applicatie-knoop en een $\lambda$-knoop zitten. Deze moeten ook uit de weg kunnen worden gehaald. Dit kan met de volgende herschrijfregel:

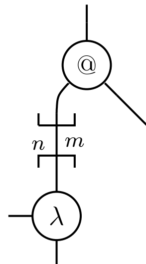
**Definitie 51 (De (re-)open-scope-push)**




Merk op dat er bij deze regel niets aan de index verandert. De knoop blijft in de scope waar hij al was. Er wordt een extra $\curlywedge$-knoop gegenereerd zodat de graaf in balans blijft. We zien dat de open-scopeknopen over de applicatie-knoop heen getild worden, terwijl de $\curlywedge$-knoop over de $\lambda$-knoop heen getild wordt. Dat beide knoopsoorten een andere richting op gaan, heeft te maken met dat de één een omkering is van de andere. Je kunt dit ook zien aan de keuze van de interactie-poort.

## 5.4 Annihilatie en wisseling van scopeknopen

Met deze regels zijn we er echter nog niet. De volgende situatie kan voor komen:



Deze situatie is niet op te lossen met de regels die we boven gegeven hebben. We kunnen dit probleem op twee manieren oplossen. We kunnen de knopen elkaar laten passeren of we kunnen de knopen elkaar laten annihileren. Dit laatste kan alleen als de knopen dezelfde index hebben:

**Definitie 52 (Annihilatie van scopeknopen)**

a, n, n, b ⇒ a, b

Wat hier gebeurt is dat twee scopes aan elkaar smelten tot één grotere scope, oftewel *scope merging*. Dit kan omdat we weten dat een van de scopes wordt geopend door een open-scopeknoop en dus een zogenaamde lege scope is.

Als de indexen van beide knopen niet gelijk zijn dan laten we de knopen elkaar passeren. Hiervoor zijn de volgende twee regels:

**Definitie 53 (Wisseling van scopeknopen)**

$n > m$ a, n, m, b ⇒ a, m, $n+1$, b

$m > n$ a, n, m, b ⇒ a, $m+1$, n, b

Bij deze regels wordt de hoogste van de twee indexen met 1 verhoogd. De knoop met deze index komt namelijk in een nieuwe scope. Intuïtief is dit als volgt te begrijpen. De geïndiceerde scopeknopen stellen scopes voor die over elkaar heen liggen. De index geeft aan hoeveel scopes zich onder de scope van de geïndiceerde scopeknoop bevinden. De scope met de hoogste index schuift over nog een scope heen en zijn index wordt dus met 1 opgehoogd. We zullen in Hoofdstuk 6 laten zien dat deze intuïtie correct is.

Met de herschrijfregels die we nu hebben gedefinieerd kunnen we alle combinaties van scopeknopen tussen een $\lambda$-operator en een applicatie weghalen.

**Lemma 2**
*Verborgen redexen kunnen altijd zichtbaar worden gemaakt door alle scopeknopen die zich tussen de linkerpoort van een applicatieknoop en een $\lambda$-knoop in bevinden*

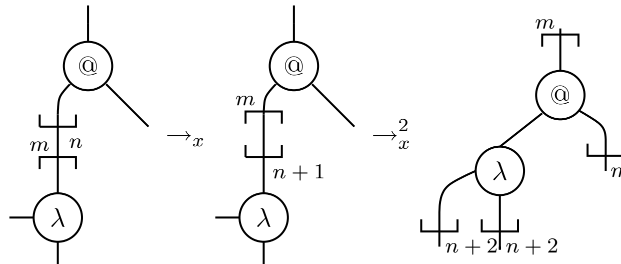
*weg te halen.* ***Bewijs:***
*Loop een pad omhoog vanaf de λ-knoop. Als de eerste knoop een applicatieknoop is, dan zijn we klaar. Anders bekijken we de volgende knoop:*

- *Als het een (re-)open-scopeknoop is, slaan we deze over en bekijken de volgende knoop.*
- *Als het een (geïndiceerde) ʎ-knoop is, dan voeren we net zo lang de scopeknopen wissel uit tot deze knoop bij de λ-knoop uitkomt. Dan voeren we een ʎ-push uit en beginnen opnieuw.*
- *Als we een applicatieknoop tegenkomen dan starten we het volgende algoritme:*
  *Loop een pad omlaag vanaf de applicatieknoop. Als de eerste knoop een λ-knoop is, dan zijn we klaar. Anders bekijken we de volgende knoop:*
  - *De volgende knoop is een (re-)open-scopeknoop, want alle ʎ-knopen zijn al verdwenen. Voer een (re-)open-scope push uit en begin opnieuw.*

*Door eerst de ʎ-knopen naar beneden te halen en daarna de (re-)open-scopeknopen naar boven, kunnen we een verborgen redex altijd zichtbaar krijgen.*

We geven een voorbeeld van de werking van dit lemma.

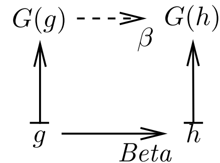
**Voorbeeld 18 (Scope-knopen uit de weg)** *Het voorbeeld uit Paragraaf 5.4 herschrijven we nu, tot de redex vrij is (met $m > n$):*



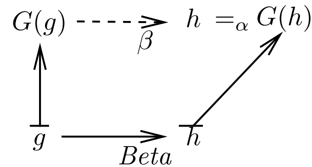

# 6 Correctheid van de herschrijfregels

In dit hoofdstuk zullen we bewijzen dat het graafherschrijfsysteem dat we in de vorige hoofdstukken hebben gedefinieerd, een semantisch correcte representatie is van de λ-calculus. We weten dat termen kunnen worden gereduceerd met behulp van $\beta$-reductie. De grafen kunnen worden herschreven door middel van de *Beta*-stap. We zullen nu moeten bewijzen dat deze regels hetzelfde semantische

gevolg hebben. Dus:
**Te Bewijzen:**




Dit is echter niet zonder meer te bewijzen. We hebben al gezien dat één graaf een klasse van $\alpha$-equivalente termen representeert. Als we een graaf naar een term vertalen, nadat we deze hebben gereduceerd met de *Beta*-stap, kan het zijn dat het resultaat niet hetzelfde is als het resultaat van de $\beta$-stap. We kunnen echter wel bewijzen dat de resultaten $\alpha$-equivalent is. Dus:

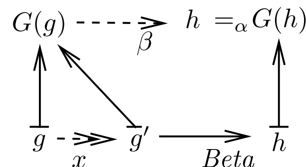



Andersom bevatten de grafen scope-informatie. De knopen die deze informatie representeren kunnen zorgen voor verborgen redexen. Deze redexen kunnen vrij worden gemaakt door middel van de andere herschrijfregels. Deze herschrijfregels zullen we vanaf nu de $x$-regels noemen.

**Definitie 54 (De $x$-regels)** *De $x$-regels zijn alle graafherschrijfregels behalve de Beta-stap.*

De $\lambda$-calculus bevat deze scope-informatie niet en dus ook geen regels die daarmee te maken hebben. We moeten bewijzen dat de $x$-regels geen semantisch effect hebben, oftewel dat het uitvoeren van een $x$-regel geen invloed heeft op de read-back van de graaf. In totaal ziet dit er als volgt uit:

**Te Bewijzen:**




Dit bewijs valt in drie delen uiteen:

1. De *Beta*-stap is een correcte representatie van de $\beta$-stap.
2. Een graaf representeert altijd een klasse van $\alpha$-equivalente termen.

3. Twee grafen die met $x$-regels naar elkaar toe te schrijven zijn, representeren altijd dezelfde klasse van $\alpha$-equivalente termen. Oftewel, $x$-regels hebben geen invloed op de read-back van de graaf.

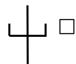
Van deze bewijzen hebben we de tweede al geleverd (zie Paragraaf 3.4). In Paragraaf 6.1 zullen we het derde bewijs leveren en in Paragraaf 6.3 geven we het eerste bewijs.

## 6.1 De $x$-regels

We weten dat de $x$-regels lokale herschrijfregels zijn. Iedere keer als we een $x$-regel uitvoeren, verandert er dus een begrensd deel van de graaf. Om te bewijzen dat het toepassen van de $x$-regels geen invloed heeft op de read-back hoeven we alleen maar het gedeelte te bekijken dat door de regel verandert. We moeten dan bekijken wat dit gedeelte voor invloed heeft op de read-back. Om sneller te kunnen zien wat de read-back-machine doet op een gedeelte van een graaf, introduceren we het *read-back pad*. Uit de definitie van het read-back algoritme volgt dat de leeskop altijd een pad naar beneden volgt door de graaf (Definitie 28 en Definitie 38). We kunnen dus een willekeurig pad door de graaf naar beneden nemen en bekijken wat de read-back-machine daarop doet.

**Definitie 55 (Het read-back-pad)** *Een read-back-pad is een pad naar beneden door een graaf, waarbij op het startpunt een context wordt gezet, de start-context. Bij iedere knoop op het pad wordt aangegeven hoe de read-back-machine de context zou manipuleren bij het lezen van de knoop en welke output zou worden gegenereerd. De context moet compatibel zijn met het read-back pad, dus het read-back algoritme loopt niet vast met de gegeven context. De context die bij de laatste knoop op het pad staat, noemen we de eindcontext.*

**Voorbeeld 19 (Een pad dat geen read-back-pad is)**



*Dit pad is geen read-back-pad want de context die op het startpunt staat is niet compatibel met de knoop op het pad. De ⋏-knoop wil een element van de stack weggooien, maar de stack is leeg (□).*

Als we nu willen bewijzen dat een regel geen invloed heeft op het resultaat van het read-back algoritme, dan moeten we het kleinste read-back pad of de kleinste read-back paden bekijken waarop de regel zich afspeelt. Als de startcontexten en de eindcontexten voor en na het toepassen van de regel hetzelfde zijn en de output die wordt gegenereerd hetzelfde is, dan heeft de regel geen invloed op het resultaat van het read-back algoritme.

## 6.2 Gebalanceerde en transparante termen

In Hoofdstuk 4 kwam al ter sprake dat het read-back-algoritme alleen werkt als een graaf in balans is. We moeten dus voor iedere herschrijfregel bekijken of een pad dat in balans is, na het toepassen van die regel op dat pad, nog steeds in balans is. In Definitie 30 beschreven we waar een pad aan moet voldoen om in balans te zijn. Na de introductie van nieuwe knoopsoorten, moeten we deze definitie aanpassen:

**Definitie 56 (Paden in balans, aangepast)** *Een pad is in balans als geldt:*

1. *Voor elke scope die op het pad wordt geopend geldt dat deze op datzelfde pad weer geheel of tijdelijk wordt gesloten.*
2. *Als een scope die niet op het pad geopend is, tijdelijk wordt gesloten, dan moet deze weer worden geopend op het pad.*
3. *Een scope die niet op het pad wordt geopend kan niet definitief worden gesloten.*

Dit betekent dat voor het read-back algoritme op een pad in balans geldt:

1. Als er een element aan de startcontext wordt toegevoegd op een pad in balans, dan wordt dit element op hetzelfde pad weer weggegooid, of als secundaire informatie opgeslagen.
2. Als een element van de primaire informatie van de startcontext wordt opgeslagen als secundaire informatie, dan wordt dit element op hetzelfde pad weer tot primaire informatie gemaakt.
3. Elementen van de startcontext worden niet weggegooid.

**Definitie 57 (Transparantie)** *Een pad is transparant als de startcontext en de eindcontext van het read-back pad dat over dit pad loopt, bijna hetzelfde zijn. Het enige verschil dat is toegestaan is dat de eindcontext meer secundaire informatie mag bevatten, oftewel* $\Pi(eindcontext) = \Pi(begincontext)$*.*

In het bewijs voor correctheid van het systeem hebben we nodig dat als een pad in balans is, het ook transparant is. Om dit te bewijzen, bekijken we wat voor manipulaties er kunnen worden uitgevoerd op de startcontext en we laten zien dat deze op een pad in balans allemaal weer ongedaan worden gemaakt.

**Lemma 3 (Als een pad in balans is, is het ook transparant)** Bewijs:

**Het toevoegen van informatie aan de context** *Als we informatie aan de startcontext toevoegen, moet deze ook weer worden weggegooid op een transparant pad, of aan het einde zijn opgeslagen als secundaire informatie. Uit Definitie 56, punt 1, volgt dat alle elementen die aan de startcontext worden toegevoegd, worden weggegooid of tot secundaire informatie worden opgeslagen. Uit Definitie 56, punt 2, volgt dat alle nieuw toegevoegde secundaire informatie die tot primaire informatie wordt gemaakt, weer wordt*

*opgeslagen als secundaire informatie. Alle elementen die aan de startcontext worden toegevoegd zullen aan het eind van het pad dus zijn weggegooid of zijn opgeslagen als secundaire informatie.*

**Het tijdelijk opslaan van informatie op de context** *Als er primaire informatie van de startcontext wordt opgeslagen als secundaire informatie, moet deze weer tot primaire informatie worden gemaakt wil het pad transparant zijn. Uit Definitie 56, punt 3, volgt dat dit altijd gebeurt.*

**Het weggooien van informatie van de context** *Als er van de startcontext informatie zou worden weggegooid, dan zou het pad niet transparant zijn, want deze informatie zouden we nooit kunnen terugkrijgen. Uit Definitie 56, punt vier, volgt echter dat dit niet kan gebeuren op een pad dat in balans is.*

Nu de begrippen *read-back pad*, *balans* en *transparantie* zijn gedefinieerd, kunnen we bewijzen dat de $x$-regels geen invloed hebben op de read-back van een graaf. Om dit te bewijzen zullen we elke regel afzonderlijk bekijken. We werken in omgekeerde volgorde van de volgorde waarin ze gedefinieerd werden in Hoofdstuk 5, omdat de laatste regels makkelijker te bewijzen zijn. We stellen bij iedere regel het kleinste read-back-pad vast waarop de regel plaats vindt en we bewijzen voor dat pad de volgende drie punten:

**Context** Als we voor en na het toepassen van de regel met dezelfde startcontext beginnen, dan zijn de eindcontexten gelijk.

**Output** De output die op het pad wordt gegenereerd is voor en na het toepassen van de regel gelijk.

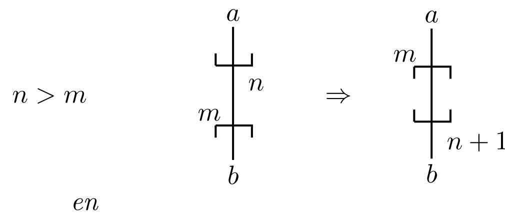
**Balans** De regel brengt een pad dat in balans is, niet uit balans.

Als deze punten zijn bewezen voor de regel, dan betekent dat dat de regel geen invloed heeft op het read-back algoritme. We zullen tijdens dit bewijs steeds zo algemeen mogelijke contexten gebruiken. We zullen van de contexten alleen de elementen geven die hij minimaal moet hebben om de read-back niet te laten falen.

**Stelling 3**
*De $x$-regels hebben geen invloed op de read-back*
***Bewijs:*** *We zullen nu de regels stuk voor stuk behandelen.*

**Het wisselen van twee scope-knopen** *We moeten laten zien dat de volgende twee regels geen invloed hebben op de read-back:*

$n > m$




*en*

$m > n$ (diagram: node $a$ on top; a node with index $n$, then a node with index $m$; node $b$ at the bottom) $\Rightarrow$ (node $a$ on top; a node with index $m+1$, then a node with index $n$; node $b$ at the bottom)

*We beginnen met de gevallen waarbij de indexen groter zijn dan* $0$*. Daarna bekijken we de gevallen waar één van de knopen geen index heeft.*

$n, m > 0$ *We bewijzen de stelling eerst voor de eerste van deze twee regels. Het read-back pad dat we bekijken begint boven de geïndiceerde* $\curlywedge$*-knoop en eindigt onder de re-open-scope knoop. De read-back paden zien er als volgt uit:*

$a$

(node $n$) $[..., m : (v, w), ..., n : x, n + 1 : y, ...]$

(node $m$) $[..., m : (v, w), ..., n : (x, y), ...]$

$b$ $[..., m : v, m + 1 : w, ..., n + 1 : (x, y), ...]$

*en*

$a$

(node $m$) $[..., m : (v, w), ..., n : x, n + 1 : y, ...]$

(node $n+1$) $[..., m : v, m + 1 : w, ..., n + 1 : x, n + 2 : y, ...]$

$b$ $[..., m : v, m + 1 : w, ..., n + 1 : (x, y), ...]$

**Context** *We zien de eindcontexten voor en na het toepassen van de regel hetzelfde zijn.*

**Output** *De output die op de read-back paden wordt gegenereerd is in beide gevallen hetzelfde, namelijk niets.*

**Balans** *Voor en na het toepassen van de regel worden dezelfde elementen in de stack gemanipuleerd, alleen in een andere volgorde. Dus in beide gevallen wordt element* $v$ *tot primaire informatie gemaakt en wordt element* $y$ *opgeslagen als secundaire informatie. Als deze regel zich afspeelt op een pad dat in balans is, dan zal het pad na het toepassen van de regel dus nog steeds in balans zijn.*

*De read-back paden van de tweede regel* ($m > n$) *zien er als volgt uit:*

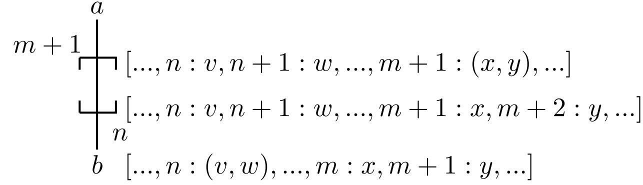
$a$

(node $n$) $[..., n : v, n + 1 : w, ..., m + 1 : (x, y), ...]$

(node $m$) $[..., n : (v, w), ..., m : (x, y), ...]$

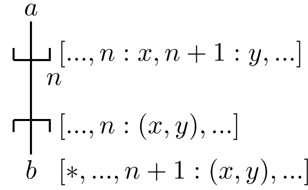
$b$ $[..., n : (v, w), ..., m : x, m + 1 : y, ...]$

*en*

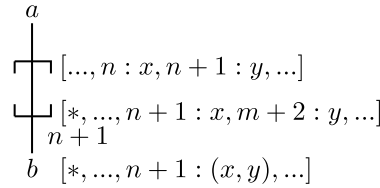



*Het bewijs voor deze regel gaat verder analoog aan dat voor de eerste regel.*

$m = 0, n > 0$ *De read-back paden voor deze regel zien er als volgt uit:*




*en*




**Context** *We zien de eindcontexten voor en na het toepassen van de regel hetzelfde zijn.*

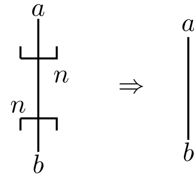
**Output** *De output die op de read-back paden wordt gegenereerd is in beide gevallen hetzelfde, namelijk niets.*

**Balans** *Ook bij deze regel worden dezelfde elementen gemanipuleerd. Dat betekent dat ook deze regel de balans van een pad niet verstoord.*

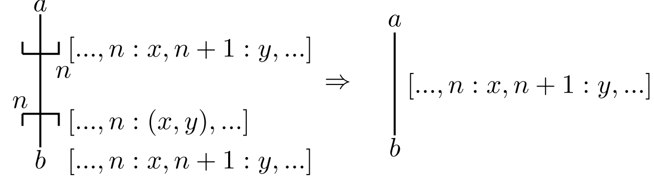
$n = 0, m > 0$ *Het geval $n = 0, m > 0$ is analoog aan het bewijs van de vorige regel. Alleen de startcontext is $[*, ..., m + 1 : (x, y), ...]$.*

### Het annihileren van twee scope-knopen

$n$ **is groter dan** $0$ *We moeten van de volgende regel bewijzen dat het toepassen ervan geen invloed heeft op de read-back:*



*Het read-back pad dat we bekijken begint boven de ⋏-knoop en eindigt onder de open-scope knoop. Na het het toepassen van de regel vallen het begin en het eind van het read-back pad samen.*

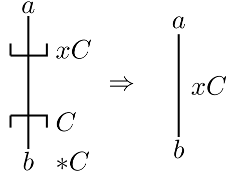


**Context** *De contexten zijn gelijk. Merk hierbij op dat na het toepassen van de regel de start- en eindcontext samen vallen.*

**Output** *Er wordt voor en na het toepassen van de regel geen output gegenereerd op dit read-back-pad.*

**Balans** *Het pad is na het toepassen van de regel nog steeds in balans omdat twee knopen verdwijnen, waarvan de tweede het werk van de eerste ongedaan maakt.*

$n$ **is** 0**, oftewel geen index** *We bekijken bij deze regel het read-back-pad dat begint boven de ⋏-knoop en eindigt onder de open-scopeknoop. Na het toepassen van de regel, vallen het begin en het einde van het read-back pad weer samen:*

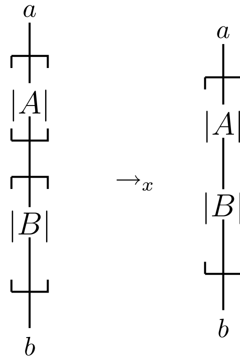


**Context** *We zien hier dat de contexten ongelijk zijn. Er volgt dus niet direct dat deze regel geen invloed heeft op de read-back van de rest van de graaf. De regel heeft echter geen invloed op de read-back en wel om de volgende reden:*
*Het element $*$ wordt in het read-back algoritme alleen maar gebruikt om aan te geven dat we met een lege scope te maken hebben, een scope waar zich geen variabelen in bevinden. Dit element wordt dus enkel gebruikt om weggegooid te worden. Dat de $*$ na de regel is vervangen door een $x$ heeft dus geen invloed op de read-back want de $x$ zal ook alleen maar worden weggegooid.*

**Output** *Er wordt geen output gegenereerd op het pad, niet voor en niet na het toepassen van de regel.*

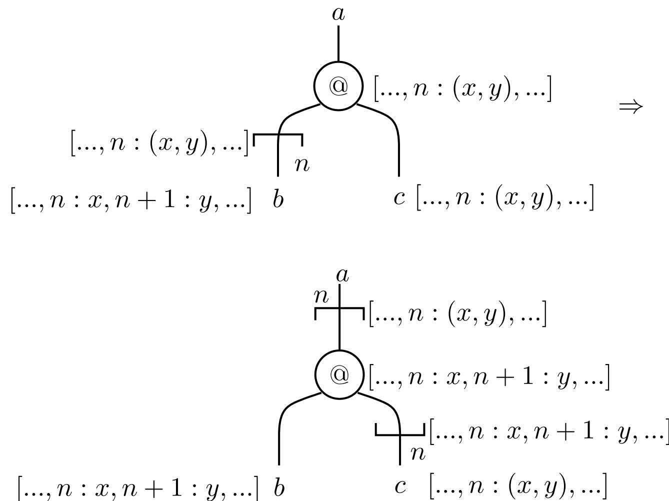
**Balans** *We weten uit Definitie 56 dat het element $*$ op een pad in balans, wordt weggegooid of tot secundaire informatie wordt opgeslagen. Het is immers nieuw toegevoegde informatie. We weten ook dat de ⋏-knoop voor de regel het element $x$ weggooit. Op een pad in balans is $x$ dus nieuw toegevoegde informatie, want informatie die al op de stack stond kan niet niet worden weggegooid.*

*Na het toepassen van de regel staat element $x$ nog op de stack. Om het pad in balans te houden moet dit element tot secundaire informatie worden opgeslagen of weggegooid. Omdat $x$ na het toepassen van de regel op de plek van $*$ is terecht gekomen, weten we dat dit inderdaad ook het geval zal zijn. Het pad blijft dus, na het toepassen van de regel in balans.*
*In het volgende plaatje wordt meer inzichtelijk gemaakt wat er door het toepassen van de regel eigenlijk gebeurt:*




*Waarbij we met $|A|$ en $|B|$ de read-back-paden door respectievelijk $A$ en $B$ bedoelen.*

**Het pushen van de re-open-scopeknoop** *Bij deze regel moeten we twee read-back paden bekijken. Eén loopt van boven de applicatieknoop tot onder de re-open-scopeknoop. De ander loopt van boven de applicatie knoop tot rechtsonder de applicatieknoop:*



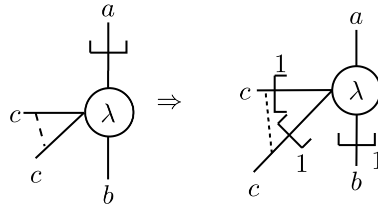

**Context** *We zien dat de contexten voor en na het toepassen van de regel gelijk zijn.*

**Output** *De enige output die op de paden wordt gegenereerd, komt van de applicatieknoop. Bij het lezen van de applicatieknoop wordt altijd dezelfde output gegenereerd, wat er ook op de context staat.*

**Balans** *De applicatieknoop heeft geen invloed op de balans van een pad, bij het lezen van een applicatieknoop wordt de context niet gemanipuleerd. De applicatieknoop kan de balans van een pad dus niet verstoren.*

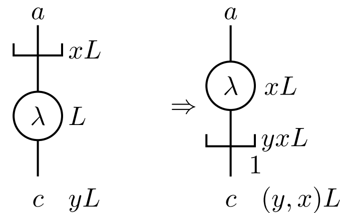
*Op het eerste read-back pad ligt voor en na het toepassen van de regel één scopeknoop. Op dit pad verandert er niets aan de balans. Op het tweede pad liggen er na het toepassen van de regel twee scope-knopen, terwijl er voor het toepassen van de regel geen scope-knopen lagen. De eerste van deze knopen maakt secundaire informatie tot primaire informatie. De tweede knoop maakt dit weer ongedaan. De balans van het pad blijft hetzelfde.*

**Het pushen van de end-of-scopeknoop** *We moeten bewijzen dat de volgende regel geen invloed heeft op de read-back:*




*In deze regel hebben we te maken met de $\lambda$-knoop. We moeten in deze regel onderscheid maken tussen twee soorten paden, namelijk de paden die uiteindelijk weer bij de $\lambda$-knoop komen en de paden die dat niet doen. Bij de eerste soort komt er na het toepassen van de regel aan het eind van het read-back pad nog een re-open-scopeknoop bij en bij de tweede soort gebeurt dat niet. We zullen eerst de tweede soort bekijken.*

**Het pad komt niet uit bij de $\lambda$-knoop** *We weten dat het read-back pad dat we bekijken niet terugkomt bij de $\lambda$-knoop. We kunnen de tak aan de variabelenpoort dus buiten beschouwing laten. We bekijken dus het read-back pad dat begint boven de $\lambda$-knoop en eindigt onder de $\lambda$-knoop. Het read-back ziet dat er als volgt uit:*




*Met $L$ een willekeurige context.*

**Context** *De eindcontexten zijn ongelijk. Na het toepassen van de regel bevat de eindcontext secundaire informatie. Het gedeelte van de graaf dat zich onder deze regel bevindt, zal dus na het toepassen van de regel met een andere context worden vertaald dan voordat de regel werd toegepast. Van secundaire informatie weten we dat deze pas invloed op de read-back uitoefent als het tot primaire informatie wordt gemaakt. We weten echter zeker dat dit niet gebeurt en wel om de volgende reden:*

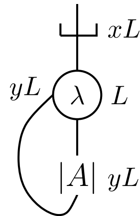
*Om van secundaire informatie, primaire informatie te maken is er een re-open-scope knoop nodig met de juiste index. We weten echter dat deze niet in de graaf aanwezig is (als we de read-back paden die weer bij de* $\lambda$*-knoop komen, buiten beschouwing laten). Als hij er wel zou zijn dan waren we voor het toepassen van de regel al vastgelopen. Deze knoop had dan namelijk secundaire informatie tot primaire informatie willen maken op een niveau waar geen secundaire informatie is.*

*De nieuwe secundaire informatie zal dus geen invloed op de read-back hebben.*

**Output** *De output voor en na het toepassen van de regel is hetzelfde, namelijk de output die door de* $\lambda$*-knoop gegenereerd wordt.*

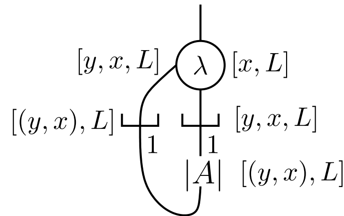
**Balans** *Als de regel plaatsvindt op een pad dat in balans is, dan volgt uit Definitie 56 dat voordat de regel wordt toegepast, de* ʎ*-knoop een scope afsluit die op het pad geopend is en dat de scope die door de* $\lambda$*-knoop geopend wordt ook weer op het pad (tijdelijk) gesloten wordt. Na het toepassen van de regel wordt de scope die eerst door de* ʎ*-knoop geheel gesloten werd, tijdelijk gesloten door de geïndiceerde* ʎ*-knoop. Volgens Definitie 56 is dit echter voldoende om het pad in balans te houden. De scope van de* $\lambda$*-knoop zal nog steeds worden afgesloten want verder op het pad is niets veranderd. Het pad is dus na het toepassen van de regel nog steeds in balans.*

**Het pad komt weer bij de $\lambda$-knoop uit** *Het read-back pad dat we nu bekijken begint boven de* ʎ*-knoop en eindigt als de* $\lambda$*-knoop voor de tweede maal wordt bereikt. Eerst bekijken we het read-back pad voordat de regel is toegepast:*




*Met* $|A|$ *bedoelen we het pad dat door de subgraaf* $A$ *loopt. Als de* $\lambda$ *operator meerdere variabelen bindt, dan zullen er ook meerdere paden door* $A$ *lopen die weer bij de* $\lambda$*-knoop uitkomen. Voor dit bewijs*

*beschouwen we echter één willekeurig gekozen pad. We weten dat $|A|$ in balans en dus transparant is. Daaruit volgt dat de primaire informatie van de context aan het begin van $|A|$ hetzelfde is als aan het einde van $|A|$. Het pad ziet er na het toepassen van de regel als volgt uit:*




**Context** *We zien dat de contexten aan het eind van het read-back-pad niet gelijk zijn. Het eerste element is echter wel gelijk. De read-back-machine gebruikt bij het lezen van de $\lambda$-knoop vanaf de variabelenpoort, het eerste element van de stack en gooit de rest weg, omdat het algoritme hier stopt. Dat de rest van de stack anders is, heeft dus geen invloed op de read-back.*

**Output** *De vraag is nu alleen nog of er voor en na de toepassing van de regel wel dezelfde output wordt gegenereerd. Er wordt op drie plaatsen output gegenereerd:*

- *De $\lambda$-knoop.*
- *De variabele (ofwel het pad, als het weer terugkomt bij de $\lambda$-knoop)*
- *Het pad $|A|$*

*We zullen bij deze drie plaatsen moeten nagaan of de output voor en na het toepassen van de regel gelijk is:*

**De $\lambda$-knoop** *Bij het lezen van een $\lambda$-knoop zet de read-back machine alleen iets op de stack en haalt er geen informatie af. Dat de context bij het lezen van de $\lambda$-knoop voor en na de regel verschilt, heeft dus geen invloed bij het vertalen van die knoop. De output is dus gelijk.*

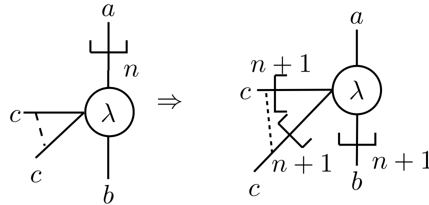
**De variabele** *Bij het vertalen van een variabele is er wel informatie van de context nodig, namelijk het bovenste element. Maar het bovenste element van de eindcontexten is gelijk, voor en na het toepassen van de regel. Ook hier wordt dus voor en na het toepassen van de regel dezelfde output gegenereerd.*

**Het pad $|A|$** *Het pad $|A|$ wordt voor en na het toepassen van de regel met een andere context vertaald. De context van na de regel heeft secundaire informatie op het eerste niveau staan, waar de context van voor het toepassen van de regel enkel primaire informatie heeft. Deze informatie heeft echter geen invloed op de read-back van $|A|$, tenzij het primaire informatie wordt. We kunnen hier echter hetzelfde argument als bij*

*het vorige geval gebruiken om aan te tonen dat dit niet zal gebeuren. Deze secundaire informatie zal dus geen invloed hebben op de output die wordt gegenereerd op $|A|$.*

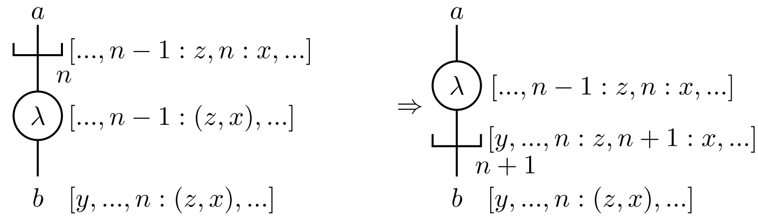
**Balans** *We zien dat het pad dat van de $\lambda$-knoop naar zichzelf loopt, na het toepassen van de regel nog steeds in balans is. In $|A|$ is niets veranderd dus dat gedeelte van het pad is in balans. De geïndiceerde ⋏-knoop die vooraan het pad is verschenen wordt in balans gehouden door de re-open-scopeknoop aan het einde van het pad in balans gehouden. Daardoor is het hele pad in balans. De re-open-scopeknoop maakt de stackmanipulaties van de geïndiceerde ⋏-knoop weer ongedaan.*

**Het pushen van de geïndiceerde end-of-scopeknoop** *De regel die we nu correct moeten bewijzen is de volgende:*




*Deze situatie lijkt veel op de regel waar de index van de ⋏-knoop 0 is, maar is eenvoudiger. We zullen echter wel dezelfde gevalsonderscheiding moeten hanteren.*

**Het pad komt niet uit bij de $\lambda$-knoop** *We bekijken eerst weer het geval van een read-back pad dat niet bij de $\lambda$-knoop uitkomt. Het read-back pad dat we nu bekijken loopt van boven de geïndiceerde ⋏-knoop tot onder de $\lambda$-knoop.*

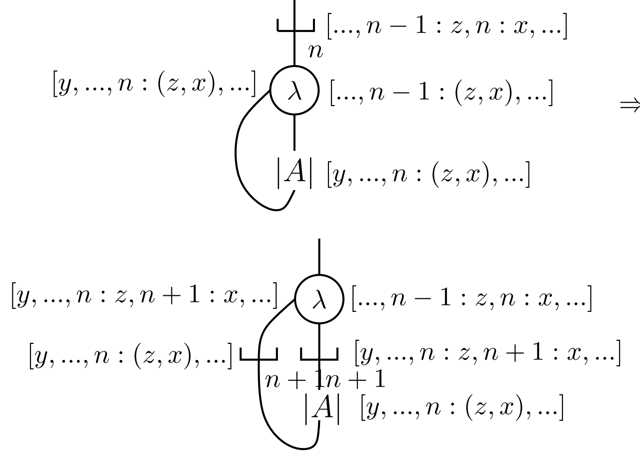



**Context** *We zien dat de contexten gelijk zijn.*

**Output** *De output is ook gelijk, omdat bij het vertalen van een $\lambda$-knoop de context niet van belang is.*

**Balans** *Het pad waarop de regel plaatsvindt blijft ook in balans. De geïndiceerde ⋏-knoop blijft invloed hebben op dezelfde scope, voor en na het toepassen van de regel.*

**Het pad komt weer bij de $\lambda$-knoop uit** *We bekijken het read-back pad van boven de geïndiceerde ƛ-knoop tot aan de variabelenpoort van de $\lambda$-knoop.*




**Context** *We zien dat ook hier de contexten aan het eind van het read-back pad ongelijk zijn, maar net als in het vorige gedeelte is het eerste element, $y$, wel gelijk en dat is genoeg om geen invloed op de read-back te hebben.*

**Output** *De output zal voor en na het toepassen van de regel hetzelfde zijn. De $\lambda$- knoop wordt op dezelfde manier vertaald, de variabele wordt op dezelfde manier vertaald en ook het pad $|A|$ zal dezelfde output genereren, want het wordt voor en na de regel met dezelfde context vertaald.*

**Balans** *Na de regel is het pad dat van de $\lambda$-knoop naar zichzelf loopt nog steeds in balans. De scope van buiten het pad dat door de ƛ-knoop tijdelijk wordt gesloten wordt vlak voor het eind van het pad weer geopend door de re-open-scopeknoop die is ontstaan. Het pad voldoet dus nog steeds aan Definitie 56 (punt 3).*

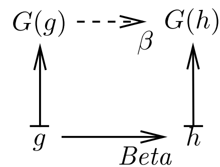
**Conclusie** *We hebben gezien dat alle scope-informatie regels uit Hoofdstuk 5, geen invloed hebben op het resultaat van de read-back functie. Als een graaf een representatie is van een $\lambda$-term en we passen er $x$-regels op toe, dan blijft de graaf dezelfde $\lambda$-term representeren.*

## 6.3 *Beta* en $\beta$

In deze paragraaf zullen we bewijzen dat de $\beta$- en de *Beta*-stap hetzelfde semantische gevolg hebben. Dit wil zeggen dat als een term en een graaf corresponderen en we zetten in de graaf een *Beta*-stap en in de term een $\beta$-stap, de resultaten ook corresponderen. De stappen kunnen natuurlijk niet willekeurig zijn. We moeten zorgen dat de $\beta$-stap die wordt gezet, wordt gerepresenteerd

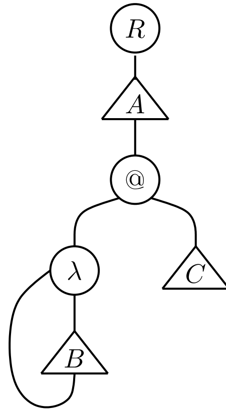
door de *Beta*-stap die we zetten. We zullen dus een graaf nemen met een *Beta*-redex erin en deze vertalen naar een term. Tijdens deze vertaling houden we expliciet bij naar welke $\beta$-redex de *Beta*-redex vertaald wordt. De $\beta$-redex die we krijgen en de *Beta*-redex die we al hadden, worden gereduceerd. Daarna vertalen we het resultaat van de *Beta*-stap naar een term. We bewijzen dat deze term $\alpha$-equivalent is met het resultaat van de $\beta$-stap. Een sterkere claim zou zijn dat ze exact hetzelfde zijn. Hiervoor zouden we kunnen zorgen door de variabelenamen die we vrij mogen kiezen bij de terugvertaling, hetzelfde te kiezen als de variabelenamen uit de oorspronkelijke term. We maken gebruik van deze vrije keuze tijdens het bewijs.

**Stelling 4**
*De Beta-stap is een correcte representatie van de $\beta$-stap.*
***Te Bewijzen:***



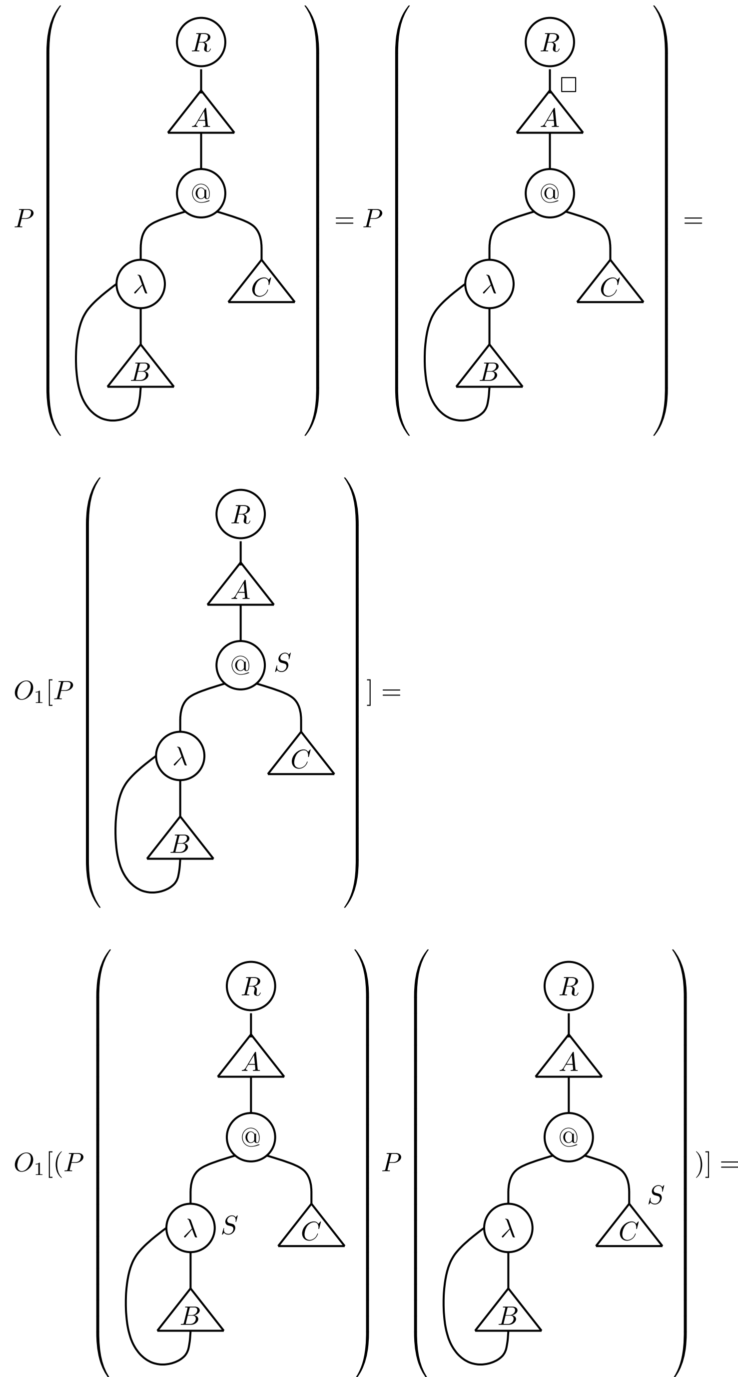

*We zullen het bewijs eerst geven voor het geval dat de $\lambda$-operator wel variabelen bindt en daarna voor het geval dat de $\lambda$-operator geen variabelen bindt.*

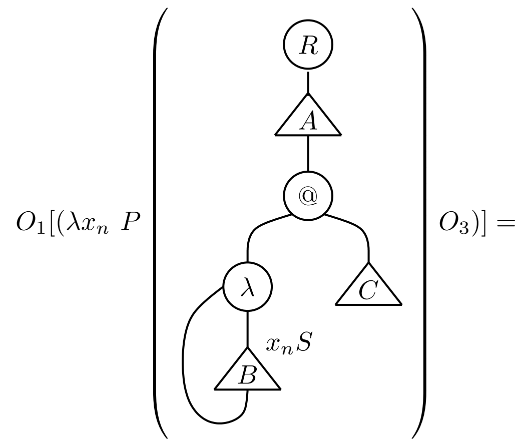
**Gebonden variabelen** *We nemen een willekeurige graaf met een Beta-redex waarin een variabele wordt gebonden door de $\lambda$-knoop die deel uitmaakt van deze redex. Dit ziet er als volgt uit:*



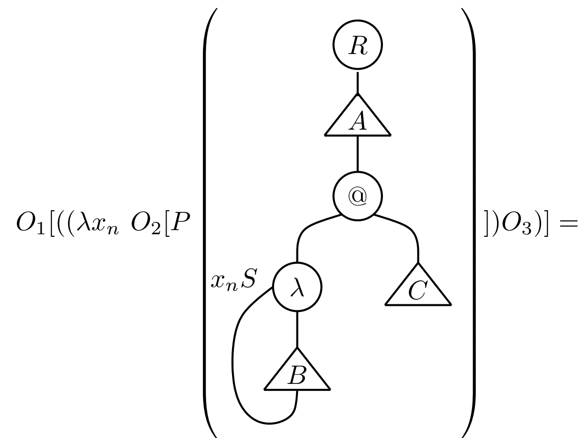

*We beschouwen hier enkel het geval dat er één variabele gebonden wordt. Het bewijs voor deze willekeurig gekozen variabele is echter analoog aan het bewijs voor elke andere gebonden variabele.*

*Deze graaf gaan we vertalen naar een term met de read-back machine:*
*Met $O_1$, $O_2$ en $O_3$ de vertalingen van respectievelijk $A$, $B$ en $C$*



$O_1[(\lambda x_n\ P$  $O_3)] =$



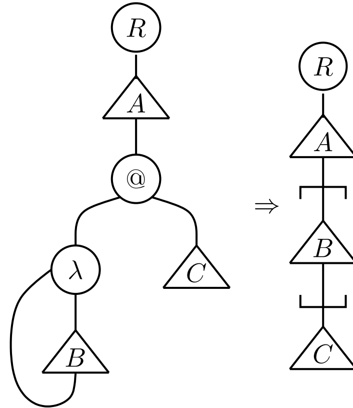
$O_1[((\lambda x_n\ O_2[P$  $])O_3)] =$



$O_1[((\lambda x_n\ O_2[x_n])O_3)]$

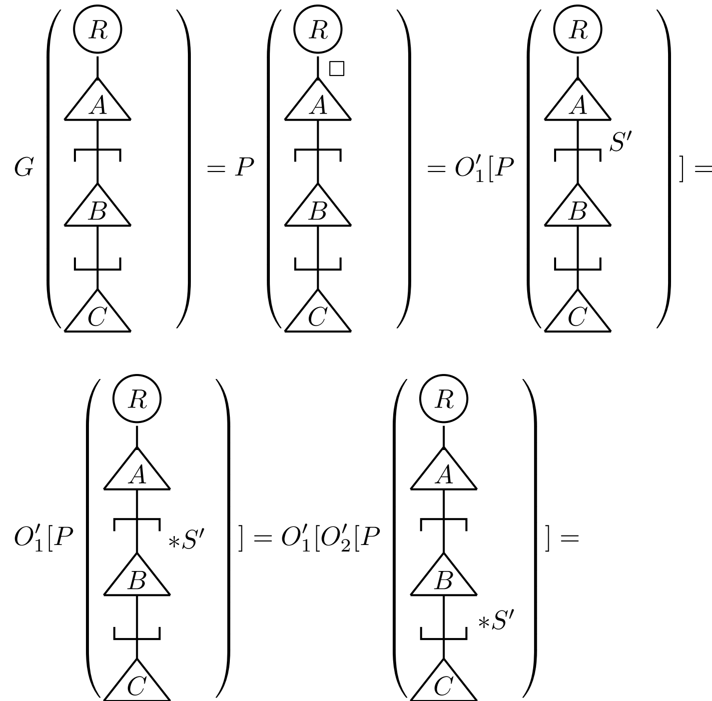
*We zetten nu in de graaf en in de term respectievelijk de Beta- en de $\beta$-stap:*

**De** *Beta***-stap**




**De** $\beta$**-stap** $O_1[((\lambda x_n\ O_2[x_n])O_3)] \ \rightarrow_\beta\ O_1[O_2[O_3]]$

*We zullen nu het resultaat van de Beta-stap vertalen naar een term en bewijzen dat deze hetzelfde is als het resultaat van de* $\beta$*-stap:*



$$O'_1[O'_2[P\left(\begin{array}{c} R \\ | \\ A \\ | \\ B \\ | \\ C \end{array} S'\right)] = O'_1[O'_2[O'_3]]$$

*We moeten nu alleen nog laten zien dat $O_i$ gelijk is aan $O_i$.*

- *$O_1$ is de output die de read-back-machine geeft als hij subgraaf $A$ vertaald met een lege context. Tijdens de vertaling moeten we vrije variabelenamen kiezen, maar we mogen zelf weten welke. $O'_1$ is de output van de read-back machine als deze subgraaf $A$ vertaald met een lege context. Als we tijdens deze vertaling dezelfde variabelenamen kiezen als in het geval dat $O_1$ verkregen werd, dan zullen we hetzelfde resultaat krijgen en zal $O'_1$ dus gelijk zijn aan $O_1$. De contexten $S$ en $S'$, die de read-back-machine zal gebruiken om de rest van de graaf te vertalen, zullen in dat geval ook gelijk zijn. Op de context staan op dat moment immers de variabelenamen die we zelf mochten kiezen.*
- *$O_3$ en $O'_3$ zijn de outputs van de read-back-machine als deze subgraaf $C$ vertaald met respectievelijk context $S$ en $S'$. Als we bij de vertaling van $A$ echter dezelfde variabelen-namen hebben gekozen, dan zullen $S$ en $S'$ gelijk zijn. Als we bij vertaling van subgraaf $S$ ook weer dezelfde variabelenamen kiezen, dan zijn $O_3$ en $O'_3$ ook gelijk zijn.*
- *$O_2$ is de output van de read-back-machine als deze subgraaf $B$ vertaald met context $xS$. We weten dat als de read-back-machine subgraaf $B$ leest, er geen knopen zijn waarbij de read-back machine element $x$ gebruikt bij het genereren van output. $x$ zal dus niet in $O_2$ voorkomen.*

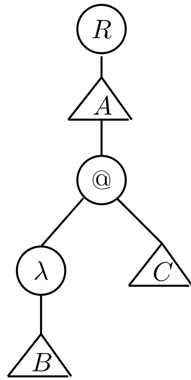
  *$O'_2$ is de output van de read-back-machine als deze subgraaf $B$ vertaald met context $*S'$. Element $*$ zal niet voorkomen in $O'_2$ om dezelfde reden als dat $x$ niet voorkomt in $O_2$. Als we bij het vertalen van subgraaf $A$ twee maal dezelfde variabelenamen gekozen hebben, dan zijn $S$ en $S'$ gelijk. Als we ook nu dezelfde variabelenamen kiezen in beide vertalingen, dan zullen $O_2$ en $O'_2$ gelijk zijn. De contexten zijn immers gelijk op hun eerste element na, maar daarvan weten we dat het niet in de output voor zal komen.*

*Dus:*

$$O_1[O_2[O_3]] = O'_1[O'_2[O'_3]]$$

*En dat betekent dat de Beta-regel een correcte representatie is van de $\beta$-regel.*

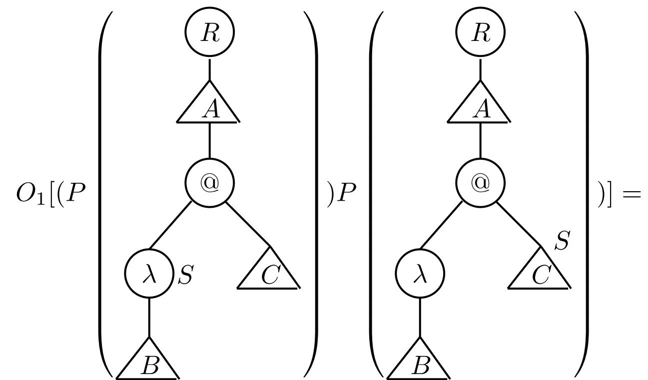
**Geen variabelen** *We nemen een willekeurige graaf met een Beta-redex waar geen backpointers aan de $\lambda$-knoop verbonden zijn:*

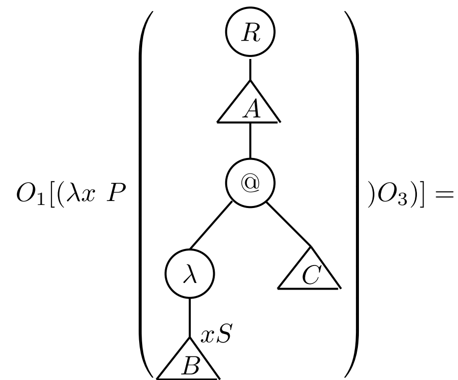



*Deze graaf vertalen we naar een term:*

$G$ ( R A @ λ C B ) $= P$ ( R □ A @ λ C B ) $=$

$O_1[P$ ( R A @ S λ C B ) $] =$



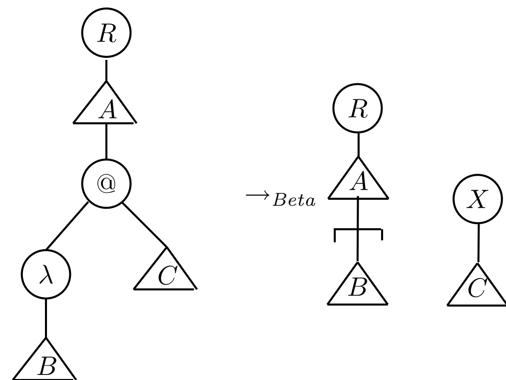




$O_1[(\lambda x\ O_2)O_3)]$

*Waarbij we weten dat $x$ niet vrij voorkomt in $O_2$.*

*Vervolgens zetten we de $\beta$- en de Beta-stap.*

*De Beta-stap*



*De β-stap*

$$O_1[(\lambda x\ O_2)O_3)] \ \rightarrow_\beta \ O_1[O_2]$$

*Nu vertalen we het resultaat van de Beta-stap naar een term. We zorgen er voor, net als in het vorige geval, dat de variabelenamen die we bij de vertaling kiezen voor en na het zetten van de Beta-stap, hetzelfde zijn:*

$G(\ldots) = P(\ldots) =$

$O_1[P(\ldots)] = O_1[P(\ldots)] =$

$O_1[O_2]$

*Ook in het geval dat er geen variabelen gebonden worden door de λ-operator, is de Beta-stap een correcte representatie van de β-stap.*

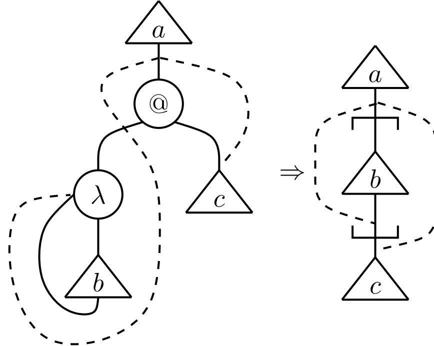
### 6.3.1 Graaf in balans na *Beta*-stap

Het read-back algoritme werkt alleen als een graaf in balans is. We moeten dus ook bewijzen dat de balans van een graaf niet wordt verstoord door de *Beta*-stap.

**Lemma 4**
*Transparantie en gebalanceerdheid blijven bewaard onder Beta-stap.*

***Bewijs:***
*De λ-knoop wordt vervangen door een open-scope knoop. Deze knoop houdt na de stap alle ⋏-knopen die de scope van de λ-knoop afsloten, in balans. Op het pad dat naar subgraaf C loopt, wordt een extra ⋏-knoop gezet om de nieuwe open-scope ook op dit pad in balans te houden en om te zorgen dat vrije variabelen in C niet onbedoeld worden gebonden. Om te bewijzen dat de graaf in balans blijft, moeten we twee read-back paden bekijken. De eerste loopt van boven de applicatie knoop (voor de stap gezet is) tot aan de onderkant van de λ-knoop.*

*De tweede loopt van boven de applicatieknoop tot subgraaf C. De paden lopen dus voor en na de Beta-stap als volgt:*




*Op het eerste pad komt voor het toepassen van de regel één knoop voor die invloed heeft op de context, dat is de $\lambda$-knoop. Na het toepassen van de regel is deze vervangen door een open-scope knoop, die een scope opent in plaats van de $\lambda$-operator. Dit pad is dus na het toepassen van de regel nog steeds in balans. Op het tweede pad staan voor het toepassen van de regel geen scope-knopen en na het toepassen van de regel twee. Deze knopen maken elkaars werk ongedaan, het pad blijft dus in balans. Ook de Beta-stap bewaard dus balans en transparantie.*

# 7 Efficiëntie

We hebben het bewijs geleverd dat ons graafherschrijfsysteem met expliciete scope-informatie correct werkt. Nu zullen we ingaan op de efficiëntie van dit systeem ten opzichte van andere systemen. We zullen bij de vergelijkingen tussen de systemen een schatting maken van de kosten die nodig zijn om bepaalde termen tot hun normaalvorm te reduceren. Hierbij moet wel worden opgemerkt dat het systeem dat in deze scriptie wordt beschreven nog niet optimaal werkt volgens Definitie 1. Dit gebeurt pas als we ook de sharing-operator aan het systeem toevoegen. De taak van de expliciete scope-operator is, naast het expliciet bijhouden van de scope, om de sharing-operator goed te laten werken. We zullen zien dat het toevoegen van de expliciete scope-operator er voor zorgt dat er meer herschrijfstappen nodig zijn om tot een normaalvorm te komen, oftewel een daling in de efficiëntie van het systeem. Dit is echter nodig om het systeem uiteindelijk optimaal te krijgen.

In de eerste paragraaf ga ik in op de efficiëntie van ons systeem na het toevoegen van de expliciete scope operatoren. Dit doe ik door het systeem te vergelijken met de $\lambda x$-calculus met De Bruijn-indices (zie [1]). Dit is een $\lambda$-calculus waar de substitutie expliciet wordt behandeld. Daardoor geeft het een redelijk beeld van de kosten die gemaakt worden in de pure $\lambda$-calculus. De Bruijn-indices zorgen ervoor dat we niet hoeven te herbenoemen.

In de tweede paragraaf zal ik een beeld geven van hoe het systeem uitgebreid met sharing-operator zich gedraagt in vergelijking met combinatoren en supercombinatoren. Ik zal aan de hand van een voorbeeld laten zien dat het systeem

optimaal werkt voor dat voorbeeld, terwijl de andere systemen dat niet doen. Het bewijs dat het systeem met sharing-operatoren in het algemeen correct en optimaal werkt, staat in [9] beschreven.

## 7.1 $\lambda x$-calculus

Zoals gezegd vergelijk ik in deze paragraaf ons systeem met de $\lambda x$-calculus [1]. Uit deze vergelijking is geen significant efficiënter systeem gekomen. Er zijn termen te verzinnen waar $\lambda x$ op wint en er zijn termen te verzinnen waar ons systeem beter op presteert. Een statistisch onderzoek zou hier meer helderheid kunnen verschaffen, maar zo'n onderzoek ligt buiten het bestek van deze scriptie. Ik zal het hier dus laten bij enkele algemene beschouwingen aan de hand van twee voorbeelden. Het eerste voorbeeld is $IS$ en het tweede $SI$, waarbij $I$ en $S$ staan voor de gelijknamige combinatoren (zie Paragraaf 7.2.1). Eerst zal ik een schatting maken van de kosten die bij elke herschrijfstap gemaakt worden.

### 7.1.1 Kosten

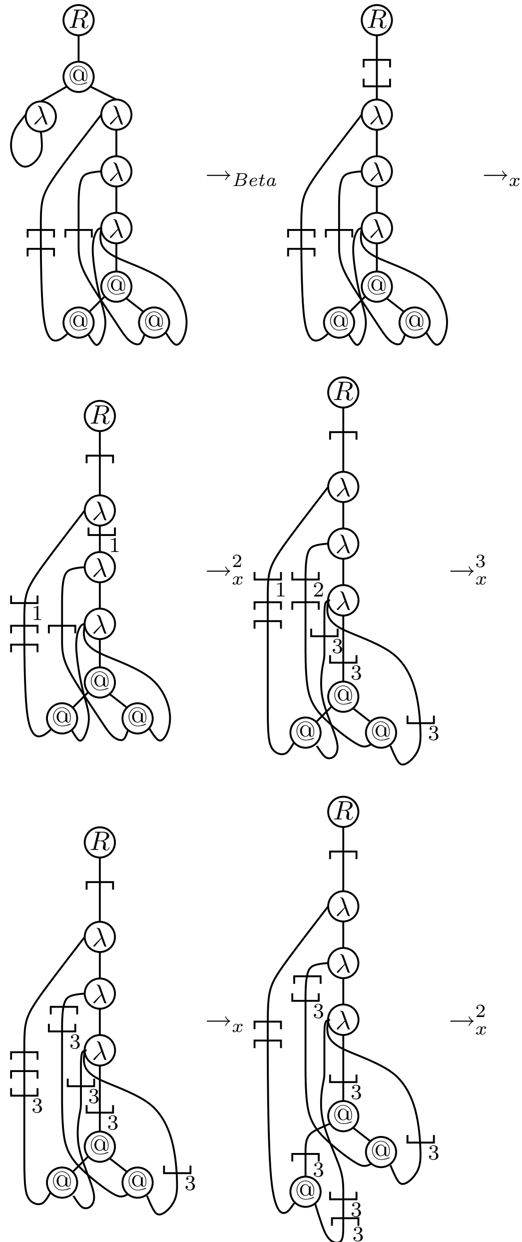
Als we voor een systeem de kosten per herschrijfstap willen bepalen, kunnen we de stappen ontleden in basisstapjes, waarvan de kosten gelijk zijn aan 1. In ons systeem zijn alle stappen lokaal, dat betekent dat ze allemaal erg eenvoudig zijn, omdat we slechts enkele knopen hoeven te onderzoeken voordat we een stap kunnen zetten. Deze stappen hoeven we dus niet verder te ontleden. De enige uitzondering is de *Beta*-stap, waar het kan voorkomen dat we het argument moeten kopiëren. De grootte van het argument en het aantal kopieën bepaalt dan hoeveel stapjes er nodig zijn om de stap te zetten. Omdat we straks echter een sharing operator hebben die dit probleem oplost, zullen we er in onze voorbeelden geen aandacht aan besteden.

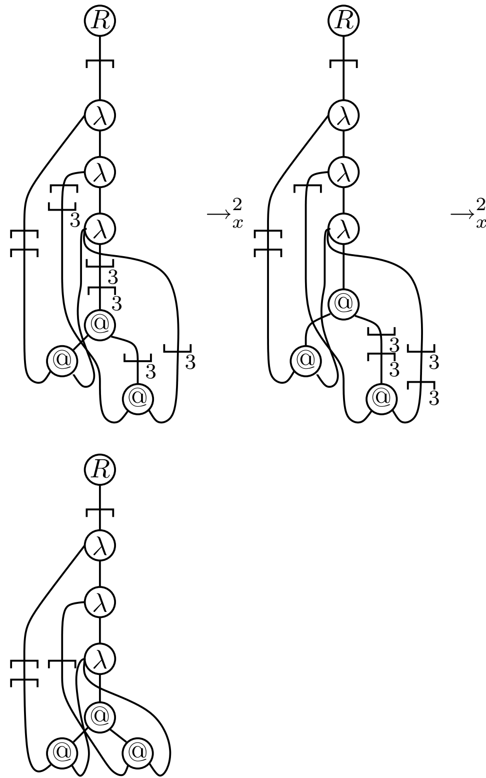
In het geval van $\lambda x$ zijn alle stappen ook zeer basaal. We zullen deze dan ook niet verder hoeven te ontleden. De enige uitzondering hierop is de volgende regel:

$$(MN)[x := P] \to M[x := P]N[x := P]$$

Hier wordt de substitutie, $[x := P]$, gekopiëerd. Hiervoor moet het argument $P$ gekopieerd worden. De kosten die hiervoor nodig zijn stellen we gelijk aan $P$.

**Voorbeeld 20** **($IS$)** *In ons systeem wordt $IS$ als volgt gereduceerd:*





*In de $\lambda x$-calculus wordt deze term als volgt gereduceerd:*

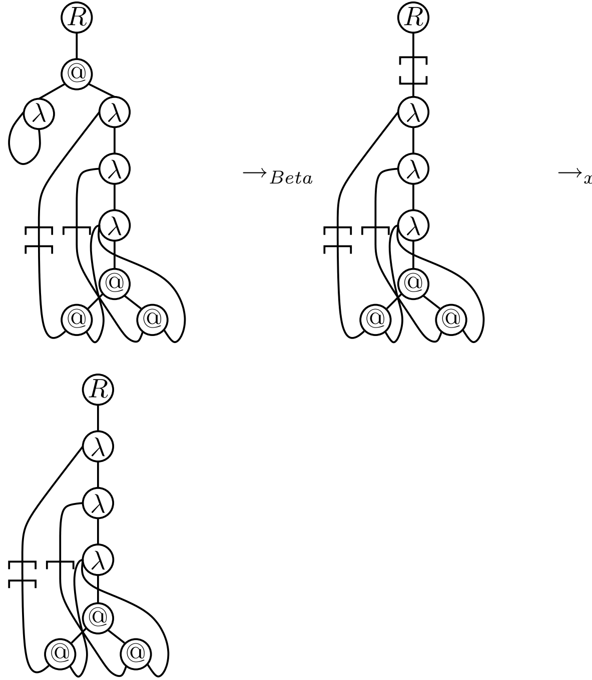
$$(\lambda 0)(\lambda\lambda\lambda(s(s(0))0)(s(0)0)) \to_\beta 0[\lambda\lambda\lambda(s(s(0))0)(s(0)0))]) \to$$

$$\lambda\lambda\lambda(s(s(0))0)(s(0)0))$$

We zien aan dit eerste voorbeeld dat de $\lambda x$-calculus veel sneller klaar is dan ons systeem. Ons systeem heeft één stap nodig voor de $\beta$-redex, maar heeft daarna nog 13 stappen nodig om tot de normaalvorm te komen. Na de *Beta*-stap is de graaf al gereduceerd tot een grafische representatie van de juiste term. De andere stappen zijn semantisch gezien overbodig. Als we ons systeem efficiënter zouden willen maken, zouden we twee dingen kunnen proberen. We zouden de volgende herschrijfregel kunnen toevoegen:

$\to_x$

Ons voorbeeld zou dan als volgt gaan:




Deze regel biedt echter niet in alle gevallen uitkomst. Bijvoorbeeld bij de vertaling van de term $KS$ zou deze regel niet werken. Na de *Beta*-redex zien we dat we deze herschrijfregel niet kunnen toepassen en nog steeds de 13 'nutteloze' stappen moeten zetten. Voor deze situatie zouden we dan weer een andere regel kunnen verzinnen. Hoewel het toevoegen van dit soort efficiëntie regels interessante resultaten kan opleveren, maken deze regels het systeem wel ingewikkelder. We zien bijvoorbeeld dat het systeem niet meer confluent is. In bovengenoemd voorbeeld zouden twee verschillende normaalvormen worden bereikt. Deze twee normaalvormen zijn wel representaties van dezelfde term. Dus dat lijkt op het eerste gezicht niet echt een probleem. We zullen het toevoegen van deze regels hier verder echter niet onderzoeken.

Een andere oplossing die mogelijk is, is een onderscheid maken tussen essentiële stappen en niet-essentiële stappen. We zouden dan enkel $x$-stappen kunnen zetten die nodig zijn voor het vrij maken van *Beta*-redexen. We weten dat verborgen redexen vrij kunnen worden gemaakt, door de knopen die tussen de $\lambda$- en de @-knoop liggen, uit de weg te ruimen door middel van herschrijfstappen. Als we enkel de herschrijfstappen willen zetten die nodig zijn voor het vrij maken van verborgen redexen, dan moeten we ervoor zorgen dat we enkel tussen @- en $\lambda$-knopen herschrijven. Dit zouden we als volgt kunnen bewerkstelligen:

- Zoek de graaf vanaf de wortel af tot de eerste @-knoop gevonden is.

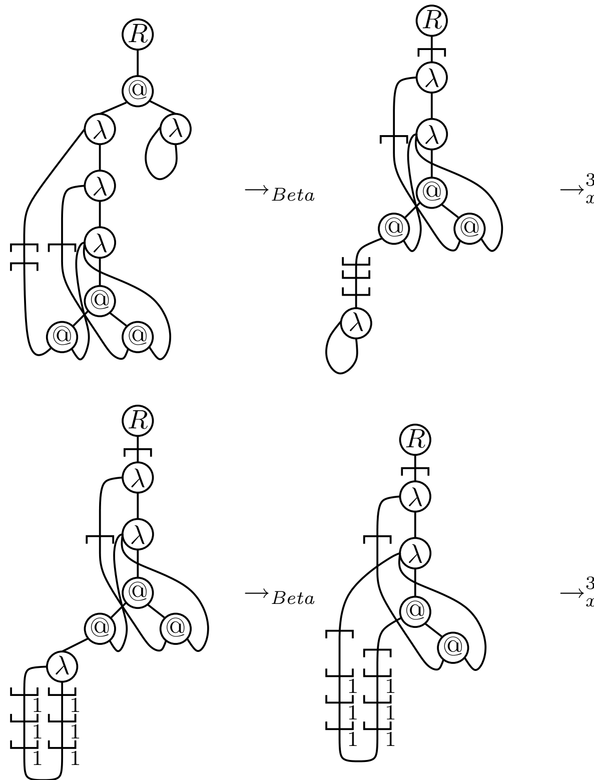
- Zoek vanaf de linkerpoort van de @-knoop naar een $\lambda$-knoop. Stop als

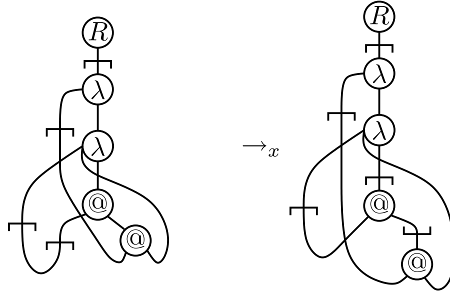
er een @-knoop of een $\lambda$-knoop vanaf de variabelenpoort bereikt wordt. Start opnieuw, maar sla de @-knopen over die je al gehad hebt.

- In het geval dat er een $\lambda$-knoop van de bovenkant bereikt wordt, start de procedure uit Lemma 2.
- Als de *Beta*-redex vrij is, wordt de bijbehorende *Beta*-stap gezet.

Als we deze optimalisatie doorvoeren dan is ons voorbeeld in één *Beta*-stap tot de normaalvorm gereduceerd. Na één stap zitten er geen verborgen redexen meer in de graaf en worden er dus ook geen extra $x$-stappen meer gezet. Voor deze optimalisatie is wel een slimmer herschrijfsysteem nodig. Bijvoorbeeld eentje die niet steeds opnieuw van bovenaf naar applicatie-knopen zoekt, maar bijhoudt waar hij gebleven is. In de praktijk zijn er echter genoeg voorbeelden van dit soort herschrijfsystemen.

**Voorbeeld 21 (*SI*)** *In ons systeem wordt SI als volgt gereduceerd:*






*In $\lambda x$ gaat het als volgt:*

$$(\lambda\lambda\lambda(s(s(0))0)(s(0)0))(\lambda 0) \;\to_\beta\; \lambda\lambda(s(s(0))0)(s(0)0)[\lambda 0] \to$$

$$\lambda, \lambda(s(s(0))0)(s(0)0)[\lambda 0]^1 \to \lambda\lambda, (s(s(0))0)(s(0)0)[\lambda 0]^2 \to_2$$

$$\lambda\lambda(s(s(0))0)[\lambda 0]^2(s(0)0)[\lambda 0]^2 \to_4 \lambda\lambda(s(s(0))[\lambda 0]^2 0[\lambda 0]^2)(s(0)[\lambda 0]^2 0[\lambda 0]^2) \to_4$$

$$\lambda\lambda(s(s(0)[\lambda 0]^1)0)(s(0[\lambda 0]^1)0) \to_2 \lambda\lambda(s(s(0[\lambda 0]))0)(s(0)0) \to$$

$$\lambda\lambda((\lambda 0)0)(s(0)0) \;\to_\beta\; \lambda\lambda(0[0])(s(0)0) \to$$

$$\lambda\lambda(0)(s(0)0)$$

In dit voorbeeld zien we dat ons systeem de normaalvorm sneller bereikt. Terwijl bij ons de beta-redexen in één stap worden herschreven, heeft $\lambda x$ in totaal 18 expliciete substitutiestappen nodig om beide redexen te herschrijven. In ons systeem moeten nog wel enkele $x$-stappen gezet worden. In dit geval zijn dat er 7. In de laatste herschrijfstap zien we echter een nieuw fenomeen verschijnen. Na het zetten van een *Beta*-stap zagen we al dat de nieuw gecreëerde ⅄-knoop het argument in kan en daar voor extra herschrijfstappen kan zorgen (Voorbeeld 20). Nu zien we echter dat na het zetten van een *Beta*-stap in een omgeving, de nieuw gecreëerde open-scopeknoop de omgeving in kan en daar voor extra herschrijfstappen kan zorgen. We zien hier dat de open-scopeknoop en zijn duplicaat na één stap worden tegengehouden, maar als de omgeving anders gevormd zou zijn, kan zo'n knoop de hele omgeving doorreizen. Het aantal stappen dat zo ontstaat is echter lineair in de grootte van de omgeving, een scopeknoop kan namelijk nooit meer dan één keer over een knoop geduwd worden.

### 7.1.2 Wat is efficiënter

Uit deze twee voorbeelden kunnen we enkele conclusies trekken. Doordat we werken met backpointers, wordt de *Beta*-stap relatief goedkoop. Alleen als we het argument moeten dupliceren, maken we meer kosten. In $\lambda x$ gaan er veel kosten zitten in de $\beta$-stap, omdat de hele body expliciet wordt afgezocht. Ons systeem verliest aan efficiëntie doordat er veel $x$-stappen gezet kunnen worden. Hierover kunnen we echter wel wat observaties doen:

Als we een term vertalen zullen er in eerste instantie geen $x$-stappen worden gedaan, omdat alle scope-informatie tegen de variabelenpoorten van de $\lambda$-knopen aanzit. Steeds als er een *Beta*-redex gedaan wordt, ontstaan er twee knopen die door de graaf heen kunnen reizen. De open-scopeknoop gaat de context van de *Beta*-redex in en de $\reflectbox{$\lambda$}$-knoop gaat het argument in. Samen kunnen ze dus de hele graaf door. Soms moeten er veel herschrijfstappen worden gezet voordat deze haakjes niet meer verder kunnen. Een knoop zal nooit meer dan één keer langs een andere knoop gaan, dus het aantal herschrijfstappen dat kan worden gezet is nooit groter dan het aantal knopen van de graaf, plus enkele stappen om de knoop en zijn duplicaten te verwijderen. Na het zetten van een *Beta*-stap, zit er altijd minstens één $\reflectbox{$\lambda$}$-knoop boven het gesubstitueerde argument. Deze kan nu het argument in en zorgt zo ook voor mogelijke herschrijfstappen. Dit aantal is echter nooit groter dan de grootte van het argument. Het aantal $x$-stappen dat in totaal kan worden gezet na het doen van een *Beta*-stap is steeds lineair in de grootte van de graaf.

In de voorbeelden die ik heb behandeld kunnen de scope-informatie-knopen bijna altijd het maximale aantal stappen zetten. Dit is meestal niet het geval. Meestal worden haakjes tegengehouden door knopen die ze van de verkeerde kant bereiken, zoals na de laatste stap in ons tweede voorbeeld. Interactie is dan niet mogelijk en hierdoor wordt het aantal mogelijke herschrijfstappen beperkt.

De conclusie die we dus kunnen trekken is dat beide systemen met hun eigen problemen zitten. Het ligt aan de vorm van de term welk systeem het meeste last van zijn problemen heeft. Het beste zouden we kunnen werken met een graafherschrijfsysteem met backpointers, zonder expliciete scope informatie. Zo zouden we *Beta*-stappen snel kunnen zetten en zouden we geen last hebben van $x$-stappen. Zo'n systeem is echter niet optimaal en kan waarschijnlijk ook niet eenvoudig worden uitgebreid tot een systeem dat dat wel is. Ons systeem kan wel eenvoudig worden uitgebreid tot een optimaal systeem (zie [9]).

## 7.2 Optimaliteit

Zoals we zagen in de vorige paragraaf is een probleem voor de $\lambda$-calculus dat de $\beta$-stap erg duur is. Er zijn verschillende manieren om met dit probleem om te gaan. Eén manier is om te werken met een sharing-operator, zodat een $\beta$-redex in ieder geval niet wordt verdubbeld en hij twee of meer keer moet worden uitgevoerd. Een andere methode is die van de (super)combinatoren. Bij deze methode wordt de $\beta$-stap geheel geëlimineerd en vervangen door een reeks simpelere stappen. In deze paragraaf zal ik beide methoden bespreken en aan de hand van een voorbeeld vergelijken.

### 7.2.1 Combinatoren

Met behulp van combinatoren kunnen we een $\lambda$-term herschrijven tot een term die enkel bestaat uit applicaties van combinatoren en variabelen. Voor iedere combinator wordt een simpele herschrijfregel gedefinieerd en met behulp van

die herschrijfregels kunnen we de term reduceren. De drie combinatoren die we hiervoor kunnen gebruiken zijn de volgende:

$I = \lambda x.x$, $K = \lambda xy.x$ en $S = \lambda xyz.xz(yz)$

We kunnen iedere $\lambda$-term herschrijven naar een applicatieve vorm van deze drie combinatoren met de volgende inductieve herschrijfregels:

$$[x]_{cl} = x$$

$$[(MN)]_{cl} = [M]_{cl}[N]_{cl}$$

$$[\lambda x.M]_{cl} = \lambda^* x.[M]_{cl}$$

Waarbij:

$\lambda^* x.x = I$
$\lambda^* x.M = KM$ als $x$ niet vrij voorkomt in $M$
$\lambda^* x.(MN) = S(\lambda^* x.M)(\lambda^* x.N)$

Bij iedere combinator definiëren we een herschrijfregel, namelijk:

$IM = M$
$KMN = M$
$SMNP = MP(NP)$

De operaties die deze combinatoren uitvoeren zijn de volgende. $I$ geeft zijn argument onveranderd terug, $K$ gooit een argument weg en $S$ distribueert een argument over twee anderen.

We kunnen combinator $I$ zelfs uitdrukken in $K$ en $S$, namelijk:

$$I = SKK$$

Zodoende kunnen we een $\lambda$-term dus herschrijven naar een combinator-term die enkel bestaat uit applicaties van $K$, $S$ en variabelen.

Voordeel is dus dat het herschrijven van de term veel simpeler is geworden. De $\beta$-stap wordt nu vervangen door een reeks eerste orde herschrijfregels die expliciet zeggen hoe de substitutie werkt. Een nadeel van de combinatoren is dat de vertaling van een $\lambda$-term naar combinatoren meestal een exponentieel groot resultaat tot gevolg heeft. De Church-numeral 2 wordt bijvoorbeeld:

$$\lambda xy.x(xy) = S(S(KS)(S(KK)I)(S(S(KS)(S(KK)I)(KI))$$

Bijkomend nadeel is dat het moeilijker is om te zien wat deze code nu eigenlijk doet en dat de structuur van de oorspronkelijke $\lambda$-term niet meer zichtbaar is. De read-back is een kostbare zaak. Als we het resultaat of een tussenresultaat als $\lambda$-term willen weergeven, dan moeten we alle combinatoren vervangen door de bijpassende $\lambda$-term en de term die we dan krijgen herschrijven.

De groei van de code kunnen we inperken door met meerdere combinatoren te werken. De volgende twee combinatoren kunnen worden toegevoegd:

$B = \lambda xyz.x(yz)$ en $C = \lambda xyz.(xz)y$

We zien dat deze twee combinatoren allebei de helft van het werk van $S$ doen. $B$ distribueert het derde element alleen naar de rechterkant en $C$ alleen naar links. De vertaling van $\lambda$-termen moet met de volgende twee regels worden uitgebreid:

$$\lambda^*x.(MN) = BM(\lambda^*x.N) \text{ Als } x \text{ niet vrij voorkomt in } M$$
$$\lambda^*x.(MN) = C(\lambda^*x.M)N \text{ Als } x \text{ niet vrij voorkomt in } N$$

De toevoeging van deze twee combinatoren scheelt significant in de grootte van de termen. Church-numeral 2 wordt bijvoorbeeld de helft kleiner:

$$\lambda xy.x(xy) = S(BBI)(C(BBI)I)$$

### 7.2.2 Supercombinatoren

Het uitbreiden van de verzameling combinatoren blijkt een gunstig effect te hebben op de groei van de term. Als we de combinatoren flexibel maken kunnen we per $\lambda$-term een ideale set van combinatoren genereren. Deze flexibele combinatoren noemen we supercombinatoren [6]. Supercombinatoren moeten aan de volgende voorwaarden voldoen:

**Definitie 58 (Supercombinatoren)**

- *Een supercombinator is een $\lambda$-term van de volgende vorm: $\lambda v_1....v_n.M$.*
- *De supercombinator mag geen vrije variabelen bevatten.*
- *De body van de supercombinator, $M$, is een applicatieve term. Dat wil zeggen dat hij enkel is opgebouwd door middel van applicaties.*

Iedere $\lambda$ term kan worden vertaald naar een supercombinator. De volgende manier is de meest simpele, maar heeft niet zo'n efficiënt resultaat:

**Definitie 59 (Van $\lambda$-term naar supercombinator)** *Neem een willekeurige $\lambda$-term, $\lambda v.M$:*

- *Begin met het omschrijven van de body, $M$, tot een applicatief geheel, door deze procedure recursief toe te passen. Zodoende krijgen we $\lambda v.N$.*
- *Bindt alle vrije variabelen door extra $\lambda$'s te plaatsen: $\lambda pq....rv.N$. We hebben nu de supercombinator $S = \lambda pq....rv.N$.*
- *Definieer bij de supercombinator $S = \lambda pq....rv.N$, de volgende herschrijfregel: $Spq...rv = N$.*
- *Vervang de oorspronkelijke $\lambda$-term, $\lambda v.M$, door $Spq...r$.*

We zien dat op deze manier de $\lambda$'s in de body verdwijnen en er een applicatieve term overblijft. Deze term is te herschrijven met de regels die we bij de supercombinatoren hebben gedefinieerd. Merk op dat een supercombinatorregel

enkel kan worden toegepast als de supercombinator op voldoende argumenten wordt toegepast.

Ik zei al dat het resultaat dat we uit bovenstaand algoritme krijgen, niet erg efficiënt is. Dit algoritme is namelijk gebaseerd op de vrije variabelen in de term. De vrije variabelen zijn de minimale vrije expressies en het is efficiënter om de maximale vrije expressies te binden. Als we bijvoorbeeld de term $\lambda x\ zyx$ willen vertalen, dan zijn $z$ en $y$ minimale vrije expressies, ten opzichte van $\lambda x$. In $\lambda x\ zyx$ is $zy$ maximale vrije expressie ten opzichte van $\lambda x$. Een maximale vrije expressie is dus een zo groot mogelijke subterm waar een bepaalde variabele (in dit geval $x$) niet in voorkomt. Als we de term vertalen met bovenstaand algoritme, krijgen we $Szyx \rightarrow zyx$. Dit is een supercombinator die drie argumenten nodig heeft. Als we de maximale vrije expressies gebruiken in de vertaling krijgen we de supercombinator $Spx \rightarrow px$ (de $\lambda$-term wordt dan vervangen door $S(zy)$). Dit is een supercombinator met slechts één argument. Een supercombinator met minder argumenten kan makkelijker worden toegepast. Het blijkt ook dat een vertaalalgoritme dat gebruik maakt van maximale vrije expressies een *fully lazy* resultaat heeft.

Naast het kiezen van maximale- in plaats van minimale vrije expressies, zijn er nog meerdere optimalisaties mogelijk. Zo kunnen supercombinatoren ook grafisch worden gerepresenteerd en dan kan ook *sharing* worden geïmplementeerd. Deze sharing is vergelijkbaar met de sharing in [12].

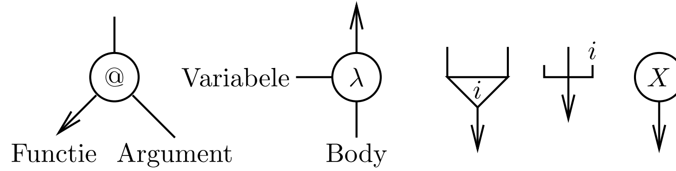
Als we bij het kiezen van de supercombinatoren rekening moeten houden met al deze optimalisaties, dan wordt het vertalen wel vrij arbeidsintensief. Daarom laten we het vertalen doen door een compiler. Een slimme compiler zal een $\lambda$-term omschrijven naar een applicatieve term met zo optimaal mogelijk gekozen supercombinatoren. Het herschrijven is daarna een relatief eenvoudige zaak omdat we alleen maar met de expliciete herschrijfregels van de supercombinatoren te maken hebben en deze zijn simpel toe te passen.

Het voordeel van supercombinatoren boven de gewone combinatoren is dat de term niet exponentieel groeit en in veel gevallen zelfs kleiner is dan de oorspronkelijke term. Een nadeel blijft dat de structuur van de oorspronkelijke term niet meer zichtbaar is. Als we een $\lambda$-term als resultaat willen, moeten we een vrij intensief vertaalproces toepassen. Voor toepassingen waar we $\lambda$-termen als tussenresultaten willen hebben, zijn supercombinatoren wat minder geschikt.

### 7.2.3 Sharing

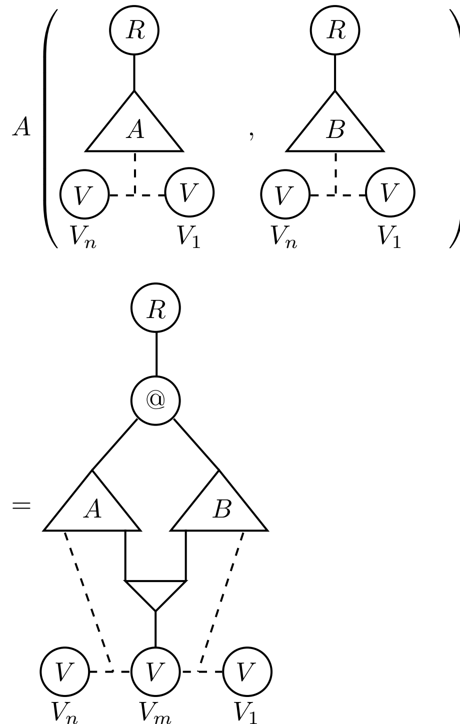
In [10] wordt het interactienet geïntroduceerd waar deze scriptie een voorstudie van is. De details van dit interactienet liggen buiten het bestek van deze scriptie, vandaar dat ik hier slechts een globale samenvatting geef. Het interactienet bestaat uit de grafen die we tot nu toe hebben besproken, uitgebreid met een extra knoop, de sharing-operator of *fan*. De signatuur van ons interactienet is

als volgt:

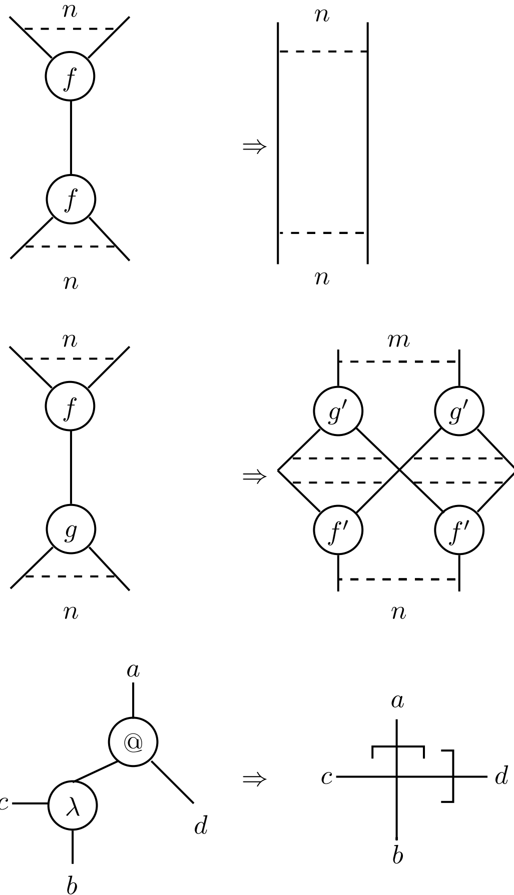



Waarbij een pijl staat voor een interactiepoort, $i$ voor een index tussen 0 en $n$, waarbij een index van 0 wordt weggelaten. De applicatie-knoop, $\lambda$-knoop, $X$-knoop en de verschillende vormen van de scopeknoop, werken zoals in de andere hoofdstukken beschreven. Merk hierbij wel op dat de variabelenpoort van den $\lambda$-knoop nog maar één verbinding heeft. De werking van de sharing-knoop is in het algemeen dat hij de knopen waarmee hij interactie heeft, verdubbelt.

Als we ons vertaalalgoritme uit Hoofdstuk 3 willen uitbreiden zodat het resultaat ook sharing-knopen bevat, dan zouden we de applicatieregel zo moeten aanpassen dat als er meerdere takken naar een $V$-knoop dreigen te gaan, deze worden geshared tot één tak. Dat zou er dan ongeveer zo uitzien:



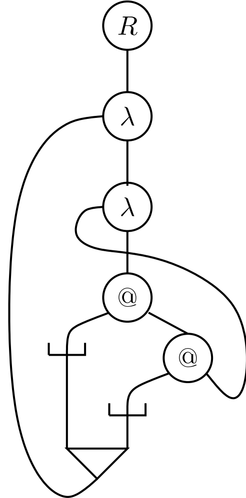
De interactie die kan plaatsvinden in het net is vast te leggen in twee interactieschema's en een expliciete regel:



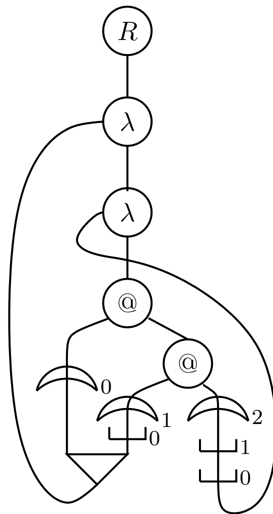

Waarbij $f'$ en $g'$ identiek of een update zijn van $f$ en $g$. Een update wil zeggen dat de index met 1 is opgehoogd. Een update vindt plaats als de knoop die gepasseerd wordt een $\lambda$-knoop is of een scopeknoop met een lagere index.

Ons systeem lijkt veel op dat van Asperti en Guerrini ([2]), behalve dan dat ons net een controle-knoop minder bevat. Vergelijk twee vertalingen van de

Church-numeral 2:




Ten opzichte van 2 bij Asperti en Guerrini.:




Omdat ons net minder knopen bevat is het te verwachten dat we in het algemeen ook minder herschrijfstappen nodig hebben.

Ons net is optimaal volgens Definitie 1. $\beta$-redexen worden dus niet verdubbeld voordat ze worden uitgevoerd. Het bewijs hiervoor staat in [9]. Ik zal het in de volgende paragraaf slechts aannemelijk maken door een voorbeeld uit te werken.

### 7.2.4 Combinatoren zijn niet optimaal

Laten we het volgende voorbeeld bekijken:

$22Ia$

Waarbij 2 staat voor Church-numeral 2, $I$ voor de gelijknamige combinator en $a$ een willekeurige subterm. In de $\lambda$-calculus reduceert deze optimaal in negen stappen:

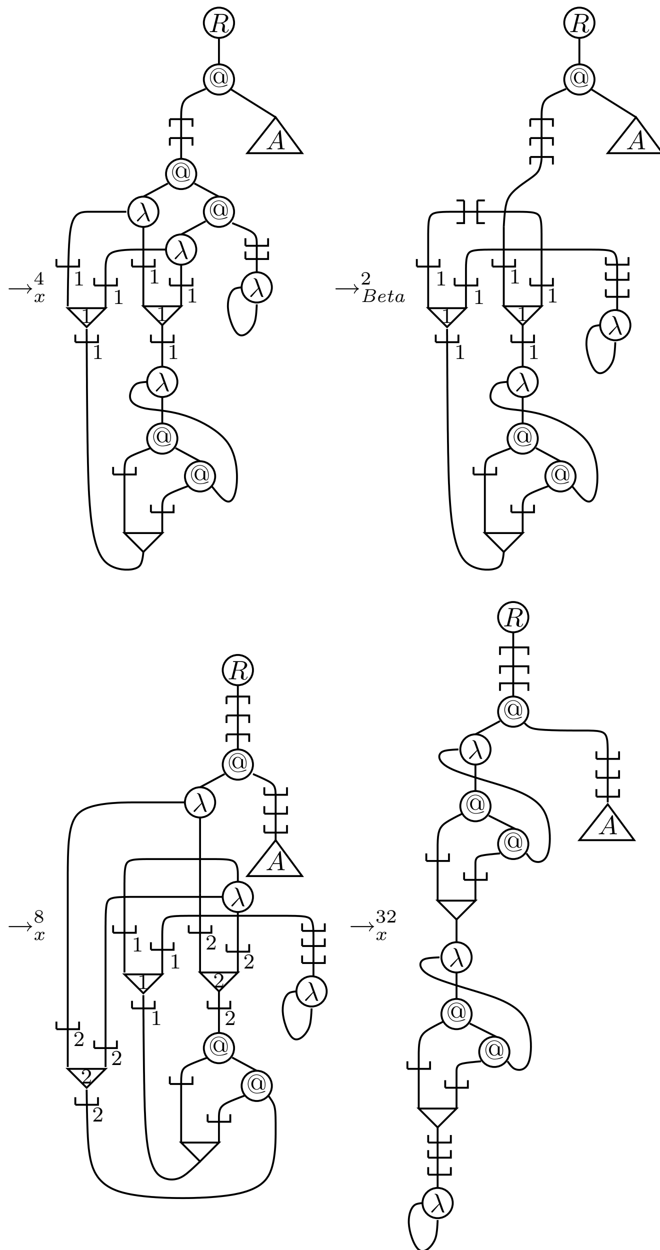
$$(\lambda xy.x(xy))2Ia \ \to_\beta \ (\lambda y.2(2y))Ia \ \to_\beta \ 2((\lambda vw.v(vw))I)a$$

$$\to_\beta 2(\lambda w.(\lambda p.p)((\lambda p.p)w))a \ \to_\beta \ 2(\lambda w.(\lambda p.p)w)a \ \to_\beta \ (\lambda qr.q(qr))(\lambda w.w)a$$

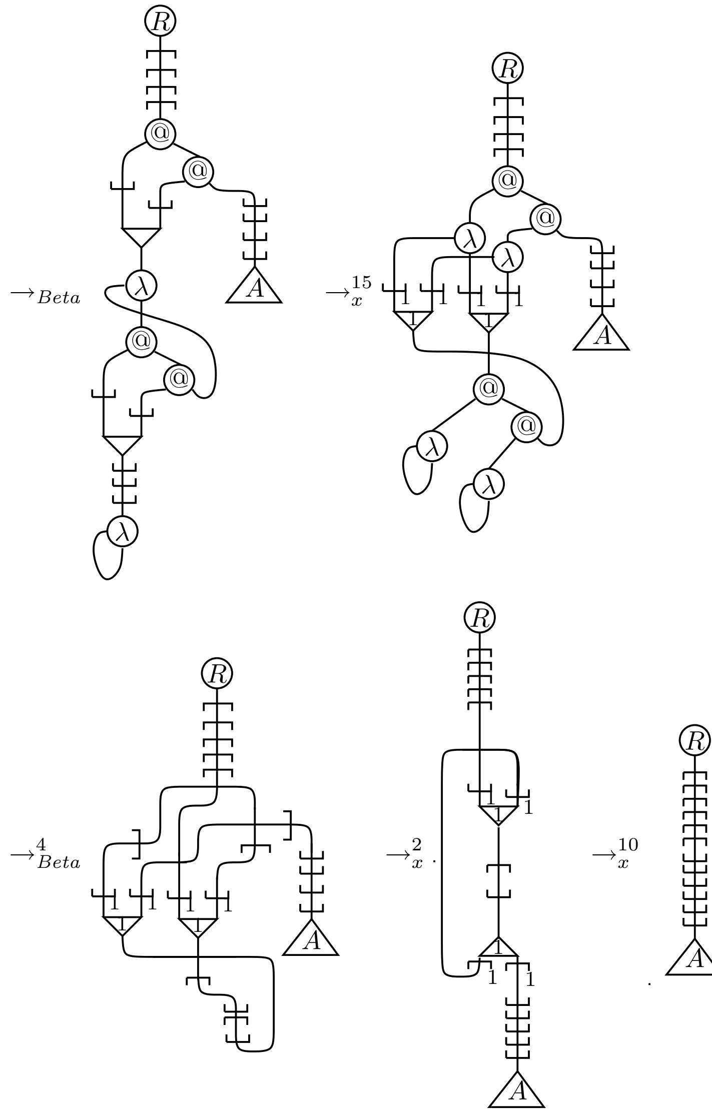
$$\to_\beta (\lambda r.(\lambda w.w)((\lambda w.w)r))a \ \to_\beta \ (\lambda w.w)((\lambda w.w)a) \ \to_\beta \ (\lambda w.w)a \ \to_\beta \ a$$

Merk op dat de *normal order reduction strategy* hier geen optimaal resultaat zou hebben. Die strategie had ons 12 stappen gekost.

In ons systeem ziet deze reductie er als volgt uit:



$\rightarrow^{4}_{x}$
$\rightarrow^{2}_{Beta}$
$\rightarrow^{8}_{x}$
$\rightarrow^{32}_{x}$




We hebben bij deze reductie in totaal 81 stappen nodig. Daarvan zijn er echter maar 9 *Beta*-stappen. 19 stappen hebben te maken met het *unsharen* van een gedeelte van de graaf en 53 stappen hebben te maken met de expliciete scope-operatoren. Het systeem is dus optimaal in het aantal $\beta$-stappen.

Bij combinatoren zijn er geen $\beta$-stappen meer te onderscheiden. Toch weten we zeker dat er dubbel werk wordt gedaan. Bijvoorbeeld de term $M = 2I$, reduceert naar $S(KI)(S(KI)I)$ (of $BI(BII)$). In bovengenoemd voorbeeld reduceert deze term naar $I$. Mocht term $M$ geshared worden, zoals in $2(2I)a$, dan moet de term $2I$ twee maal op een argument worden toegepast. Met deze kleine term is dit probleem nog te overzien, maar in het geval van $n(2I)a$ (waarbij $n$ de Church-numeral $n$) groeit het probleem naarmate $n$ groter wordt.

Ook bij supercombinatoren kan de verdubbeling van werk niet worden voorkomen. Neem bijvoorbeeld weer de term $22Ia$. Merk op dat 2 en $I$ allebei supercombinatoren zijn. De herschrijfregels zijn als volgt:

$$2AB = A(AB)$$
$$IA = A$$

De reductie ziet er als volgt uit:

$$22Ia = 2(2I)a = 2I(2I)a$$

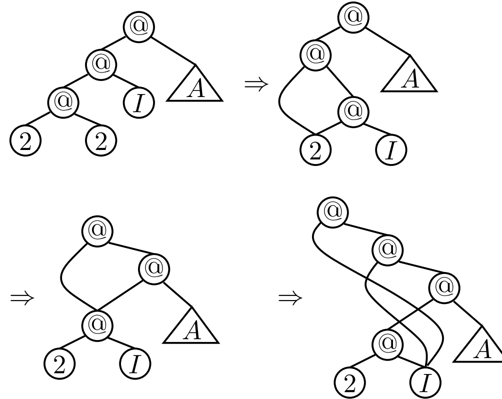
Ten eerste zien we dat de term $2I$ niet geshared is. Ten tweede zou in de $\lambda$-calculus de term $2I$ reduceren tot $I$. De optimale reductie zou er zo uitzien:

$$2I(2I)a = I(2I)a = 2Ia = Ia = a$$

Dat gebeurt niet bij de supercombinatoren, want in de subterm $2I$ heeft 2 te weinig argumenten. De reductie gaat dus verder als volgt:

$$2I(2I)a = I(I(2I))a = I(2I)a = 2Ia = I(Ia) = Ia = a$$

Ook als we de supercombinatoren in graafnotatie met sharing zouden toepassen komen we in de problemen:




We zien hier dat na twee stappen de subterm $2I$ geshared is. Deze kan niet worden uitgerekend omdat 2 te weinig argumenten heeft. Daardoor moet deze term twee maal apart worden uitgerekend (in de derde stap wordt de eerste instantie van de gesharede subterm uitgerekend). Dit probleem wordt groter naarmate de term $2I$ vaker geshared wordt.

### 7.2.5 Wat is beter?

Als we supercombinatoren en ons systeem vergelijken, zien we dat ze allebei hun voor- en nadelen hebben. Supercombinatoren hebben over het algemeen weinig stappen nodig om termen te herschrijven. Ze zijn echter niet optimaal en in het geval dat er termen moeten worden gereduceerd waar vaak argumenten moeten

worden gekopieerd, is dubbel werk moeilijk te voorkomen. Een ander nadeel is dat we in de supercombinator vertaling niet meer de oorspronkelijke code terugzien. Ons systeem is optimaal. Dit kost echter veel boekhoudstappen om bij te houden welke *fans* bij elkaar horen. We hebben echter wel gezien dat het aantal stappen steeds lineair in de grootte van de graaf is. Alle stappen zijn ook lokaal en dus simpel. Wat betreft de sharing-stappen: de supercombinatoren zullen ook kosten moeten maken om termen te kopiëren. Deze kosten heb ik hier niet expliciet genoemd. We kunnen concluderen dat in termen waar veel sharing nodig is, ons systeem waarschijnlijk beter zal presteren. Daarnaast is in ons systeem nog redelijk goed te zien wat de oorspronkelijke $\lambda$-code is en de read-back is ook redelijk makkelijk te verkrijgen.

Uiteindelijk zal het vooral aan de toepassing liggen welke aanpak het beste is.

# 8 Conclusie

We hebben een graafherschrijfsysteem gedefinieerd voor de $\lambda$-calculus. Dit herschrijfsysteem bevat informatie over het einde van de scope van de $\lambda$-operatoren. Dit hebben we als volgt gedaan:

- We begonnen met het definiëren van een vertaalfunctie die termen naar een grafische representatie vertaalt. Vervolgens hebben we bewezen dat twee $\alpha$-equivalente termen worden vertaald naar dezelfde graafrepresentatie.
- Daarna hebben we een machine gedefinieerd die grafen weer kan terugvertalen naar termen. We hebben bewezen dat als we een term naar zijn grafische representatie vertalen en weer terug, het resultaat $\alpha$-equivalent is aan de term. Zodoende hebben we de correspondentie tussen termen en grafen vastgelegd.
- We hebben herschrijfregels voor grafen gedefinieerd waarmee we deze kunnen herschrijven. Deze regels zijn lokaal.
- We hebben bewezen dat we met de herschrijfregels de $\beta$-reductie in de termen kunnen nabootsen. Ons graafherschrijfsysteem is dus een correcte representatie van de $\lambda$-calculus.
- Uiteindelijk hebben we bekeken hoe efficiënt ons systeem is.

We hebben dus een lokale versie gegeven van de $\reflectbox{$\lambda$}$-calculus. We hebben gezien dat in een lokale versie van de $\reflectbox{$\lambda$}$-calculus een re-open-scopeknoop ontstaat. We hebben gezien dat de herschrijfregels voor dit systeem simpel zijn. Nadeel is wel dat we vrij veel stappen nodig hebben om een graaf te reduceren. Dit komt vooral om dat we door de lokaliteit geen overzicht hebben over welke stappen nodig zijn voor het reduceren van een graaf. We hebben echter ook gezien dat het aantal stappen steeds lineair zal zijn in de grootte van de graaf.

## 8.1 Gerelateerd werk

In Hoofdstuk 7 hebben we laten zien dat dit systeem kan worden uitgebreid tot een optimaal herschrijfsysteem, door een shating-operator toe te voegen. De details hierover staan beschreven in [9]. Het zal nog moeten blijken hoe efficiënt dit uitgebreide herschrijfsysteem is en hoe we dit eventueel nog efficiënter kunnen krijgen.

# Referenties


[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[2] A. Asperti and S. Guerrini. *The Optimal Implementation of Functional Programming Languages*, volume 45 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.

[3] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.

[4] N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indagationes Mathematicae*, 34:381–392, 1972.

[5] D. Hendriks and V. van Oostrom. ʎ. In *CADE 19*, volume 2741 of *LNAI*, pages 136–150. Springer, 2003.

[6] R.J.M. Hughes. Supercombinators: A new implementation method for applicative languages. In *ACM Conference on LISP and Functional Programming*, 1982.

[7] Y. Lafont. Interaction nets. In *POPL 17*, pages 95–108. ACM Press, 1990.

[8] J.-J. Lévy. *Réductions correctes et optimales dans le λ-calcul*. Thèse de doctorat d'état, Université Paris VII, 1978.

[9] Kees-Jan van de Looij. Tba. Master's thesis, 2004. Forthcoming.

[10] Vincent van Oostrom, Kees-Jan van de Looij, and Marijn Zwitserlood. Lambdascope - another optimal implementation of the lambda-calculus. 2004.

[11] Universiteit Utrecht. *Studiegids CKI*. UU, 2004.

[12] C. Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus*. PhD thesis, 1971.