Inleiding tot de λ calculus

D. Vermeir B. Manderick W. De Meuter
S. Casteleyn

1 Inleiding

Alonzo Church en Stephen Kleene ontwierpen de pure λ-calculus[1] in de jaren '30 als een systeem dat kan gebruikt worden om het begrip "functie", en meerbepaald functie definitie, applicatie en recursie, abstract te bestuderen.

Hiermee wordt bedoeld dat de nadruk ligt op het functie**voorschrift** eerder dan een verzamelingtheoretische benadering, waar een functie bekeken wordt als een verzameling van geordende paren.

Bijvoorbeeld kan de functie

$$f(x) = 2 + x$$

over de gehele getallen bekeken worden d.m.v. haar voorschrift

$$2 + x$$

of als de verzameling

$$f = \{\ldots, (-1, 1), (0, 2), (1, 3), \ldots\}$$

waarbij het idee van het toepassen van het functievoorschrift naar de achtergrond wordt verbannen.

In λ-notatie[2] zou f geschreven worden als:

$$\lambda x.((+)2)x \tag{1}$$

Intuitief kan men dit lezen als: om de waarde van $f(x)$ te berekenen voor een zekere parameter x (dit laatste wordt aangeduid door de $\lambda x.$) gaat men als volgt te werk:

- Neem de functie + (optelling) en pas deze toe op het element 2. Dit levert een nieuwe functie (laten we ze $+_2$ noemen) op van één enkele parameter die als volgt kan beschreven worden:

$$+_2(y) = y + 2$$

[1] Er bestaan ook zgn. getypeerde varianten waarover we het in deze tekst niet zullen hebben. Pure λ-calculus is net als Scheme niet getypeerd.

[2] Merk op dat we het hebben over λ-notatie niet λ-calculus: de hier gegeven expressie is immers, zoals we later zullen zien, geen geldige λ-calculus expressie, maar geeft wel een idee van de gebruikte notatie.

Deze nieuwe functie $+_2$ heeft dus één parameter, en doet niets anders dan 2 optellen bij (de waarde van) deze parameter.

- In de uitdrukking (1) wordt $+_2$ aldus verder toegepast op de waarde x wat tot het uiteindelijk resultaat leidt.

Uit (1) valt nog op te maken dat men in λ-calculus blijkbaar geen naam moet geven aan een functie (f komt helemaal niet voor in het voorbeeld) en dat functies slechts één argument hebben. Door echter functies te definieren waarvan de functiewaarde voor een gegeven parameter weer een functie ($+_2$ in het voorbeeld) is, is het mogelijk om functies met meerdere parameter te simuleren ($+$ in het voorbeeld)[3].

Het blijkt dus al direkt dat de benadering van λ-calculus intiem verbonden is met het idee van functieëvaluatie als *berekening*, en dus ook met functievoorschrift als *programma*. Het hoeft dan ook niet te verwonderen dat λ-calculus diverse toepassingen kent binnen de informatica. Het is bijvoorbeeld de basis voor zogenaamde *functionele programmeertalen*, talen die de nadruk leggen op functie evaluatie i.p.v. data en staat veranderingen, zoals gebruikelijk bij imperatieve programmeertalen. Lisp, de voorganger van Scheme, is rechtstreeks gebaseerd op λ-calculus en is een voorbeeld van een (niet-pure) functionele programmeertaal. Later werden heel wat nieuwere functionele programmeertalen ontwikkeld, zoals ML, Miranda, Haskell en Gofer, die eigenlijk allen gebaseerd zijn op en uitbreidingen zijn van de originele λ-calculus. Ook het zogenaamde *call-by-name* ('lazy evaluation') mechanisme dat in sommige programmeertalen voorhanden is komt uit de λ-calculus. Er dient ook opgemerkt te worden dat de beïnvloeding in twee richtingen gaat vermits vele recente resultaten in de λ-calculus rechtstreeks geïnspireerd werden door problemen in de informatica.

Naast bovengenoemde invloed op de ontwikkeling van programmeertalen was de λ-calculus eveneens van groot belang voor de theorie van de "berekenbaarheid". Berekenbaarheid is de tak van de wiskunde en (theoretische) informatica die uitspraken doet over welke functies al dan niet berekenbaar zijn, of anders gezegd, welke (wiskundige) problemen oplosbaar zijn, en welke niet. Historisch gezien is het beslissen of twee λ-expressies equivalent zijn het eerste probleem waarvan werd aangetoond dat het onbeslisbaar is: het is onmogelijk een programma te schrijven dat, gegeven twee willekeurige λ-expressies bepaalt of beiden λ-expressies equivalent zijn of niet.

2 Basisbegrippen

λ-calculus gaat uit van twee fundamentele bewerkingen omtrent functies: aanroepen (applicatie) en aanmaken (abstractie). Met "aanroepen" bedoelen we hier het toepassen van het functievoorschrift op een parameter. Met "abstractie" maken we van een voorschrift een 'nieuwe' functie.

[3] Zie verder over "Currying".

2.1 Lambda Expressies

De volgende definitie definieert de verzameling der geldige uitdrukkingen (λ-expressies) in de λ-calculus.

Definitie 1 *Weze V een verzameling* **variabelen**. *De verzameling van λ-expressies Λ wordt als volgt gedefinieerd:*

- $V \subseteq \Lambda$
- *Als $M \in \Lambda$ en $N \in \Lambda$ dan is $(M)N \in \Lambda$ (***applicatie** *of* **toepassing**).
- *Als $x \in V$ en $M \in \Lambda$ dan is $\lambda x.M \in \Lambda$ (***abstractie**).
- *Niets anders zit in Λ*

Bovenstaande (constructieve) definitie definieert wat geldige λ-expressies zijn. Merk op dat hier nog geen enkele uitspraak wordt gedaan over de *betekenis* van de expressies; op dit moment maken we enkel uitspraken over wat syntactisch gezien geldige λ-expressies zijn. Voor een beter begrip geven we wel al intuitief aan wat de betekenis van de λ-expressies is; een formele definitie van wat we nu juist kunnen doen met deze λ-expressies, en dus wat de semantiek ervan is, komt later.

- De eerste regel zegt simpelweg dat een iedere variabele een λ-expressie is.
- De tweede regel zegt dat, gegeven twee geldige λ-expressies M en N, we een nieuwe geldige λ-expressie kunnen vormen door haakjes rond één te zetten en de andere erachter te plaatsen. Deze regel correspondeert met een functieaanroep:

  $$(functie)actuele\_parameter$$

  De actuele parameter (ook wel *argument*) van een functie is de parameter die een oproeper van de functie 'aan de functie geeft'.
- De derde regel zegt dat, gegeven een variabele en een geldige λ-expressie, we een nieuwe geldige λ-expressie kunnen vormen door een λ voor de variabele te zetten, gevolgd door een punt en de geldige λ-expressie. Deze regel beschrijft de vorm van een "functie" in λ-calculus:

  $$\lambda formele\_parameter\_variabele.functie\_voorschrift$$

  De formele parameter van een functie is de parameter zoals die in de *definitie* van de functie voorkomt.
- De vierde regel zorgt ervoor dat al wat niet gevormd kan worden door (opeenvolgende) toepassing van de vorige drie regels geen geldige λ-expressie kan zijn.

Intuïtief moet b.v. $(\lambda x.x)y$ gelezen worden als "pas de functie $\lambda x.x$ toe op de parameter y". Aan de functie $\lambda x.x$ kan men zien dat x de formele parameter is (de formele parameter is de variabele vóór het punt) terwijl de tweede x (in dit voorbeeld) het voorschrift geeft.

Merk op dat de pure λ-calculus geen constanten kent. We zullen later zien dat het toch mogelijk is om b.v. (gehele) getallen door middel van λ-expressies voor te stellen. Evenzo zullen we zien dat het mogelijk is om bepaalde arithmetische functies, programmeerconcepten en -constructies, zoals b.v. "+" en "−", booleans, if-then expressies, arithmetische functies, lijsten (allen niet aanwezig in λ-calculus) voor de stellen door middel van λ-expressies.

Merk ook op dat we het in deze tekst steeds (verkeerdelijk) over "functies" in λ-calculus hebben. Eigenlijk bevat λ-calculus helemaal geen functies, ze is, zoals eerder reeds vermeld, slechts een verzameling van goed gevormde expressies (volgens 1) waaraan wij mensen een bepaalde interpretatie kunnen geven.

Weze x en y variabelen $(x, y \in V)$[4], dan zijn volgende voorbeelden correcte λ-expressies.

Voorbeeld 1

$$x$$
$$\lambda x.x$$
$$\lambda x.\lambda y.(y)x$$
$$(\lambda y.(x)y)\lambda x.(u)x$$
$$\lambda x.\lambda y.x$$
$$\lambda f.\lambda x.(f)x$$
$$\lambda f.\lambda x.(f)(f)x$$

Merk op dat uit definitie 1 volgt dat 'functie-applicatie' (waarover we het later nog uitvoerig gaan hebben) van rechts naar links gebeurt. Stel b.v. dat M en N geldige λ-expressies zijn, en x een variabele, dan verkrijgt men b.v. het resultaat van

$$(P)(Q)x$$

door eerst Q toe te passen op x, en daarna P toe te passen op het resultaat (van (Q)x). M.a.w., $(P)(Q)x$ zou in "gewone" notatie geschreven worden als

$$P(Q(x))$$

Inderdaad kan men volgens de definitie $(P)(Q)x$ slechts opbouwen door eerst $(Q)x$ te vormen uit Q en x (regel 2 in Definitie 1), en dan weer een applicatie te vormen (opnieuw regel 2 in Definitie 1), waarbij $(Q)x$ de (actuele) parameter is.

[4]In het vervolg zullen we niet steeds expliciet vermelden wat de variabelen zijn; we gaan er in het vervolg vanuit dat letters gebruikt als variabelen in λ-expressies inderdaad als dusdanig gedefiniëerd werden

De normale orde kan gewijzigd worden door de applicaties anders op te bouwen: in

$$((P)Q)x$$

staat een λ-expressievan de vorm $(M)x$, waarbij $M = (P)Q$. We moeten dus eerst P toepassen op Q, waarna de resulterende λ-expressiewordt toegepast op x. Tenslotte kan nog opgemerkt worden dat

$$((P)(Q))x$$

geen geldige λ-expressie is vermits de applicatie (Q) geen argument heeft.

2.2 Currying

Uit Definitie 1 blijkt dat een 'functie' in λ-calculus slechts één parameter kan hebben. Men zou zich kunnen afvragen of dit geen beperking vormt indien men functies van meerdere variabelen (b.v. de optelfunctie voor gehele getallen) wil voorstellen. Het blijkt dat dit geen probleem is omdat we b.v. een functie van 2 variabelen steeds kunnen beschouwen als een functie van 1 variabele die zelf een functie van 1 variabele als resultaat geeft (herinner het voorbeeld van $+_2$ uit de inleiding; we gaan hier nu dieper op in).

Wanneer we een functie F voorstellen door een pijl

$$F : \mathit{domein} \to \mathit{codomein}$$

van het domein naar het codomein, waarbij het codomein zelf weer functies kan bevatten (m.a.w. bijvoorbeeld $(E \to F)$ stelt een verzameling van functies van E naar F voor) komt dit neer op het vervangen van

$$f : A \times B \to C$$

door

$$g : A \to (B \to C)$$

waarbij voor alle $a \in A, b \in B$ $g(a) = f_a$ zo gedefinieerd is dat $f_a(b) = f(a, b)$.

Toegepast op de optelling van twee (gehele) getallen komt dit neer op het definieren van een functie $+$ die een willekeurig getal a omzet in een functie $+_a$ die zelf weer gedefinieerd wordt door $+_a(x) = a + x$.

Het is duidelijk dat men op deze manier een functie van n variabelen kan omzetten naar een van een functie van 1 variabele die een functie van $n - 1$ variabelen teruggeeft, waarbij we dus

$$A_1 \times \ldots A_n \to B$$

vervangen door

$$A_1 \to (A_2 \times \ldots A_n \to B)$$

Indien $n > 2$, dan kan men

$$A_2 \times \ldots A_n \to B$$

weer vervangen door

$$A_2 \to (A_3 \times \ldots A_n \to B)$$

enz., tot men uiteindelijk

$$A_1 \to (A_2 \to \ldots (A_n \to B))$$

bekomt.

Dit mechanisme wordt "currying" genoemd naar Haskell B. Curry. We kunnen "currying" gebruiken in de λ-calculus omdat deze toelaat om functies van hogere orde, d.w.z. functies die b.v. een functie als functiewaarde teruggeven, te definiëren.

Het bovenstaande laat ons toe om b.v. de λ-expressie[5]

$$G \equiv \lambda x.\lambda f.(f)x$$

te beschouwen als een functie G van twee variabelen (in "gewone" notatie):

$$G(x, f) = f(x)$$

die zijn tweede parameter (f) toepast op zijn eerste parameter (x).

Currying is op zich interessant voor informatica vermits het toelaat om een gedeeltelijke evaluatie te doen wanneer b.v. niet alle parameters van een functie met verschillende parameters beschikbaar zijn. Het gebruik van zogenaamde "objecten" in de taal Scheme is hiervan een voorbeeld.

2.3 Vrije en gebonden variabelen

Het toepassen van een functie wordt in de λ-calculus gesimuleerd door het *herschrijven* van een λ-expressie door een andere (hopelijk eenvoudiger) expressie.

B.v. kan $(\lambda x.x)y$ herschreven worden als y waarbij we dit bekomen door de declaratie van de formele parameter λx te laten vallen en in het voorschrift "x" de formele parameter variabele x te vervangen door de actuele parameter y. Formeel zullen we dit definieren door middel van substitutie (van formele door actuele parameters). Om bvb.

$$(\lambda x.(x)x)y$$

'uit te rekenen' dienen we alle voorkomens van x (de formele parameter) in $(x)x$ (het voorschrift) te substitueren door y (de actuele parameter) zodat het resultaat $(y)y$ wordt. We moeten dus een definitie opstellen die precies vastlegt hoe we alle voorkomens van een variabele in een λ-expressie kunnen vervangen door een andere λ-expressie. Men dient echter zeer zorgvuldig te zijn bij de precieze definitie van substitutie. Men kan b.v.

$$(\lambda x.\lambda y.x)y$$

[5]Het symbool $\equiv$ wordt in deze tekst (en in vele andere wiskundige teksten) gebruikt om 'is per definitie gelijk aan' aan te geven. $\equiv$ is als het ware de `define` van de wiskunde.

niet zomaar herschrijven als

$$\lambda y.y$$

waarbij in het voorschrift horend bij λx (t.t.z. $\lambda y.x$) x vervangen werd door y. Inderdaad, intuïtief stelt $(\lambda x.\lambda y.x)$ een functie van twee parameters voor (we gebruiken currying) die de eerste parameter als functiewaarde geeft. Echter, door in $(\lambda x.\lambda y.x)y$ x te vervangen door y verkrijgen we $\lambda y.y$, een functie die steeds de (oorspronkelijke) tweede parameter teruggeeft! Het probleem zit hem in het feit dat y een "vrije" variabele is [6] die ongewild in de substitutie gebonden wordt door de "binding" λy in $(\lambda x.\lambda y.x)y$. Inderdaad is de tweede y in $\lambda y.y$ niet vrij maar verwijst die gewoon naar de parameter van de functie.

Merk op dat het probleem zich niet voordoet als we de λy in $(\lambda x.\lambda y.x)y$ vervangen door een ongebruikte variabele, b.v. z:

$$(\lambda x.\lambda z.x)y$$

wordt dan, na subsitutie van x door y:

$$\lambda z.y$$

wat een functie is die steeds y (de oorspronkelijke tweede parameter) als functiewaarde geeft.

De onderstaande definities zullen ons toelaten om substitutie zo te definiëren dat problemen zoals boven geschetst niet voorkomen.

Definitie 2

- *Voor een λ-expressie van de vorm*

  $$\lambda x.M$$

  zegt men dat

  - λx *een* **binding** *is van x in M*
  - *het* **bereik** *van de binding is M, dit wil zeggen dat alle (nog niet gebonden) voorkomens van x in $\lambda x.M$* **gebonden** *zijn.*

- *In een λ-expressie heten alle voorkomens van variabelen die niet gebonden zijn* **vrij**.

We definiëren nu de verzameling $VV(M)$ die alle variabelen uit M bevat die minstens één keer vrij voorkomen in M.

Definitie 3 *Weze $M \in \Lambda$. De verzameling $VV(M)$ van* **vrije variabelen** *van M wordt gedefinieerd door:*

- $\forall x \in V \ : \ VV(x) = \{x\}$

[6] In informatica termen kan men een vrije variabele beschouwen als een globale variabele (of in elk geval een variabele die in een hogere bereik ("scope") werd gedefinieerd.

- $VV(\lambda x.N) = VV(N) \backslash \{x\}$
- $VV((M)N) = VV(M) \cup VV(N)$

Een λ-expressie zonder vrije variabelen heet **gesloten** *of een* **combinator**. *We noteren de verzameling van combinatoren als $\Lambda_0 \subset \Lambda$.*

2.4 Substitutie

Intuïtief willen we een toepassing (functieaanroep)

$$(\lambda x.M)P$$

uitwerken door in het voorschrift M elk voorkomen van de formele parameter x die "hoort bij" de binding λx te vervangen door de actuele parameter P. De voorkomens van x die "horen bij" de binding λx zijn natuurlijk precies de voorkomens van x die *vrij* zijn in M (let op: vrij in M, niet in $\lambda x.M$).

Formeel zullen we die substitutie noteren als

$$[P/x]M$$

Intuïtief lezen we dit als "in M alle voorkomens van x vervangen door P". Echter, bij de precieze definitie van $[P/x]M$ zullen we er moeten voor zorgen dat vrije voorkomens van variabelen in P niet "per vergissing" (zoals in het voorbeeld aangehaald in de vorige sectie) gebonden worden (in de gesubstitueerde voorkomens van P).

Definitie 4 *Beschouw twee λ-expressies P en M, en een variabele $x \in V$. De* **substitutie** *$[P/x]M$ van P voor x in M, wordt als volgt gedefinieerd:*

$[P/x]x = P$		*(S1)*
$[P/x]y = y$	*als* $y \in V \backslash \{x\}$	*(S2)*
$[P/x](F)Q = ([P/x]F)[P/x]Q$		*(S3)*
$[P/x]\lambda x.M = \lambda x.M$		*(S4)*
$[P/x]\lambda y.M = \lambda y.[P/x]M$	*als* $y \neq x$ *en* $y \notin VV(P)$	*(S5)*
$[P/x]\lambda y.M = \lambda z.[P/x][z/y]M$	*als* $y \neq x$ *en* $z \notin VV(P)$ *en* $y \in VV(P)$	*(S6)*

De regels in Definitie 4 dekken alle mogelijke gevallen van λ-expressies, en vertellen ons voor ieder geval hoe we de substitutie moeten doorvoeren. Regels (S1) en (S2) behandelen variabelen, regel (S3) behandelt applicaties, en regels (S4), (S5) en (S6) behandelen abstractie. Intuïtief moeten we Definitie 4 als volgt begrijpen.

- Regel 1 is triviaal, en zegt dat als we in x, x moeten vervangen door P, we dan P bekomen.
- Regel 2 is eveneens triviaal, en zegt dat als we in y, x moeten vervangen door P, we dan y bekomen (immers, $x \neq y$ en dus is er helemaal geen x om te vervangen).

- Regel 3 zegt dat als we met een applicatie te maken hebben, we de substitutie moeten doorvoeren op beide deel-λ-expressies van de applicatie.
- Regel 4 zegt dat als we in $\lambda x.M$ (alle) x'en moeten vervangen door P, we dan helemaal niets moeten doen (immers, de x'en in $\lambda x.M$ zijn gebonden door de λx!)
- Regel 5 zegt dat we, in het algemeen, als we in $\lambda y.M$ (alle) x'en moeten vervangen door P, we de λy moeten laten staan, en de substitutie doorvoeren in M (zie regel 6 voor de uitzondering op deze regel).
- Regel 6 behandelt de uitzonderingen op regel 5, namelijk wanneer we "per vergissing" een vrije variabele in M (verkeerdelijk) zouden kunnen binden. Dit is het geval wanneer y vrij voorkomt in P: bij klakkeloze substitutie zouden de oorspronkelijk vrij voorkomende y's (in P) immers plots (verkeerdelijk) gebonden worden door de $\lambda.y$ van $\lambda y.M$! Om ervoor te zorgen dat zo'n verkeerdelijke binding niet mogelijk is, gaan we in dit geval eerst een "hernoeming" doorvoeren: we herschrijven $\lambda y.M$ tot de vorm $\lambda z.M$, waarbij we alle voorkomens y in M vervangen door z (m.a.w. we voeren de substitutie $[z/y]M$ door). Hierbij mag z uiteraard niet vrij voorkomen in P, anders zitten we met hetzelfde probleem. Eenmaal deze hernoeming doorgevoerd, komen we in het geval van regel 5, en kunnen we de substitutie doorvoeren als in regel 5, m.a.w. houden we

  $$\lambda z.[P/x]M'$$

  over, waarbij we weten dat $z \notin VV(P)$

We beschouwen nu enkele voorbeelden van substituties:

Voorbeeld 2

$$\begin{array}{lll} & [u/x]\lambda u.x & (u \in VV(u) \text{ dus S6}) \\ = & \lambda z.[u/x][z/u]x & (\text{S2}) \\ = & \lambda z.[u/x]x & (\text{S1}) \\ = & \lambda z.u & \end{array}$$

$$\begin{array}{lll} & [u/x]\lambda u.u & (u \in VV(u) \text{ dus S6}) \\ = & \lambda z.[u/x][z/u]u & (\text{S1}) \\ = & \lambda z.[u/x]z & (\text{S2}) \\ = & \lambda z.z & \end{array}$$

$$\begin{array}{lll} & [u/x]\lambda y.x & (\text{S5}) \\ = & \lambda y.[u/x]x & (\text{S1}) \\ = & \lambda y.u & \end{array}$$

$$\begin{array}{lll} & [u/x]\lambda y.u & (\text{S5}) \\ = & \lambda y.[u/x]u & (\text{S2}) \\ = & \lambda y.u & \end{array}$$

Voorbeeld 3

$$
\begin{array}{lll}
 & [\lambda u.(y)u/x]\lambda y.(x)y & (y \in VV(P) \text{ dus S6}) \\
= & \lambda z.[\lambda u.(y)u/x][z/y](x)y & (\text{S3}) \\
= & \lambda z.[\lambda u.(y)u/x]([z/y]x)[z/y]y & (\text{S2}) \\
= & \lambda z.[\lambda u.(y)u/x](x)[z/y]y & (\text{S1}) \\
= & \lambda z.[\lambda u.(y)u/x](x)z & (\text{S3}) \\
= & \lambda z.([\lambda u.(y)u/x]x)[\lambda u.(y)u/x]z & (\text{S1}) \\
= & \lambda z.(\lambda u.(y)u)[\lambda u.(y)u/x]z & (\text{S2}) \\
= & \lambda z.(\lambda u.(y)u)z &
\end{array}
$$

2.5 Object symbolen en Meta symbolen

In de λ-calculus zijn we tot nu toe twee soorten symbolen tegengekomen. De eerste soort symbolen zijn de 'lettertjes' die deel uitmaken van de λ-calculus. Voorbeelden zijn variabelen, haakjes, de punt (.) en de lambda (λ). Deze symbolen noemt men *object symbolen* . Ze zitten als het ware *in* het systeem van λ-calculus .

De tweede soort symbolen zijn 'wiskundige symbolen die we gebruiken om λ-calculus te beschrijven'. Deze symbolen behoren tot ons wiskundig arsenaal en zitten niet echt *in* de λ-calculus. Ze worden gebruikt om *over* het λ-systeem te redeneren. Men noemt ze dan ook *meta symbolen.* Voorbeelden van meta symbolen zijn $VV()$ en de substitutie-operator [./.]. die 2 λ-expressies en een variabele (allen object symbolen) omzet in een nieuwe λ-expressie.

3 Lambda expressies en berekenbaarheid

3.1 Beta-gelijkheid

De λ-calculus kan beschouwd worden als een theorie over gelijkheid tussen λ-expressies. De onderstaande definitie definieert deze zogenaamde β-gelijkheid.

Definitie 5 *De relatie* $=_\beta \subset \Lambda \times \Lambda$ *wordt gedefinieerd door de onderstaande axiomas:*

$$
\begin{array}{ll}
(\lambda x.M)P =_\beta [P/x]M & (\beta) \\
\lambda x.M =_\beta \lambda z.[z/x]M \ \textit{als}\ z \notin VV(M) & (\alpha) \\
M =_\beta M & \textit{(reflexief)} \\
M =_\beta N \Rightarrow N =_\beta M & \textit{(symmetrisch)} \\
M =_\beta N \wedge N =_\beta L \Rightarrow N =_\beta L & \textit{(transitief)} \\
M =_\beta M' \wedge P =_\beta P' \Rightarrow (M)P =_\beta (M')P' & \textit{(congruent)} \\
M =_\beta M' \Rightarrow \lambda x.M =_\beta \lambda x.M' & \textit{(congruent)}
\end{array}
$$

Definitie 5 geeft ons regels die ons toelaten een λ-expressie te herschrijven in een equivalente (hopelijk simpelere) λ-expressie. Het laat ons dus als het ware toe λ-expressies *uit te werken* (te vergelijken met *evalutatie* van expressies in programmeertalen). Het β-axioma [7] geeft onze intuitie weer over het toepassen van

[7] Historisch was (β) het tweede axioma, na (α), vanwaar de naam.

een functie op een actuele parameter: het volstaat om in het functievoorschrift de formele parameter te vervangen door de actuele parameter. Het α-axioma zegt dat men de naam van een formele parameter in een functiedefinitie mag vervangen door een andere.

De volgende drie axoma's zijn nodig om de gelijkheid "netjes" te definïeren: ze zorgen ervoor dat de β-gelijkheid zich gedraagt zoals we dat van een gelijkheid verwachten, nl. reflexief, symmetrisch en transitief (de β-gelijkheid bepaalt m.a.w. een equivalentie relatie over de set van λ-expressies).

De laatste twee expressies definïeren de congruentie van de β-gelijkheid. Intuitief gezien zegt regel 6 (de eerste congruentie regel) dat het in een gegeven applicatie $(M)P$ niet uitmaakt of we eerst M reduceren (m.a.w. β-gelijkheid regels toepassen op M), of eerst P reduceren. De laatste regel (de tweede congruentie regel) zegt dat we, gegeven een λ-expressievan de vorm $\lambda x.M$, we de $\lambda x.$ mogen laten staan, en de M reduceren.

Laten we een voorbeeld beschouwen waarin we de β-gelijkheidsregels toepassen:

Voorbeeld 4 In de onderstaande berekening zijn de deelexpressies waarop een axioma wordt toegepast onderlijnd.

$$\begin{array}{lll}
 & ((\lambda x.\lambda y.\lambda z.((x)z)(y)z)\lambda x.x)\underline{\lambda x.x} & (\alpha)\\
=_\beta & ((\lambda x.\lambda y.\lambda z.((x)z)(y)z)\underline{\lambda x.x})\lambda u.u & (\alpha)\\
=_\beta & (\underline{(\lambda x.\lambda y.\lambda z.((x)z)(y)z)\lambda v.v})\lambda u.u & (\beta)\\
=_\beta & \underline{(\lambda y.\lambda z.((\lambda v.v)z)(y)z)\lambda u.u} & (\beta)\\
=_\beta & \underline{(\lambda y.\lambda z.(z)(y)z)\lambda u.u} & (\beta)\\
=_\beta & \lambda z.(z)\underline{(\lambda u.u)z} & (\beta)\\
=_\beta & \lambda z.(z)z &
\end{array}$$

3.2 Rekenen met Lambda Expressies

Hoewel de λ-calculus geen constanten kent blijkt het toch mogelijk te zijn om zowel de natuurlijke getallen als de gewone rekenkundige functies voor te stellen d.m.v. λ-expressies, waarbij β-gelijkheid kan gebruikt worden om het gewenste effekt van die functies aan te tonen.

Eerst hebben we een hulpdefinitie nodig:

Definitie 6 *Beschouw λ-expressies F en M. De expressie $(F)^n M$ wordt inductief gedefinieerd door:*

$$\begin{array}{rcl}
(F)^0 M & \equiv & M\\
(F)^{1+n} M & \equiv & (F)(F)^n M
\end{array}$$

Merk op dat voor elke n, $(.)^n$. een meta operatie is. Zij neemt 2 λ-expressies F en M en levert de λ-expressie $(F)(F)(F)\ldots(F)M$ op. Deze meta operatie zit niet *in* de verzameling der λ-expressies. Ze laat ons enkel toe om bepaalde elementen van Λ veel korter op te schrijven. De volgende eigenschap zal later nuttig blijken.

Lemma 1 *Beschouw λ-expressies F en M. Voor alle $n, m \in \mathbf{N}$ geldt dat*

$$(F)^{n+m}M \equiv (F)^n(F)^m M \tag{2}$$

Bewijs. We gebruiken inductie op n.

1. Als $n = 0$ dan geldt dat

$$\begin{array}{lll} (F)^{0+m}M & \equiv (F)^m M & \\ & \equiv M' & \text{neem } M' \equiv (F)^m M \\ & \equiv (F)^0 M' & \text{definitie 6} \\ & \equiv (F)^0(F)^m M & \text{definitie } M' \end{array}$$

2. Stel dat (1) geldt voor $n = k$. Als $n = k + 1$ dan geldt dat

$$\begin{array}{lll} (F)^{1+k+m}M & \equiv (F)^{1+(k+m)}M & \\ & \equiv (F)(F)^{k+m}M & \text{definitie 6} \\ & \equiv (F)(F)^k(F)^m M & \text{inductiehypothese} \\ & \equiv (F)(F)^k M' & \text{neem } M' \equiv (F)^m M \\ & \equiv (F)^{1+k}M' & \text{definitie 6} \\ & \equiv (F)^{1+k}(F)^m M & \text{definitie } M' \end{array}$$

Q.E.D

Het getal 0 zal nu voorgesteld worden door de λ-expressie $\lambda f.\lambda x.x$, het getal 1 door $\lambda f.\lambda x.(f)x$, 2 door $\lambda f.\lambda x.(f)(f)x$, enz. In het algemeen hebben we de volgende definitie:

Definitie 7 *De zogenaamde "***Church getallen***" $\mathbf{c}_n$ ($n \in \mathbf{N}$) worden gedefinieerd door*

$$\mathbf{c}_n \equiv \lambda f.\lambda x.(f)^n x$$

Intuïtief kunnen we de definitie van een Church getal c_n lezen als: "gegeven een functie (f) en een parameter (x) (aangeduid door de beide formele parameters), dan (het voorschrift volgt) wordt deze functie (f) n keer toegepast op x". Onthoud deze intuïtieve interpretatie van een Church getal, ze zal toelaten het vervolg beter te begrijpen. Nu we getallen kunnen voorstellen in λ-calculus kunnen we gaan denken over de voorstelling van operaties op getallen. We zullen dan zeggen dat een operatie op getallen in λ-calculus is voor te stellen (λ-definieerbaar), indien we een combinator kunnen vinden waarvan het effect op Church getallen hetzelfde is als de operatie toegepast op de 'echte' getallen die met deze Church getallen overeenkomen. Formeel:

Definitie 8 *Een numerieke functie $f : \mathbf{N}^p \to \mathbf{N}$ met $p \in N$ parameters is* **λ-definieerbaar** *als er een combinator F bestaat zodat*

$$((((F)\mathbf{c}_{n_1})\mathbf{c}_{n_2})\ldots)\mathbf{c}_{n_p} =_\beta \mathbf{c}_{f(n_1,n_2,\ldots,n_p)}$$

We zullen eerste aantonen dat de "opvolger" functie

$$\mathbf{succ}(n) = n + 1$$

λ-definieerbaar is.

Een juiste definitie van **succ** moet zo zijn dat b.v.

$$(\mathbf{succ})\mathbf{c}_0 \equiv (\mathbf{succ})\lambda f.\lambda x.x =_\beta \lambda f.\lambda x.(f)x \equiv \mathbf{c}_1$$

en ook

$$(\mathbf{succ})\mathbf{c}_1 \equiv (\mathbf{succ})\lambda f.\lambda x.(f)x =_\beta \lambda f.\lambda x.(f)(f)x \equiv \mathbf{c}_2$$

In het algemeen willen we dus dat $(\mathbf{succ})\mathbf{c}_n$ een extra "aanroep" van f toevoegt aan het voorschrift van $\mathbf{c}_n$:

$$(\mathbf{succ})\mathbf{c}_n =_\beta \lambda f.\lambda x.(f)voorschrift_n \tag{3}$$

waarbij $voorschrift_n$ een afkorting is voor $(f)^n x$.

Maar uit de definitie van een Church getal (herinner u de (intuïtieve) betekenis van een Church-getal; we hebben deze niet toevallig zo gedefinïeerd):

$$\mathbf{c}_n \equiv \lambda f.\lambda x.(f)^n x$$

geldt dat

$$\begin{array}{lll} ((\mathbf{c}_n)f)x & \equiv ((\lambda f.\lambda x.(f)^n x)f)x & (\beta) \\ & =_\beta (\lambda x.(f)^n x)x & (\beta) \\ & =_\beta (f)^n x & \\ & \equiv voorschrift_n & \end{array}$$

M.a.w., wanneer we een Church getal nemen (wat uiteraard een λ-expressie is), en we passen die achtereenvolgens toe op f en x, dan bekomen we het voorschrift van het Church getal!

We kunnen nu deze kennis gebruiken om (3) te herschrijven als

$$(\mathbf{succ})\mathbf{c}_n =_\beta \lambda f.\lambda x.(f)((\mathbf{c}_n)f)x$$

waaruit we de gewenste definitie van **succ** afleiden (waarbij de n in $\lambda n.$ de formele parameter is waarin het Church getal zal terechtkomen bij applicatie):

$$\mathbf{succ} \equiv \lambda n.\lambda f.\lambda x.(f)((n)f)x$$

Intuïtief kunnen we **succ** dus lezen als "gegeven een Church getal ($\lambda n.$), dan geven we een λ-expressie terug die de vorm heeft van een Church getal ($\lambda f.\lambda x.$), waarbij het voorschrift van dit Church getal het oorspronkelijke voorschrift is $((n)f)x$ met daarvoor een extra aanroep van f, nl. (f)".

Als voorbeeld berekenen $(\mathbf{succ})\mathbf{c}_0 =_\beta \mathbf{c}_1$ en $(\mathbf{succ})\mathbf{c}_1 =_\beta \mathbf{c}_2$:

Voorbeeld 5

$$\begin{array}{lll}
(\mathbf{succ})\mathbf{c}_0 & \equiv \underline{(\lambda n.\lambda f.\lambda x.(f)((n)f)x)\lambda f.\lambda x.x} & (\beta)\\
& =_\beta \lambda f.\lambda x.(f)(\underline{(\lambda f.\lambda x.x)f})x & (\beta)\\
& =_\beta \lambda f.\lambda x.(f)\underline{(\lambda x.x)x} & (\beta)\\
& =_\beta \lambda f.\lambda x.(f)x & \\
& \equiv \mathbf{c}_1 &
\end{array}$$

$$\begin{array}{lll}
(\mathbf{succ})\mathbf{c}_1 & \equiv \underline{(\lambda n.\lambda f.\lambda x.(f)((n)f)x)\lambda f.\lambda x.(f)x} & (\beta)\\
& =_\beta \lambda f.\lambda x.(f)(\underline{(\lambda f.\lambda x.(f)x)f})x & (\beta)\\
& =_\beta \lambda f.\lambda x.(f)\underline{(\lambda x.(f)x)x} & (\beta)\\
& =_\beta \lambda f.\lambda x.(f)(f)x & \\
& \equiv \mathbf{c}_2 &
\end{array}$$

Eigenlijk is **succ** een bijzonder geval van de meer algemene functie **plus** die de optelling van twee natuurlijke getallen simuleert:

Stelling 1 *Definieer*

$$\mathbf{plus} \equiv \lambda n.\lambda m.\lambda f.\lambda x.((n)f)((m)f)x$$

dan geldt voor alle $m, n \in \mathbf{N}$

$$((\mathbf{plus})\mathbf{c}_n)\mathbf{c}_m =_\beta \mathbf{c}_{n+m}$$

Opnieuw maken we gebruik van de definitie van een Church getal. Intuïtief lezen we **plus** als "gegeven twee Church getallen (aangeduid door $\lambda n.\lambda m.$), dan geven we een λ-expressie terug die de vorm heeft van een Church getal ($\lambda f.\lambda x.$), waarbij het voorschrift van dit Church getal het oorspronkelijk voorschrift van het tweede Church getal is, nl. $((m)f)x$ (achteraan) met daarvoor het aantal functie-aanroepen van het eerste Church getal toegevoegd, nl. $((n)f)$". Om in te zien waarom het aantal functie-aanroepen van het eerste Church getal inderdaad netjes voor het voorschrift van het tweede Church getal komt, beschouw volgende reductie:

$$\begin{array}{lll}
((c_n)f)voorschrift_m & \equiv (\underline{(\lambda f.\lambda x.(f)^n x)f})voorschrift_m & \text{definitie } c_n\\
& =_\beta \underline{(\lambda x.(f)^n x)voorschrift_m} & (\beta)\\
& =_\beta (f)^n \underline{voorschrift_m} & (\beta)\\
& \equiv (f)^n(f)^m x & \text{definitie } voorschrift_m
\end{array}$$

Het formele bewijs is als volgt:

Bewijs.

$$
\begin{array}{lll}
((\mathbf{plus})\mathbf{c}_n)\mathbf{c}_m & \equiv (\underline{(\lambda n.\lambda m.\lambda f.\lambda x.((n)f)((m)f)x)\mathbf{c}_n})\mathbf{c}_m & \text{definitie } \mathbf{plus} \\
& =_\beta \underline{(\lambda m.\lambda f.\lambda x.((\mathbf{c}_n)f)((m)f)x)\mathbf{c}_m} & (\beta) \\
& \equiv (\lambda m.\lambda f.\lambda x.(\underline{(\lambda f.\lambda x.(f)^n x)f})((m)f)x)\mathbf{c}_m & \text{definitie } \mathbf{c}_n \\
& =_\beta (\lambda m.\lambda f.\lambda x.\underline{(\lambda x.(f)^n x)((m)f)x})\mathbf{c}_m & (\beta) \\
& =_\beta \underline{(\lambda m.\lambda f.\lambda x.(f)^n((m)f)x)\mathbf{c}_m} & (\beta) \\
& =_\beta \underline{\lambda f.\lambda x.(f)^n((\mathbf{c}_m)f)x} & (\beta) \\
& \equiv \lambda f.\lambda x.(f)^n(\underline{(\lambda f.\lambda x.(f)^m x)f})x & \text{definitie } \mathbf{c}_m \\
& =_\beta \lambda f.\lambda x.(f)^n\underline{(\lambda x.(f)^m x)x} & (\beta) \\
& =_\beta \lambda f.\lambda x.(f)^n(f)^m x & (\beta) \\
& \equiv \lambda f.\lambda x.(f)^{n+m}x & \text{lemma 1} \\
& \equiv \mathbf{c}_{n+m} & \text{definitie } \mathbf{c}_{n+m}
\end{array}
$$

Q.E.D

Men kan ook de Church getallen vermenigvuldigen. Eerst hebben we een hulpstelling nodig.

Lemma 2 *Voor alle $n, m \in N$ geldt dat*

$$((\mathbf{c}_n)f)^m y =_\beta (f)^{n\times m} y \tag{4}$$

Intuïtief zegt dit lemma dat, als we een λ-expressie $((c_n)f)$ m keer herhalen en deze toepassen op y, dit overeenkomt met $n * m$ keer de functie f toe te passen op y. Feitelijk zorgt de $((c_n)f)^m$ ervoor dat we m keer "copieren". Om dit in te zien, beschouw de volgende deductie[8]:

$$
\begin{array}{lll}
((c_n)f)^m y & \equiv (\underline{(\lambda f.\lambda x.(f)^n x)f})^m y & \text{definitie } c_n \\
& =_\beta \underline{(\lambda x.(f)^n x)^m y} & (\beta) \\
& \equiv (\lambda x.(f)^n x)\ldots(\lambda x.(f)^n x)\underline{(\lambda x.(f)^n x)y} & \text{definitie } (f)^m \text{ (met m herhalingen)} \\
& =_\beta (\lambda x.(f)^n x)\ldots\underline{(\lambda x.(f)^n x)(f)^n y} & (\beta) \\
& =_\beta (\lambda x.(f)^n x)\ldots\underline{(f)^n(f)^n y} & (\beta)
\end{array}
$$

Het formele bewijs is als volgt:

Bewijs. We gebruiken inductie over m.

1. Als $m = 0$ dan wordt (4):

$$
\begin{array}{lll}
((\mathbf{c}_n)f)^m y & \equiv ((\mathbf{c}_n)f)^0 y & \\
& \equiv y & \text{definitie 6} \\
& \equiv (f)^0 y & \text{definitie 6} \\
& \equiv (f)^{n\times 0} y &
\end{array}
$$

[8]Het gebruik van de ... notatie is hieronder niet formeel correct te noemen, doch ze wordt hier ter illustratie van het principe gebruikt

2. Stel dat (4) geldt voor $m = k$:

$$\begin{array}{lll}
((\mathbf{c}_n)f)^{k+1}y & \equiv ((\mathbf{c}_n)f)((\mathbf{c}_n)f)^k y & \text{definitie 6} \\
& \equiv ((\mathbf{c}_n)f)(f)^{n\times k}y & \text{inductiehypothese} \\
& \equiv (\underline{(\lambda f.\lambda x.(f)^n x)f})(f)^{n\times k}y & \text{definitie } \mathbf{c}_n \\
& =_\beta \underline{(\lambda x.(f)^n x)(f)^{n\times k}y} & (\beta) \\
& =_\beta (f)^n(f)^{n\times k}y & (\beta) \\
& \equiv (f)^{n+n\times k}y & \text{lemma 1} \\
& \equiv (f)^{(n+1)\times k}y &
\end{array}$$

Q.E.D

We kunnen nu de vermenigvuldiging definïeren:

Stelling 2 *Definieer*

$$\mathbf{times} \equiv \lambda n.\lambda m.\lambda f.(n)(m)f$$

dan geldt voor alle $m, n \in \mathbf{N}$

$$((\mathbf{times})\mathbf{c}_n)\mathbf{c}_m =_\beta \mathbf{c}_{n\times m}$$

Intuïtief valt bovenstaande stelling makkelijk te begrijpen wanneer we terugdenken aan het "copieer-principe" van Lemma 2, immers (beschouw het voorschrift $\lambda f.(n)(m)f$):

$$\begin{array}{lll}
\lambda f.(c_n)(c_m)f & \equiv \lambda f.(\lambda f.\lambda x.(f)^n x)(c_m)f & \text{definitie } c_n \\
& =_\beta \lambda f.\lambda x.((c_m)f)^n x & (\beta) \\
& =_\beta \lambda f.\lambda x.(f)^{n\times m}x & (\text{lemma 2})
\end{array}$$

Het volledige formeel bewijs is als volgt:

Bewijs.

$$\begin{array}{lll}
((\mathbf{times})\mathbf{c}_n)\mathbf{c}_m & \equiv (\underline{(\lambda n.\lambda m.\lambda f.(n)(m)f)\mathbf{c}_n})\mathbf{c}_m & \text{definitie } \mathbf{times} \\
& =_\beta \underline{(\lambda m.\lambda f.(\mathbf{c}_n)(m)f)\mathbf{c}_m} & (\beta) \\
& \equiv (\lambda m.\lambda f.\underline{(\lambda f.\lambda x.(f)^n x)(m)f})\mathbf{c}_m & \text{definitie } \mathbf{c}_n \\
& =_\beta \underline{(\lambda m.\lambda f.\lambda x.((m)f)^n x)\mathbf{c}_m} & (\beta) \\
& =_\beta \lambda f.\lambda x.((\mathbf{c}_m)f)^n x & (\beta) \\
& =_\beta \lambda f.\lambda x.(f)^{m\times n}x & \text{lemma 2} \\
& \equiv \mathbf{c}_{n\times m} & \text{definitie } \mathbf{c}_{n\times m}
\end{array}$$

Q.E.D

Het bedenken van een combinator voor de **minus**functie is wat gecompliceerder.

We definiëren eerst een paar combinatoren die verder nog van pas zullen komen.

Definitie 9

$$
\begin{array}{ll}
\textbf{true} & \equiv \lambda t.\lambda f.t \\
\textbf{false} & \equiv \lambda t.\lambda f.f \\
\textbf{if} & \equiv \lambda c.\lambda d.\lambda e.((c)d)e \\
\textbf{iszero} & \equiv \lambda n.((n)\lambda x.\textbf{false})\textbf{true} \\
\textbf{cons} & \equiv \lambda a.\lambda d.\lambda z.((z)a)d \\
\textbf{car} & \equiv \lambda a.\lambda d.a \\
\textbf{cdr} & \equiv \lambda a.\lambda d.d
\end{array}
$$

Eerst en vooral valt op te merken dat bovenstaande combinatoren zo gecontrueerd zijn zodat ze goed met elkaar samenwerken. Zo is bijv. de if-combinator zo geconstrueerd zodat hij goed $true$ en $false$ samenwerkt. M.a.w., if verwacht als eerste argument een λ-expressie die reduceert naar de vorm $true$ of $false$ zoals hierboven gedefinïeerd. Onder deze voorwaarde zal onze if-combinator functioneren zoals we dat van een if verwachten. Echter, als we als argument een λ-expressie van een andere vorm gebruiken (en er niets in de λ-calculus die ons dat verbiedt), dan zal onze if-combinator niet meer naar behoren werken, of beter gezegd, hij zal zich onvoorspelbaar gedragen, en mogelijk niet meer zoals we verwachten dat een if-combinator zou moeten werken. In het algemeen is het probleem dat de originele λ-calculus zoals wij die hier bestuderen niet getypeerd is: we kunnen voor een gegeven λ-expressie niet eisen dat hij enkel op welbepaalde λ-expressies toegepast mag worden. Dit heeft zware gevolgen: er werd voor de niet-getypeerde λ-calculus aangetoond dat men bij bepaalde reducties ongewenste resultaten kan bekomen! Verschillende oplossingen werden ontwikkeld om dit probleem op te lossen, waaronder getypeerde vormen van λ-calculus doch een discussie over de specifieke problemen met de niet-getypeerde λ-calculusen de oplossingen zou ons voor het doel van deze cursus veel te ver leiden. Je mag aldus aannemen dat, zolang we onze combinatoren "correct" gebruiken (m.a.w. we ze gebruiken zoals ze bedoeld zijn, en onze combinatoren dus niet toepassen op andere, ongewenste λ-expressies), we niet in de problemen zullen komen.

Untuïtief moeten we bovenstaande combinatoren als volgt begrijpen:

- **true**: gegeven twee argumenten ($\lambda t.$ en $\lambda f.$), dan (het voorschrift volgt) geeft **true** het eerste argument terug.
- **false**: gegeven twee argumenten ($\lambda t.$ en $\lambda f.$), dan (het voorschrift volgt) geeft **false** het tweede argument terug.
- **if**: gegeven een conditie ($\lambda c.$), een $true$ en een $else$ take (respektievelijk $\lambda d.$ en $\lambda e.$, dan (het voorschrift $((c)d)e$ volgt) passen we de conditie toe op (achtereenvolgend) beide argumenten d en e. Gezien **true** het eerste argument teruggeeft, en **false** het tweede, zal de **if** combinator inderdaad werken zoals we dat verwachten. Hier komt dus duidelijk tot uiting dat **true** en **false** zo geconstrueerd zijn zodat ze met **if** samenwerken.

- **iszero**: gegeven een Church getal ($\lambda n.$), dan (het voorschrift volgt) passen we het Church getal n toe op de argumenten $\lambda x.\mathbf{false}$ en **true**. Het eerste argument $\lambda x.\mathbf{false}$ is een λ-expressie die, om het even wat het argument is, steeds **false** teruggeeft. Herinner u nu de (intuïtieve) definitie van een Church getal: gegeven een functie en een parameter, dan wordt die functie n keer toegepast op deze parameter. De functie is hier $\lambda x.\mathbf{false}$ (die steeds **false** teruggeeft!), het argument **true**. M.a.w., vanaf het moment dat $\lambda x.\mathbf{false}$ uitgevoerd wordt, dan zal het resultaat **false** zijn. Er is slecht één geval waarin deze functie *niet* zal uitgevoerd worden, nl. als het Church getal (n) c_0 was (dit is immers de definitie van c_n, nl. $\lambda f.\lambda x.x$): in dit geval krijgen we dus het tweede argument van n terug, nl. **true**.

- **cons**: gegeven twee argumenten ($\lambda a.$, de car, en $\lambda d.$, de cdr), dan (het voorschrift volgt) geeft **cons** een λ-expressie terug die één argument verwacht ($\lambda z.$) en, wanneer toegepast, dit argument zal toepassen op op de beide (eerdere) argumenten car a en cdr d. Een cons cell in de λ-calculus is dus eigenlijk een dispatch-functie die, gegeven de juiste functie (nl. car of cdr), het eerste of tweede argument zal teruggeven (bekijk aandachtig voorbeeld 8 om dit volledig te begijpen). **car** en **cdr** worden dus ook als dusdanig geconstrueerd (zie volgend).

- **car**: gegeven twee argumenten ($\lambda a.$ en $\lambda d.$), dan (het voorschrift volgt) geeft **car** het eerste argument (de c**a**r) terug.

- **cdr**: gegeven twee argumenten ($\lambda a.$ en $\lambda d.$), dan (het voorschrift volgt) geeft **cdr** het eerste argument (de c**d**r) terug.

Merk op dat **car** en **cdr** identiek gedefiniëerd zijn als **true** en **false**. De combinatoren van definitie 9 gedragen zich zoals verwacht; ziehier enkele voorbeelden:

Voorbeeld 6

$$\begin{aligned}
((\mathbf{true})A)B &\equiv \underline{((\lambda t.\lambda f.t)A)}B && \text{voor willekeurige } A, B \in \Lambda\\
&=_\beta \underline{(\lambda f.A)B}\\
&=_\beta A
\end{aligned}$$

$$\begin{aligned}
((\mathbf{false})A)B &\equiv \underline{((\lambda t.\lambda f.f)A)}B && \text{voor willekeurige } A, B \in \Lambda\\
&=_\beta \underline{(\lambda f.f)B}\\
&=_\beta B
\end{aligned}$$

$$\begin{aligned}
(\mathbf{if})\mathbf{true} &\equiv \underline{(\lambda c.\lambda d.\lambda e.((c)d)e)\mathbf{true}}\\
&=_\beta \lambda d.\lambda e.((\mathbf{true})d)e\\
&\equiv \lambda d.\lambda e.\underline{((\lambda t.\lambda f.t)d)}e\\
&=_\beta \lambda d.\lambda e.\underline{(\lambda f.d)e}\\
&=_\beta \lambda d.\lambda e.d\\
&\equiv \mathbf{true}
\end{aligned}$$

Voorbeeld 7

$$
\begin{array}{lll}
(\mathbf{true})\mathbf{false} & \equiv \underline{(\lambda t.\lambda f.t)\mathbf{false}} & \\
& =_\beta \lambda f.\mathbf{false} & \\
(\lambda f.\mathbf{false})^n M & \equiv \underline{(\lambda f.\mathbf{false}))(\lambda f.\mathbf{false})^{n-1} M} & \text{als } n > 0 \\
& =_\beta \mathbf{false} &
\end{array}
$$

$$
\begin{array}{lll}
(\mathbf{iszero})\mathbf{c}_0 & \equiv \underline{(\lambda n.((n)\lambda.\mathbf{false})\mathbf{true})\mathbf{c}_0} & \\
& =_\beta ((\mathbf{c}_0)\lambda.\mathbf{false})\mathbf{true} & \\
& \equiv \underline{((\lambda f.\lambda x.x)\lambda.\mathbf{false})}\mathbf{true} & \\
& =_\beta \underline{(\lambda x.x)\mathbf{true}} & \\
& =_\beta \mathbf{true} &
\end{array}
$$

$$
\begin{array}{lll}
(\mathbf{iszero})\mathbf{c}_n & \equiv \underline{(\lambda n.((n)\lambda.\mathbf{false})\mathbf{true})\mathbf{c}_n} & \text{stel } n > 0 \\
& =_\beta ((\mathbf{c}_n)\lambda f.\mathbf{false})\mathbf{true} & \\
& \equiv \underline{((\lambda f.\lambda x.(f)^n x)\lambda f.\mathbf{false})}\mathbf{true} & \\
& =_\beta \underline{(\lambda x.(\lambda f.\mathbf{false})^n x)}\mathbf{true} & \\
& =_\beta \underline{(\lambda x.\mathbf{false})\mathbf{true}} & \\
& =_\beta \mathbf{false} &
\end{array}
$$

De combinatoren **car** en **cdr** kunnen gebruikt worden om een component van een paar $((\mathbf{cons})A)B$ te selecteren:

Voorbeeld 8

$$
\begin{array}{ll}
((\mathbf{cons})A)B & \equiv \underline{((\lambda a.\lambda d.\lambda z.((z)a)d)A)}B \\
& =_\beta \underline{(\lambda d.\lambda z.((z)A)d)B} \\
& =_\beta \lambda z.((z)A)B
\end{array}
$$

$$
\begin{array}{ll}
(((\mathbf{cons})A)B)\mathbf{car} & =_\beta \underline{(\lambda z.((z)A)B)\mathbf{car}} \\
& =_\beta \underline{((\mathbf{car})A)B} \\
& \equiv ((\lambda a.\lambda d.a)A)B \\
& =_\beta (\lambda d.A)B \\
& =_\beta A
\end{array}
$$

$$
\begin{array}{ll}
(((\mathbf{cons})A)B)\mathbf{cdr} & =_\beta \underline{(\lambda z.((z)A)B)\mathbf{cdr}} \\
& =_\beta \underline{((\mathbf{cdr})A)B} \\
& \equiv ((\lambda a.\lambda d.d)A)B \\
& =_\beta (\lambda d.d)B \\
& =_\beta B
\end{array}
$$

Met behulp van de voorgaande definities kunnen we nu een "voorganger" functie **pred** definieren waarbij

$$\mathbf{pred}(n) = n - 1 \text{ als } n > 0$$

Het idee is om paren van de vorm (c_n, c_{n-1}) aan te maken:

$$((\mathbf{cons})\mathbf{c}_n)\mathbf{c}_{n-1})$$

Als we erin slagen zo'n paren te vormen, dan kunnen we **pred** van c_n definïeren als de **cdr** van het paar (c_n, c_{n-1}):

$$(((\mathbf{cons})\mathbf{c}_n)\mathbf{c}_{n-1})\mathbf{cdr}$$

De uitdaging is dus om de paren (c_n, c_{n-1}) te definïeren (of beter gezegd *construeren*). Een eerste stap hiertoe is een hulpfunctie **nextp** te definïeren die, gegeven een paar (c_n, c_{n-1}), ons het "volgende paar" (c_{n+1}, c_n) teruggeeft:

$$(\mathbf{nextp})((\mathbf{cons})\mathbf{c}_n)\mathbf{c}_{n-1} =_\beta ((\mathbf{cons})\mathbf{c}_{n+1})\mathbf{c}_n$$

De combinator **nextp** is eenvoudig te construeren: het volstaat de eerste component (m.a.w. de **car**) van het argument (een **cons** cell) naar de tweede component in het resultaat te brengen, en de opvolger (**succ**) van de eerste component (de **car**) van het argument (de **cons** cell) in de eerste component van het resultaat te plaatsen:

$$\begin{aligned}\mathbf{nextp} &\equiv \lambda p.((\mathbf{cons})(\mathbf{succ})(p)\mathbf{car})(p)\mathbf{car} \\ &\equiv \lambda p.\lambda z.((z)(\lambda n.\lambda f.\lambda x.(f)((n)f)x)(p)\mathbf{car})(p)\mathbf{car}\end{aligned}$$

Om nu een cons cell (c_n, c_{n-1}) te construeren, volstaat het om, vertrekkend vanuit (c_0, c_0), n keer **nextp** toe te passen. Zoals we reeds weten kunnen we een functie g n keer itereren door gebruik te maken van (de definitie van) het Church getal $\mathbf{c}_n$, vermits b.v.

$$\begin{aligned}((\mathbf{c}_n)g)y &\equiv \underline{((\lambda f.\lambda x.(f)^n x)g)}y \\ &=_\beta \underline{(\lambda x.(g)^n x)y} \\ &=_\beta (g)^n y\end{aligned}$$

We kunnen aldus (c_n, c_{n-1}) genereren door:

$$((c_n)\mathbf{nextp})((\mathbf{cons})\mathbf{c}_0)\mathbf{c}_0$$

Uiteindelijk zijn we klaar om **pred** te definïeren, nl. als volgt

$$\mathbf{pred} \equiv \lambda n.(((n)\mathbf{nextp})((\mathbf{cons})\mathbf{c}_0)\mathbf{c}_0)\mathbf{cdr} \tag{5}$$

Indien we alle afkortingen in (5) uitschrijven bekomen we

$$\begin{aligned}\mathbf{pred} \equiv\ & \lambda n.(\lambda t.\lambda f.f)((n)\lambda p.\lambda z.((z)(\lambda n.\lambda f.\lambda x.(f)((n)f)x) \\ & (p)\lambda t.\lambda f.t)(p)\lambda t.\lambda f.t)\lambda z.((z)\lambda f.\lambda x.x)\lambda f.\lambda x.x\end{aligned}$$

wat welliswaar niet erg leesbaar is.

Met het gebruik van **pred** wordt de definitie van **minus** eenvoudig: om $n - m$ uit te rekenen volstaat het om **pred** m keer te itereren op $\mathbf{c}_n$:

$$\mathbf{minus} \equiv \lambda m.\lambda n.((m)\mathbf{pred})n$$

3.3 Fixpunten en recursie

We hebben in de vorige sectie reeds heel wat begrippen uit programmeertalen gedefiniëerd, o.a. getallen, bepaalde arithmetische functies (v.b. $+$, $-$, $*$), booleanse waarden ($true$ en $false$), if-expressies, cons-cellen met bijhorende operaties (car en cdr). Toch missen we nog één fundamenteel concept aanwezig in praktisch alle programmeertalen: recursie. Alvorens we echter kunnen overgaan tot het definiëren van recursieve functies in λ-calculus, hebben we een ander stukje cruciale theorie nodig, nl. deze van fixpunten. Een fixpunt (of dekpunt) van een functie kennen we uit de wiskunde: het is een punt dat, gegeven als input voor een functie, zichzelf als resultaat oplevert. Zo is bijv. 0 een fixpunt voor de functie $f(x) = 2x$, gezien $f(0) = 0$. Een fixpunt kan als volgt gedefiniëerd worden:

Definitie 10 *Voor een functie*

$$f : D \to D$$

heet $x \in D$ een **fixpunt** *van f als en slechts als*

$$f(x) = x$$

Voor λ-expressies wordt dit:

Definitie 11 *$X \in \Lambda$ is een fixpunt van $F \in \Lambda$ als en slechts als*

$$(F)X =_\beta X$$

Men kan zich nu afvragen of er überhaupt λ-expressies zijn die een fixpunt hebben, en als dat zo is, welke dit dan zijn en wat hun fixpunt is. Het antwoord op deze vragen werd ontdekt door Haskell B. Curry, en kan enigzins verrassend lijken: niet alleen is het zo dat iedere λ-expressie een fixpunt heeft, er bestaat bovendien een combinator die, gegeven een willekeurige λ-expressie, voor ons een fixpunt berekent! Dit wordt weergegeven in de volgende stelling:

Stelling 3

1. *Iedere λ-expressie heeft een fixpunt:*

$$\forall F \in \Lambda \, \exists X : (F)X =_\beta X$$

2. *Er is een* **fixpunt combinator**

$$\mathbf{Y} \equiv \lambda f.(\lambda x.(f)(x)x)\lambda x.(f)(x)x$$

waarvoor geldt dat

$$\forall F \in \Lambda : \ (F)(\mathbf{Y})F =_\beta (\mathbf{Y})F$$

Bewijs.

1. Definieer
$$W \equiv \lambda x.(F)(x)x$$
en
$$X \equiv (W)W$$
Dan geldt dat
$$\begin{aligned} X &\equiv (W)W \\ &\equiv (\lambda x.(F)(x)x)W \\ &=_\beta (F)(W)W \\ &=_\beta (F)X \end{aligned}$$
en dus is X een fixpunt van F.

2.
$$\begin{aligned} (\mathbf{Y})F &\equiv (\lambda f.(\lambda x.(f)(x)x)\lambda x.(f)(x)x)F \\ &=_\beta (\lambda x.(F)(x)x)\lambda x.(F)(x)x \\ &=_\beta (W)W \\ &=_\beta X \end{aligned}$$
Samen met punt (1) hierboven toont dit aan dat $(\mathbf{Y})F$ een fixpunt is van F.

Q.E.D

Gegeven een willekeurige λ-expressie M, is het dus een koud kunstje om een fixpunt voor deze λ-expressie te bereken: we passen simpelweg de $\mathbf{Y}$ combinator toe op deze λ-expressie: $(\mathbf{Y})M$. We zijn nu klaar om recursie te definïeren. Beschouw het volgende Scheme fragment dat de faculteitsfunctie berekent d.m.v. een recursieve functie *fac*.

```
(define fac (lambda n)
               (if (= x 0)
                   1
                   (* x (fac (- x 1)))))
```

Dit kan zonder veel moeite omgezet worden naar een λ-expressie:

$$\mathbf{fac} \equiv \lambda n.(((\mathbf{if})(\mathbf{iszero})n)\mathbf{c}_1)((\mathbf{times})n)(\mathbf{fac})(\mathbf{pred})n \tag{6}$$

Nochthans is dit geen geldige definitie omdat ze circulair is: wat gedefinieerd wordt (**fac**) wordt ook gebruikt in de definitie. We zijn als het ware een oneindig lang woord aan het definiëren. De vraag is nu of we toch het effect van een recursieve definitie op een natuurlijke manier kunnen simuleren in de λ-calculus.

De oplossing bestaat er in om de identiteit ($\equiv$) in (6) te vervangen door β-gelijkheid ($=_\beta$):

$$\mathbf{fac} =_\beta \lambda n.(((\mathbf{if})(\mathbf{iszero})n)\mathbf{c}_1)((\mathbf{times})n)(\mathbf{fac})(\mathbf{pred})n \tag{7}$$

We zoeken dus een λ-expressie **fac** die zodanig is dat (7) geldt. Echter, als (7) geldt, en we maken gebruik van het feit dat voor een willekeurige λ-expressie M geldt dat $\lambda x.(M)x =_\beta \underline{(\lambda f.\lambda x.(\underline{f})x)}\underline{M}$, dan moet ook:

$$\mathbf{fac} =_\beta \underline{(\lambda f.}\lambda n.(((\mathbf{if})(\mathbf{iszero})n)\mathbf{c}_1)((\mathbf{times})n)(\underline{f})(\mathbf{pred})n)\underline{\mathbf{fac}} \tag{8}$$

gelden.

Mits we nu in (8) de volgende vervanging doorvoeren (en merk op dat (9) niet circulair is!):

$$\mathbf{FAC} \equiv \lambda f.\lambda n.(((\mathbf{if})(\mathbf{iszero})n)\mathbf{c}_1)((\mathbf{times})n)(f)(\mathbf{pred})n \tag{9}$$

dan kan (8) met behulp van (9) herschreven worden als:

$$\mathbf{fac} =_\beta (\mathbf{FAC})\mathbf{fac}$$

M.a.w. de gezochte λ-expressie **fac** is een fixpunt van de functie **FAC**!! Zoals eerder reeds vermeld hebben we een manier om, gegeven een λ-expressie (in dit geval, **FAC**), een fixpunt voor deze λ-expressie te berekenen, nl. door gebruik te maken van de **Y** combinator, nl.

$$\mathbf{fac} \equiv (\mathbf{Y})\mathbf{FAC}$$

We zijn er dus in geslaagd om **fac**, een recursieve functie, te definïeren in de λ-calculus!

Het volgende voorbeeld illustreert hoe een recursieve functie die d.m.v. de **Y** operator werd gedefinieerd kan toegepast worden.

Voorbeeld 9 We definieren **fac** en $W_{\mathbf{FAC}}$ door

$$\begin{aligned}
\mathbf{FAC} &\equiv \lambda f.\lambda n.(((\mathbf{if})(\mathbf{iszero})n)\mathbf{c}_1)((\mathbf{times})n)(f)(\mathbf{pred})n \\
\mathbf{Y} &\equiv \lambda f.(\lambda x.(f)(x)x)\lambda x.(f)(x)x \\
\mathbf{fac} &\equiv (\mathbf{Y})\mathbf{FAC} \\
W_{\mathbf{FAC}} &\equiv \lambda x.(\mathbf{FAC})(x)x
\end{aligned}$$

Dan geldt

$$\begin{aligned}
(\mathbf{fac})\mathbf{c}_0 &\equiv ((\mathbf{Y})\mathbf{FAC})\mathbf{c}_0 \\
&\equiv ((\lambda f.(\lambda x.(f)(x)x)\lambda x.(f)(x)x)\mathbf{FAC})\mathbf{c}_0 \\
&=_\beta \underline{((\lambda x.(\mathbf{FAC})(x)x)\lambda x.(\mathbf{FAC})(x)x)}\mathbf{c}_0 \\
&\equiv \underline{((\lambda x.(\mathbf{FAC})(x)x)W_{\mathbf{FAC}})}\mathbf{c}_0 \\
&=_\beta ((\mathbf{FAC})(W_{\mathbf{FAC}})W_{\mathbf{FAC}})\mathbf{c}_0 \\
&\equiv \underline{((\lambda f.\lambda n.(((\mathbf{if})(\mathbf{iszero})n)\mathbf{c}_1)((\mathbf{times})n)(f)(\mathbf{pred})n)(W_{\mathbf{FAC}})W_{\mathbf{FAC}})}\mathbf{c}_0 \\
&=_\beta \underline{(\lambda n.(((\mathbf{if})(\mathbf{iszero})n)\mathbf{c}_1)((\mathbf{times})n)((W_{\mathbf{FAC}})W_{\mathbf{FAC}})(\mathbf{pred})n)\mathbf{c}_0} \\
&=_\beta (((\mathbf{if})\underline{(\mathbf{iszero})\mathbf{c}_0})\mathbf{c}_1)((\mathbf{times})\mathbf{c}_0)((W_{\mathbf{FAC}})W_{\mathbf{FAC}})(\mathbf{pred})\mathbf{c}_0 \\
&=_\beta (((\mathbf{if})\mathbf{true})\mathbf{c}_1)((\mathbf{times})\mathbf{c}_0)((W_{\mathbf{FAC}})W_{\mathbf{FAC}})(\mathbf{pred})\mathbf{c}_0 \\
&=_\beta ((\underline{(\mathbf{true})\mathbf{c}_1})((\mathbf{times})\mathbf{c}_0)((W_{\mathbf{FAC}})W_{\mathbf{FAC}})(\mathbf{pred})\mathbf{c}_0 \\
&=_\beta \underline{(\lambda f.\mathbf{c}_1)((\mathbf{times})\mathbf{c}_0)((W_{\mathbf{FAC}})W_{\mathbf{FAC}})(\mathbf{pred})\mathbf{c}_0} \\
&=_\beta \mathbf{c}_1
\end{aligned}$$

lll

Merk op dat we bij het toepassen van (β) telkens de "juiste" deelexpressie kozen: b.v. hebben we in

$$(((\mathbf{if})\underline{(\mathbf{iszero})\mathbf{c}_0})\mathbf{c}_1)((\mathbf{times})\mathbf{c}_0)((W_{\mathbf{FAC}})W_{\mathbf{FAC}})(\mathbf{pred})\mathbf{c}_0$$

(β) niet toegepast op $(W_{\mathbf{FAC}})W_{\mathbf{FAC}}$. Indien we dat wel gedaan hadden dan zou ons dit een nutteloze extra expansie van het voorschrift van **FAC** opgeleverd hebben.

Verder blijkt uit het voorbeeld dat we het fixpunt

$$X_{\mathbf{FAC}} \equiv (\mathbf{Y})\mathbf{FAC} \equiv (\mathbf{Y})\lambda f.\mathbf{voorschrift}_{\mathbf{FAC}}$$

kunnen gebruiken om het voorschrift $\{X_{\mathbf{FAC}}/f\}\mathbf{voorschrift}_{\mathbf{FAC}}$ van **FAC** te realizeren waarbij het fixpunt $X_{\mathbf{FAC}}$ zelf weer beschikbaar is in het voorschrift en eventueel verder kan geëxpandeerd worden ten behoeve van een recursieve toepassing.

Het bovenstaande kan veralgemeend worden.

Gevolg 1 *Beschouw een combinator F van de vorm*

$$F \equiv \lambda f.\mathbf{voorschrift}_F$$

en zijn fixpunt

$$X_F \equiv (\mathbf{Y})F$$

Dan geldt dat

$$X_F =_\beta \{X_F/f\}\mathbf{voorschrift}_F$$

Bewijs. We weten dat X_F een fixpunt is van F en dus:

$$(F)X_F =_\beta X_F$$

Nu is

$$(F)X_F \equiv (\lambda f.\mathbf{voorschrift}_F)X_F =_\beta \{X_F/f\}\mathbf{voorschrift}_F$$

Q.E.D

In het volgende voorbeeld wordt dit toegepast.

Voorbeeld 10 **FAC** en **fac** zijn gedefinieerd zoals in voorbeeld 9. Merk op dat

$$\mathbf{voorschrift_{FAC}} \equiv \lambda n.(((\mathbf{if})(\mathbf{iszero})n)\mathbf{c}_1)((\mathbf{times})n)(f)(\mathbf{pred})n$$

Er geldt:

$$\begin{array}{lll}
(\mathbf{fac})\mathbf{c}_1 & =_\beta \underline{(\lambda n.(((\mathbf{if})(\mathbf{iszero})n)\mathbf{c}_1)((\mathbf{times})n)(\mathbf{fac})(\mathbf{pred})n)\mathbf{c}_1} & \text{gevolg 1} \\
& =_\beta (((\mathbf{if})\underline{(\mathbf{iszero})\mathbf{c}_1})\mathbf{c}_1)((\mathbf{times})\mathbf{c}_1)(\mathbf{fac})(\mathbf{pred})\mathbf{c}_1 & \\
& =_\beta (((\mathbf{if})\mathbf{false})\mathbf{c}_1)((\mathbf{times})\mathbf{c}_1)(\mathbf{fac})(\mathbf{pred})\mathbf{c}_1 & \\
& =_\beta (\underline{(\mathbf{false})\mathbf{c}_1})((\mathbf{times})\mathbf{c}_1)(\mathbf{fac})(\mathbf{pred})\mathbf{c}_1 & \\
& =_\beta \underline{(\lambda f.f)((\mathbf{times})\mathbf{c}_1)(\mathbf{fac})(\mathbf{pred})\mathbf{c}_1} & \\
& =_\beta ((\mathbf{times})\mathbf{c}_1)(\mathbf{fac})\underline{(\mathbf{pred})\mathbf{c}_1} & \\
& =_\beta ((\mathbf{times})\mathbf{c}_1)\underline{(\mathbf{fac})\mathbf{c}_0} & \\
& =_\beta ((\mathbf{times})\mathbf{c}_1)\underline{(\lambda n.(((\mathbf{if})(\mathbf{iszero})n)\mathbf{c}_1)((\mathbf{times})n)(\mathbf{fac})(\mathbf{pred})n)\mathbf{c}_0} & \text{gevolg 1} \\
& =_\beta ((\mathbf{times})\mathbf{c}_1)(((\mathbf{if})\underline{(\mathbf{iszero})\mathbf{c}_0})\mathbf{c}_1)((\mathbf{times})\mathbf{c}_0)(\mathbf{fac})(\mathbf{pred})\mathbf{c}_0 & \\
& =_\beta ((\mathbf{times})\mathbf{c}_1)(((\mathbf{if})\mathbf{true})\mathbf{c}_1)((\mathbf{times})\mathbf{c}_0)(\mathbf{fac})(\mathbf{pred})\mathbf{c}_0 & \\
& =_\beta ((\mathbf{times})\mathbf{c}_1)(\underline{(\mathbf{true})\mathbf{c}_1})((\mathbf{times})\mathbf{c}_0)(\mathbf{fac})(\mathbf{pred})\mathbf{c}_0 & \\
& =_\beta ((\mathbf{times})\mathbf{c}_1)\underline{(\lambda f.\mathbf{c}_1)((\mathbf{times})\mathbf{c}_0)(\mathbf{fac})(\mathbf{pred})\mathbf{c}_0} & \\
& =_\beta ((\mathbf{times})\mathbf{c}_1)\mathbf{c}_1 & \\
& =_\beta \mathbf{c}_1 &
\end{array}$$

lll