Academiejaar 2005-2006, 12 september 2006, 08.30u

Examen: Toepassingsgerichte Formele Logica 1

1. Beschouw de propositie

$$x \Rightarrow (y \Rightarrow z) \equiv (x \Rightarrow y) \Rightarrow (x \Rightarrow z)$$

   (a) Geef een calculationeel bewijs voor bovenstaande propositie. Je mag gebruik maken van de axioma's en van stellingen 1 t.e.m. 40 zonder deze afzonderlijk te bewijzen. Indien je gebruik wenst te maken van andere stellingen, dien je die wel uitdrukkelijk te bewijzen.

   (b) Ga na wat de waarheidswaarde is van bovenstaande propositie in alle mogelijke toestanden in de gebruikelijke tweewaardige semantiek.

   (c) Leg de begrippen "tautologie" en "stelling" uit en illustreer a.d.h.v. bovenstaand voorbeeld.

2. De operator μ is gedefinieerd als $\mu = f : \mathbb{N} \to \mathbb{N} \,.\, a : \mathcal{N} \,.\, f \circ a$. Hierbij stelt $\mathcal{N}$ de verzameling voor van alle eindige rijtjes waarvan de elementen natuurlijke getallen zijn.

   (a) Formuleer een axioma dat de verzameling $\mathcal{N}$ formeel definieert.

   (b) Schrijf een Haskell–functie `mu` die het gedrag van de operator μ implementeert.

   (c) De operator δ is gedefinieerd als $\delta = a : \mathcal{N} \,.\, b : \mathcal{N} \,.\, a \mathbin{+\!\!+} b$. Geef een calculationeel bewijs voor de volgende eigenschap:

$$\delta(\mu\, l\, a)(\mu\, l\, e) = \mu\, l(\delta\, a\, e)$$

   waarbij gegeven wordt dat $a \in \square 26 \to \mathbb{N}$, $b \in \square 8 \to \mathbb{N}$ en $l \in \mathbb{N} \to \mathbb{N}$. Verder mag je gebruik maken van de volgende lemma's, zonder deze te bewijzen:

$$\#(f \circ x) = \# x$$
$$f(p\,?\,x \nmid y) = p\,?\,f\,x \nmid f\,y$$

3. (a) Zij R een relatie in X. Bewijs dat R inductie toelaat a.s.a. R goed geordend is. Je mag, zonder dit uitdrukkelijk te bewijzen, steunen op het feit dat

$$A = \emptyset \equiv \forall(x : X \,.\, \neg(x \in A))$$

   voor elke deelverzameling A van X.

   (b) Voor elk oneven natuurlijk getal n geldt dat $n^2 - 1$ deelbaar is door 8. Bewijs deze eigenschap door gebruik te maken van inductie. Formuleer zorgvuldig het inductiepredikaat en lever een calculationeel bewijs!

4. Hieronder wordt "aangetoond" dat elke symmetrische en transitieve relatie ook reflexief is. Leg uit wat er fout gaat in dit "bewijs".

   Zij R een symmetrische en transitieve relatie. Als xRy dan geldt wegens de symmetrie ook yRx. Met de transitiviteit volgt dan xRx. Dus R is reflexief.

Functioneel programmeren in Haskell

1 Programma's in lamdacalculus

Lambdacalculus kan beschouwd worden als de eerste functionele programmeertaal (ook al is lambdacalculus niet ontworpen met dit doel voor ogen). Een programma in lambdacalculus is niets anders dan een lambdaterm, meer bepaald een abstractie L. Ook de invoer M van het programma is een lambdaterm. In wat volgt zullen we zowel over L als LM spreken als "programma" (in het tweede geval bedoelen we eigenlijk het programma L met een bepaalde invoer M). Om het programma uit te voeren, brengt men de applicatie LM naar haar β–normaalvorm. Dat is dan de uitvoer van het programma. Als er geen β–normaalvorm bestaat, dan is het programma niet uitvoerbaar.

Vaak wil men meerdere invoergegevens geven aan een programma (denk b.v. aan het optellen van 2 getallen). Voor invoer M en N bekomt men dan een applicatie van de vorm LMN, die volgens onze afspraken omtrent linksassociativiteit staat voor $(LM)N$ (en dus niet voor $L(MN)$!). Men kan dit ook opvatten als het geven van invoer M aan programma L waardoor een nieuw programma LM ontstaat dat op haar beurt N als invoer krijgt. Deze manier van werken wordt curry-en genoemd naar de Amerikaanse logicus Haskell B. Curry, die belangrijke bijdragen geleverd heeft aan lambdarekenen (zoals ook de lambdacombinator Y).

Indien de β–normaalvorm van een programma ML bestaat, dan is die uniek. De uitvoer van een programma is dus ondubbelzinning bepaald. Een lambdaterm wordt teruggebracht naar haar β–normaalvorm door het toepassen van β–conversies. Dit kan vaak op verschillende manieren. Stel dat we een programma L hebben van de vorm $(\lambda z.zz)$, en de invoer M is $(\lambda x.x)y$. Dan kunnen we ofwel M in eerste instantie tot β–normaalvorm brengen, ofwel dat juist zo lang mogelijk uitstellen.

Voorbeeld 1

$$\begin{aligned}(\lambda z.zz)((\lambda x.x)y) &= \langle\beta\text{–conversie}\rangle & (\lambda z.zz)y\\ &= \langle\beta\text{–conversie}\rangle & yy\end{aligned}$$

$$\begin{aligned}(\lambda z.zz)((\lambda x.x)y) &= \langle\beta\text{–conversie}\rangle & ((\lambda x.x)y)((\lambda x.x)y)\\ &= \langle\beta\text{–conversie}\rangle & y((\lambda x.x)y)\\ &= \langle\beta\text{–conversie}\rangle & yy\end{aligned}$$

Het eindresultaat is hetzelfde (de β–normaalvorm is immers uniek) maar de hoeveelheid werk is duidelijk verschillend (2 t.o.v. 3 β–conversies). Als tweede voorbeeld beschouwen we een programma L van de vorm $\lambda xy.y$ of nog $\lambda x.\lambda y.y$ met invoergegevens M en N gegeven door respectievelijk $(\lambda k.k)p$ en $(\lambda k.k)z$. Er zijn ook nu weer verschillende volgordes voor de β–conversies mogelijk, b.v.

Voorbeeld 2

$$\begin{aligned}(\lambda x.\lambda y.y)((\lambda k.k)p)((\lambda k.k)z) &= \langle\beta\text{–conversie}\rangle & (\lambda y.y)((\lambda k.k)z)\\ &= \langle\beta\text{–conversie}\rangle & (\lambda k.k)z\\ &= \langle\beta\text{–conversie}\rangle & z\end{aligned}$$

$$\begin{aligned}(\lambda x.\lambda y.y)((\lambda k.k)p)((\lambda k.k)z) &= \langle\beta\text{–conversie}\rangle & (\lambda x.\lambda y.y)p((\lambda k.k)z)\\ &= \langle\beta\text{–conversie}\rangle & (\lambda x.\lambda y.y)pz\\ &= \langle\beta\text{–conversie}\rangle & (\lambda y.y)z\\ &= \langle\beta\text{–conversie}\rangle & z\end{aligned}$$

Opnieuw stellen we een verschil vast in het aantal β–conversies dat nodig is om de β–normaalvorm te vinden. De studie van de volgorde van β–conversies is een belangrijk deel van het onderzoek in functionele programmeertalen, enerzijds omwille van efficiëntie zoals hierboven geïllustreerd, maar anderzijds ook omdat de juiste volgorde soms cruciaal is om de β–normaalvorm te vinden! Illustreer dit voor jezelf met $(\lambda xy.y)((\lambda z.zz)(\lambda z.zz))$.

De uitwerking die het minste aantal β–conversies vergde in bovenstaande voorbeelden, noemt men "luie evaluatie" (*Engels: lazy evaluation*). Daarbij worden de invoergegevens slechts omgezet naar β–normaalvorm op het ogenblik dat men ze echt nodig heeft (zie de eerste uitwerking in voorbeeld 2), maar ook niet meer dan eens (zie de eerste uitwerking in voorbeeld 1).

2 Van lambdacalculus naar Haskell

De taal van de lambdacalculus bestaat enkel uit veranderlijken en de symbolen $\underline{\lambda}$, $\underline{.}$, $\underline{(}$ en $\underline{)}$ en is dus zeer beperkt. Toch is de lambdacalculus krachtig genoeg om lambdatermen te construeren die volwaardige computerprogramma's zijn met alle te verwachten facetten, inclusief conditionele uitvoering en recursie als controlestructuren en lijsten als datastructuur. Om het schrijfwerk te beperken en de leesbaarheid te verhogen, zijn we gaandeweg afkortingen beginnen gebruiken, met name voor lambdacombinatoren, zoals b.v. de voorstelling van natuurlijke getallen en hun bewerkingen. Zo hebben we b.v. $(\lambda x.(\lambda y.x(yy))(\lambda y.x(yy)))$ $(\lambda zxy.((\lambda x.(x(\lambda xy.x)))x)y(z((\lambda x.(x(\lambda xy.y)))x)((\lambda x.((\lambda xyz.zxy)(\lambda xy.y)x))y)))(\lambda xyz.zxy)$ $(\lambda xy.y)(\lambda x.x)(\lambda xyz.zxy)(\lambda xy.y)((\lambda xyz.zxy)(\lambda xy.y)(\lambda x.x))$ afgekort als

$$\mathsf{YH}\overline{1}\,\overline{2}$$

wat niet alleen heel wat beknopter is maar ook veel leesbaarder als je weet dat YH staat voor optelling. In bruikbare programmeertalen die gebaseerd zijn op lambdarekenen zitten dergelijke afkortingen ook ingebakken. Daardoor kan je in zo'n taal programmeren zonder dat je je eigenlijk bewust hoeft te zijn van het feit dat daarachter lambdacalculus schuil gaat.

De functionele programmeertaal die we zullen bestuderen is Haskell. Het woord "functioneel" houdt verband met het oorspronkelijk doel van lambdarekenen, nl. het bestuderen van functies. Een programma in lambdacalculus is a.h.w. een functie, de invoer is het argument en de uitvoer is dan de functiewaarde. We zullen daarom ook de benamingen Haskell–programma en Haskell–functie door elkaar gebruiken. De naam "Haskell" verwijst naar Haskell Curry. Haskell is een luie (of nog: niet–strikte) functionele programmeertaal.

In Haskell kan je op de gebruikelijke manier getallen en bewerkingen schrijven en daarvan gebruik maken in andere lambdatermen. Zo staat de lambdaterm

```
(\x -> 4*x)3
```

voor een programma, inclusief de invoer (namelijk 3). De uitvoer die je bekomt na β–conversie is 12. Merk op dat we de symbolen uit de taal van de lambdacalculus lichtjes anders noteren in Haskell. \ komt overeen met λ, terwijl het . vervangen is door ->. Bovendien kunnen we aan een zelf gebouwde lambdaterm ook een naam geven, b.v.

```
viervoud = \x -> 4*x
```

Dit is juist hetzelfde als

```
viervoud x = 4*x
```

In de praktijk zal men de laatste notatie veel vaker gebruiken omdat die beter overeenkomt met het begrip functie zoals we dat kennen uit de wiskunde. Zo verdwijnt ogenschijnlijk het symbool λ. Men hoeft in principe niets van lambdacalculus af te weten om te kunnen programmeren in Haskell. Voor voorwaardelijke uitvoering kunnen we in Haskell gebruik maken van een if–then–else constructie zoals

```
plus x y = if (x == 0) then y else plus (x-1) (y+1)
```

maar eveneens van het krachtige mechanisme van **patroonherkenning**.

```
som 0 y = y
som x y = plus2 (x-1) (y+1)
```

De functiedefinities worden in volgorde van voorkomen afgelopen. Wanneer het eerste van de twee op te tellen getallen 0 is, wordt in bovenstaand voorbeeld de eerste definitie gekozen, in alle andere gevallen de tweede. Bovenstaande voorbeelden illustreren ook meteen het gebruik van recursie.

3 Lijsten in Haskell

De kracht van een functionele programmeertaal komt pas goed tot uiting wanneer we lijsten gebruiken. De lege lijst wordt voorgesteld door `[]`, een lijst met elementen 1, 2 en 3 als `[1,2,3]`. Haskell laat ook toe om lijsten te bouwen zonder dat we alle elementen één voor één moeten opsommen. Daartoe maken we gebruik van een lijstgenerator, b.v. `[1..10]` staat voor `[1,2,3,4,5,6,7,8,9,10]`. Concatenatie (aan elkaar plakken) van lijsten gebeurt als volgt

```
[1,2,3] ++ [4,5]
```

wat de lijst `[1,2,3,4,5]` oplevert. Tabel 1 geeft een overzicht van deze en andere bewerkingen op lijsten in Haskell (probeer ze allemaal minstens één keer uit).

Lijstcomprehensies Als we niet geïnteresseerd zijn in de getallen van 1 tot 10 maar wel in hun kwadraten, dan doen we dat als volgt

```
[n*n | n <- [1..10]]
```

wat `[1,4,9,16,25,36,49,64,81,100]` oplevert. Als we enkel geinteresseerd zijn in de kwadraten van de even getallen tussen 1 en 10, dan gebruiken we

	omschrijving
`[m..n]`	lijst met getallen van m tot n
`a ++ b`	concatenatie
`head a`	eerste element van lijst a
`tail a`	staart van lijst a (lijst zonder kop)
`length a`	lengte van lijst a
`take n a`	lijst met eerste n elementen van lijst a
`drop n a`	lijst zonder eerste n elementen van lijst a
`reverse a`	lijst met elementen van lijst a achterste voren
`zip a b`	lijst met koppels samengesteld uit a en b, in dezelfde volgorde
`sum a`	som van elementen uit lijst a
`product a`	produkt van elementen uit lijst a

Tabel 1: Overzicht van bewerkingen met lijsten in Haskell

	omschrijving
`x + y`	som van x en y
`x - y`	verschil van x en y
`x * y`	produkt van x en y
`x / y`	deling van x door y
`x 'div' y`	gehele deling van x door y
`x 'mod' y`	rest van deling van x door y
`x == y`	x gelijk aan y
`x /= y`	x verschillend van y
`x <= y`	x kleiner dan of gelijk aan y
`x >= y`	x groter dan of gelijk aan y
`x < y`	x kleiner dan y
`x > y`	x groter dan y

Tabel 2: Bewerkingen en testen op gelijkheid en ongelijkheid in Haskell

```
[n*n | n <- [1..10], n 'mod' 2 == 0]
```

wat het verwachte `[4,16,36,64,100]` oplevert. Een uitdrukking van de vorm

[lichaam | voorwaarden]

wordt een lijstcomprehensie genoemd. Wat in het lichaam staat is wat uiteindelijk terecht komt in de lijst, tenminste als het voldoet aan de voorwaarden.

Belangrijke opmerking Het gelijkheidsteken = wordt in Haskell gebruikt voor de definitie van een functie. Om te testen op gelijkheid (b.v. van 2 getallen) wordt daarom dubbel gelijkheidsteken == gebruikt. Om ongelijkheid te testen wordt /= gebruikt (zie ook Tabel 2). De operatoren mod en div kunnen gebruikt worden maar moeten tussen backquotes geplaatst worden.

Een ander mooi voorbeeld van het gebruik van lijstcomprehensies is de volgende functie voor het sorteren van de elementen in een lijst.

```
sort [] = []
sort (x:l) = sort [y | y <- l, y <= x] ++ (x: sort[y|y <- l, y > x])
```

Dit illustreert tevens hoe lijsten kunnen opgebouwd worden als (x:l) waarbij x het kopelement is en l de staart. Het is bovendien een voorbeeld van patroonherkenning. Een ander voorbeeld is een functie die het aantal positieve getallen in een lijst telt.

```
posit [] = 0
posit (a:l) = if (a >= 0) then 1 + posit l else posit l
```

We kunnen dit zelfs nog bondiger programmeren door gebruik te maken van een lijstcomprehensie.

```
posit2 l = length [x|x <- l, x >= 0]
```

4 Hugs

Programma's schrijven in de functionele programmeertaal Haskell is één zaak; we willen ze uiteraard ook kunnen uitvoeren. Daarvoor maken we gebruik van Hugs[1]. Hugs is een interpreter voor Haskell; d.w.z. dat Hugs programmacode in Haskell lijn per lijn leest en uitvoert. Hugs kan zowel code lezen die ingetikt wordt aan de prompt als uit een bestand. Zo kunnen we b.v. aan de prompt intikken:

```
Prelude> 2 + 40
42
Prelude> reverse "Hugs is cool"
"looc si sguH"
Prelude> (\x -> x + x + x)2
6
```

Hierbij is Prelude> de prompt. Het woord "prelude" verwijst naar een bestand met basisdefinities dat steeds wordt ingeladen wanneer Hugs opgestart wordt. Wat daarna volgt, is wat we als gebruiker zelf intikken; het gaat hier reeds over (zij het bescheiden) programma's in Haskell. Wanneer we op Enter drukken, antwoordt Hugs telkens met de uitvoer van het programma. Behalve programma's kunnen we aan de Hugs-prompt ook enkele andere commando's intikken. Deze commando's behoren niet tot de programmeertaal Haskell en beginnen daarom steeds met een :. De belangrijkste om te onthouden is

```
:?
```

omdat dit commando ons een overzicht geeft van alle beschikbare commando's waaronder

```
:load <filenames>    load modules from specified files
:edit <filename>     edit file
:?                   display this list of commands
:cd dir              change directory
:quit                exit Hugs interpreter
```

[1] Versies van Hugs98 voor alle courante besturingssystemen zijn vrij beschikbaar op http://www.haskell.org/hugs/

Door b.v. in te tikken

```
:edit practicum.hs
```

start een tekstverwerker (editor) op in het (nieuw aangemaakte) bestand practicum.hs. Hierin kunnen we Haskell-functies of Haskell-programma's schrijven. Daarna bewaren we dit bestand, sluiten het af en tikken aan de prompt

```
:load practicum.hs
```

zodat we de zelf geschreven functies nu ook kunnen gebruiken.

Axioma's van de propositiecalculus

Axioma 1	Associativiteit van $\equiv$	$((x \equiv y) \equiv z) \equiv (x \equiv (y \equiv z))$
Axioma 2	Symmetrie van $\equiv$	$x \equiv y \equiv y \equiv x$
Axioma 3	Definitie van 1	$1 \equiv y \equiv y$
Axioma 4	Definitie van 0	$0 \equiv \neg 1$
Axioma 5	Distributiviteit van $\neg$ t.o.v. $\equiv$	$\neg(x \equiv y) \equiv \neg x \equiv y$
Axioma 6	Definitie van $\not\equiv$	$(x \not\equiv y) \equiv \neg(x \equiv y)$
Axioma 7	Symmetrie van $\vee$	$x \vee y \equiv y \vee x$
Axioma 8	Associativiteit van $\vee$	$(x \vee y) \vee z \equiv x \vee (y \vee z)$
Axioma 9	Idempotentie van $\vee$	$x \vee x \equiv x$
Axioma 10	Distributiviteit van $\vee$ t.o.v. $\equiv$	$x \vee (y \equiv z) \equiv x \vee y \equiv x \vee z$
Axioma 11	Uitgesloten derde	$x \vee \neg x \equiv 1$
Axioma 12	Gouden regel	$x \wedge y \equiv x \equiv y \equiv x \vee y$
Axioma 13	Definitie van implicatie	$x \Rightarrow y \equiv x \vee y \equiv y$
Axioma 14	Gevolg	$x \Leftarrow y \equiv y \Rightarrow x$
Axioma 15		$(p \equiv q) \;\Rightarrow\; (r[v := p] \equiv r[v := q])$

Stellingen uit de propositiecalculus

Stelling 1			$x \equiv x \equiv y \equiv y$
Stelling 2			1
Stelling 3	Reflexiviteit van $\equiv$		$x \equiv x$
Stelling 4			$y \equiv 1 \equiv y$
Stelling 5			$\neg x \equiv y \equiv x \equiv \neg y$
Stelling 6	Dubbele negatie		$\neg\neg x \equiv x$
Stelling 7	Negatie van 0		$\neg 0 \equiv 1$
Stelling 8			$(x \not\equiv y) \equiv \neg x \equiv y$
Stelling 9			$\neg x \equiv x \equiv 0$
Stelling 10	Symmetrie van $\not\equiv$		$(x \not\equiv y) \equiv (y \not\equiv x)$
Stelling 11	Associativiteit van $\not\equiv$		$((x \not\equiv y) \not\equiv z) \equiv (x \not\equiv (y \not\equiv z))$
Stelling 12	Onderlinge associativiteit		$((x \not\equiv y) \equiv z) \equiv (x \not\equiv (y \equiv z))$
Stelling 13	Onderlinge verwisselbaarheid		$x \not\equiv y \equiv z \equiv x \equiv y \not\equiv z$
Stelling 14	Opslorpend element van $\vee$		$x \vee 1 \equiv 1$
Stelling 15	Eenheidselement van $\vee$		$x \vee 0 \equiv x$
Stelling 16	Distributiviteit van $\vee$ t.o.v. $\vee$		$x \vee (y \vee z) \equiv (x \vee y) \vee (x \vee z)$
Stelling 17			$x \vee y \equiv x \vee \neg y \equiv x$
Stelling 18	Symmetrie van $\wedge$		$x \wedge y \equiv y \wedge x$
Stelling 19	Associativiteit van $\wedge$		$(x \wedge y) \wedge z \equiv x \wedge (y \wedge z)$
Stelling 20	Idempotentie van $\wedge$		$x \wedge x \equiv x$
Stelling 21	Eenheidselement van $\wedge$		$x \wedge 1 \equiv x$
Stelling 22	Opslorpend element van $\wedge$		$x \wedge 0 \equiv 0$
Stelling 23	Distributiviteit van $\wedge$ t.o.v. $\wedge$		$x \wedge (y \wedge z) \equiv (x \wedge y) \wedge (x \wedge z)$
Stelling 24	Contradictie		$x \wedge \neg x \equiv 0$
Stelling 25	Absorptie	(a)	$x \wedge (x \vee y) \equiv x$
		(b)	$x \vee (x \wedge y) \equiv x$
Stelling 26	Absorptie	(a)	$x \wedge (\neg x \vee y) \equiv x \wedge y$
		(b)	$x \vee (\neg x \wedge y) \equiv x \vee y$
Stelling 27	Distributiviteit van $\vee$ t.o.v. $\wedge$		$x \vee (y \wedge z) \equiv (x \vee y) \wedge (x \vee z)$
Stelling 28	Distributiviteit van $\wedge$ t.o.v. $\vee$		$x \wedge (y \vee z) \equiv (x \wedge y) \vee (x \wedge z)$
Stelling 29	De Morgan	(a)	$\neg(x \wedge y) \equiv \neg x \vee \neg y$
		(b)	$\neg(x \vee y) \equiv \neg x \wedge \neg y$
Stelling 30			$x \wedge y \equiv x \wedge \neg y \equiv \neg x$

Stelling 31			$x \wedge (y \equiv z) \equiv x \wedge y \equiv x \wedge z \equiv x$
Stelling 32			$x \wedge (y \equiv x) \equiv x \wedge y$
Stelling 33	Vervanging		$(x \equiv y) \wedge (z \equiv x) \equiv (x \equiv y) \wedge (z \equiv y)$
Stelling 34			$x \equiv y \equiv (x \wedge y) \vee (\neg x \wedge \neg y)$
Stelling 35	Exclusieve of		$x \not\equiv y \equiv (\neg x \wedge y) \vee (x \wedge \neg y)$
Stelling 36	Alternatieve definitie van $\Rightarrow$		$x \Rightarrow y \equiv \neg x \vee y$
Stelling 37	Alternatieve definitie van $\Rightarrow$		$x \Rightarrow y \equiv x \wedge y \equiv x$
Stelling 38	Contrapositie		$x \Rightarrow y \equiv \neg y \Rightarrow \neg x$
Stelling 39			$x \Rightarrow (y \equiv z) \equiv x \wedge y \equiv x \wedge z$
Stelling 40	Distributiviteit van $\Rightarrow$ t.o.v. $\equiv$		$x \Rightarrow (y \equiv z) \equiv x \Rightarrow y \equiv x \Rightarrow z$
Stelling 41			$x \Rightarrow (y \Rightarrow z) \equiv (x \Rightarrow y) \Rightarrow (x \Rightarrow z)$
Stelling 42	Aftakking		$x \wedge y \Rightarrow z \equiv x \Rightarrow (y \Rightarrow z)$
Stelling 43			$x \wedge (x \Rightarrow y) \equiv x \wedge y$
Stelling 44			$x \wedge (y \Rightarrow x) \equiv x$
Stelling 45			$x \vee (x \Rightarrow y) \equiv 1$
Stelling 46			$x \vee (y \Rightarrow x) \equiv y \Rightarrow x$
Stelling 47			$x \vee y \Rightarrow x \wedge y \equiv x \equiv y$
Stelling 48	Reflexiviteit van $\Rightarrow$		$x \Rightarrow x \equiv 1$
Stelling 49			$x \Rightarrow 1 \equiv 1$
Stelling 50			$1 \Rightarrow x \equiv x$
Stelling 51			$x \Rightarrow 0 \equiv \neg x$
Stelling 52			$0 \Rightarrow x \equiv 1$
Stelling 53	Verzwakken	(a)	$x \Rightarrow x \vee y$
		(b)	$x \wedge y \Rightarrow x$
		(c)	$x \wedge y \Rightarrow x \vee y$
		(d)	$x \vee (y \wedge z) \Rightarrow x \vee y$
		(e)	$x \wedge y \Rightarrow x \wedge (y \vee z)$
Stelling 54	Modus ponens		$x \wedge (x \Rightarrow y) \Rightarrow y$
Stelling 55	Gevallenonderzoek		$(x \Rightarrow z) \wedge (y \Rightarrow z) \equiv (x \vee y) \Rightarrow z$
Stelling 56	Gevallenonderzoek		$(x \Rightarrow z) \wedge (\neg x \Rightarrow z) \equiv z$
Stelling 57	Wederzijdse implicatie		$(x \Rightarrow y) \wedge (y \Rightarrow x) \equiv x \equiv y$
Stelling 58	Antisymmetrie		$(x \Rightarrow y) \wedge (y \Rightarrow x) \Rightarrow (x \equiv y)$
Stelling 59	Transitiviteit	(a)	$(x \Rightarrow y) \wedge (y \Rightarrow z) \Rightarrow (x \Rightarrow z)$
		(b)	$(x \equiv y) \wedge (y \Rightarrow z) \Rightarrow (x \Rightarrow z)$
		(c)	$(x \Rightarrow y) \wedge (y \equiv z) \Rightarrow (x \Rightarrow z)$

Stelling 60	Vervang p door q	(a)	$(p \equiv q) \wedge r[v := p] \;\equiv\; (p \equiv q) \wedge r[v := q]$
		(b)	$(p \equiv q) \Rightarrow r[v := p] \;\equiv\; (p \equiv q) \Rightarrow r[v := q]$
		(c)	$s \wedge (p \equiv q) \Rightarrow r[v := p] \;\equiv\; s \wedge (p \equiv q) \Rightarrow r[v := q]$
Stelling 61	Vervang door 1	(a)	$x \Rightarrow r[v := x] \;\equiv\; x \Rightarrow r[v := 1]$
		(b)	$y \wedge x \Rightarrow r[v := x] \;\equiv\; y \wedge x \Rightarrow r[v := 1]$
Stelling 62	Vervang door 0	(a)	$r[v := x] \Rightarrow x \;\equiv\; r[v := 0] \Rightarrow x$
		(b)	$r[v := x] \Rightarrow x \vee y \;\equiv\; r[v := 0] \Rightarrow x \vee y$
Stelling 63	Vervang door 1		$x \wedge r[v := x] \;\equiv\; x \wedge r[v := 1]$
Stelling 64	Vervang door 0		$x \vee r[v := x] \;\equiv\; x \vee r[v := 0]$