d $M, w \models \Box\varphi \quad \Leftrightarrow$ voor alle $w' \in W$ geldt: als Rww', dan $M, w' \models \varphi$
e $M, w \models \Diamond\varphi \quad \Leftrightarrow$ er is een $w' \in W$ zodat Rww' en $M, w' \models \varphi$

Als duidelijk is over welke M het gaat, schrijven we $w \models \varphi$.

Verband tussen $\Box$ en $\Diamond$

Met deze waarheidsdefinitie in de hand kunnen we nu het volgende verband tussen $\Box$ en $\Diamond$ bewijzen:

BEWERING 13.1

$M, w \models \Diamond\varphi \Leftrightarrow M, w \models \neg\Box\neg\varphi$

Bewijs

$M, w \models \Diamond\varphi$
$\Leftrightarrow$ er is een $w' \in W$ zodat Rww' en $M, w' \models \varphi$
$\Leftrightarrow$ niet voor alle $w' \in W$ geldt: als Rww' dan $M, w' \not\models \varphi$
$\Leftrightarrow$ niet voor alle $w' \in W$ geldt: als Rww' dan $M, w' \models \neg\varphi$
$\Leftrightarrow M, w \not\models \Box\neg\varphi$
$\Leftrightarrow M, w \models \neg\Box\neg\varphi$ □

In het vervolg kunnen we er daarom mee volstaan om begrippen, regels en stellingen alleen voor de $\Box$–operator te definiëren en te bewijzen.

De waarheidsdefinitie geeft aan hoe de waarheid van een formule 'lokaal' in een *wereld* bepaald wordt. Daarnaast definiëren we globale waarheid van een formule in een *model*:

DEFINITIE 13.5

Waarheid in model
Laat $M = \langle W, R, V\rangle$ een mogelijke–wereldenmodel zijn en φ een formule:

$M \models \varphi$ ('φ is waar in M') $\Leftrightarrow$ voor elke $w \in W$ geldt: $M, w \models \varphi$.

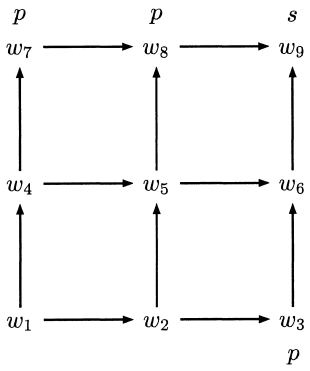
Voorbeeld 13.3

In het model M uit voorbeeld 13.2 geldt onder meer:

- $w_1 \models \Box p$ vanuit w_1 zijn w_1 zelf en w_3 toegankelijk en er geldt $w_1 \models p$ en $w_3 \models p$
- $w_2 \models \Box p$ in elke wereld die vanuit w_2 toegankelijk is (en dat is alleen w_3), is p waar
- $w_3 \models \Box p$ in elke wereld die vanuit w_3 toegankelijk is (en dat zijn er geen), is p waar
- $M \models \Box p$ dit volgt uit de vorige drie punten
- $M \not\models \Diamond p$ want $w_3 \not\models \Diamond p$
- $w_1 \models \Diamond\Box q$ er geldt Rw_1w_3 en $w_3 \models \Box q$
- $w_1 \not\models \Box\Diamond q$ want Rw_1w_3 en $w_3 \not\models \Diamond q$

Voorbeeld 13.4

In een schateiland zijn negen punten gemarkeerd: 0, 1, ..., 9. Vanuit elk punt zijn alleen het oostelijke en het noordelijke buurpunt direct toegankelijk. In punt 9 bevindt zich de schat, maar er zijn piraten in de punten 3, 7 en 8. Met modale logica kunnen we dit modelleren. De werelden w_i staan voor de punten i van het eiland, en zijn verbonden door de toegankelijkheidsrelatie. (De intuïtie bij 'mogelijke wereld' blijft beter bewaard, als we een wereld w_i laten staan voor de toestand van het eiland vanuit het gezichtspunt van een schatzoeker die punt i bereikt heeft.) Atoom s staat voor de aanwezigheid van de schat, en atoom p voor die van piraten. In de figuur geven we alleen aan waar s en p waar zijn, niet waar ze niet waar zijn.



Vervolgens kunnen we ons bijvoorbeeld afvragen vanuit welke punten we de schat in één stap kunnen bereiken, ofwel: voor welke wereld w geldt dat $w \vDash \Diamond s$? Hieraan voldoen w_6 en w_8. De werelden waarin geldt dat $\Diamond \Box p$ zijn w_4, w_6 (want $w_9 \vDash \Box p$) en w_8. Aan de hand van dit voorbeeld kunnen we ook het belang van het bereik van de modale operatoren illustreren: er geldt bijvoorbeeld $w_5 \vDash p \rightarrow \Box s$, maar $w_5 \nvDash \Box(p \rightarrow s)$. Een eigenschap van dit model is dat er voor elk punt een formule te vinden is die alleen in dat betreffende punt waar is en elders niet, zie opgave 13.3.

Voor elk van de werelden uit voorbeeld 13.4 geldt: als $w \vDash \Diamond p \vee \Diamond s$, dan geldt ook dat $w \vDash \Diamond (p \vee s)$. Dit geldt niet alleen voor dit model, maar voor *elk* mogelijke-wereldenmodel. Veronderstel namelijk dat M, $w \vDash \Diamond p \vee \Diamond s$, dan volgt uit de definitie dat M, $w \vDash \Diamond p$ of M, $w \vDash \Diamond s$, en dus is er een w' te vinden waarvoor geldt dat Rww' en M, $w' \vDash p$ of waarvoor geldt Rww' en M, $w' \vDash s$. Er bestaat dus zeker een w' zodat

$M, w' \models p \vee s$, waaruit volgt dat $M, w \models \Diamond (p \vee s)$. In de propositielogica en de predikaatlogica spraken we in dergelijke situaties van een 'geldig gevolg'. We definiëren dit begrip nu ook voor modale logica.

DEFINITIE 13.6

Geldig gevolg
Zij Σ een verzameling modale formules en ψ een modale formule:

$\Sigma \models \psi$ ('ψ is een geldig gevolg van Σ')
$\Leftrightarrow$ voor elk model M en wereld w, als $M, w \models \Sigma$, dan $M, w \models \psi$

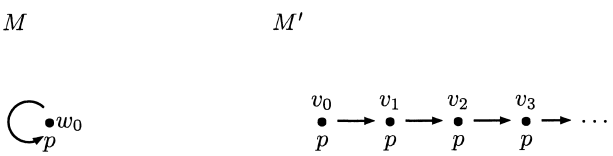
Geldig gevolg in de modale logica is net als in de predikaatlogica een complexer begrip dan in de propositielogica. Er zijn immers oneindig veel mogelijke–wereldenmodellen en voor elk model moet in principe gecontroleerd worden of de voorwaarde in definitie 13.6 geldt. Er bestaat voor de modale logica ook een eindige methode om geldigheid te testen: net als in de propositie– en predikaatlogica valt ook voor modale logica met een *semantisch tableau* te testen of $\Sigma \models \psi$. Hoe dit precies werkt zullen we in dit boek niet behandelen. Met een modaal semantisch tableau in een *eindig* aantal stappen is na te gaan of een gegeven formule al dan niet een geldig gevolg is van een formule-verzameling. Daarom is modale geldigheid *beslisbaar*. Er bestaan diverse implementaties van semantische tableaus voor modale logica waarmee automatisch gecontroleerd kan worden of $\Sigma \models \psi$.

13.4 BISIMULATIES

Van twee verschillende modellen kunnen we ons afvragen of ze in zekere zin equivalent zijn. De vraag naar equivalentie van verschillende structuren is een vraag die niet alleen in de wiskunde of de logica maar bijvoorbeeld ook in de informatica regelmatig gesteld wordt.

Voorbeeld 13.5

Neem bijvoorbeeld twee verschillende beschrijvingen van het gedrag van een tikkende klok. De wijzerstand laten we buiten beschouwing, het enige waar we in geïnteresseerd zijn is het uitvoeren van de 'tik'. Het is daarom redelijk om aan te nemen dat door het uitvoeren van een tik de toestand van de klok niet wijzigt. We kunnen dit op de volgende twee manieren modelleren:



Het linkermodel, M, bestaat uit één wereld met de eigenschap p die de toestand van de klok beschrijft, en een relatie R die de toestand voor en na uitvoeren van een tik verbindt. Omdat in dit voorbeeld de toestand na uitvoeren van de tik niet wijzigt, hebben we in het linkermodel de toestand voor en na uitvoeren van de tik laten samenvallen. In het rechtermodel, M', wordt de toestand na elke tik als een nieuwe wereld omschreven, en heeft elke wereld eigenschap p. Dit is als het ware de oneindig voortlopende tijd. Dat beide modellen dezelfde beschrijving modelleren, kunnen we ook als volgt inzien: wereld w_0 in M en wereld v_0 in M' maken precies dezelfde (modaal-)logische formules waar. We noemen deze werelden *modaal equivalent*.

DEFINITIE 13.7

Modale equivalentie
Wereld w in M en wereld v in M' heten modaal equivalent als voor elke modale formule φ geldt:
$M, w \vDash \varphi \Leftrightarrow M', v \vDash \varphi$

Gegeven twee modellen is het niet altijd zo eenvoudig als in voorbeeld 13.5 om te controleren of twee werelden w en v uit deze modellen modaal equivalent zijn. In ieder geval zullen in beide werelden dezelfde atomaire formules waar moeten zijn. Is dit het geval, dan maken deze werelden ook dezelfde propositielogische formules waar. Om ervoor te zorgen dat beide werelden ook dezelfde modale formules waar maken, moet de structuur van beide modellen tot op zekere hoogte overeenkomen. Als vanuit w een wereld w' met een (in de logica te beschrijven) eigenschap ψ toegankelijk is, dan zal vanuit v er ook een relatie met een wereld v' moeten zijn waar ψ waar is. Omdat formules geneste modale operatoren kunnen bevatten, kunnen we ons niet beperken tot werelden die toegankelijk zijn vanuit w, maar moeten we vervolgens ook de werelden bekijken die toegangelijk zijn vanuit w'. Het begrip *bisimulatie* geeft een voldoende voorwaarde voor modale equivalentie, zoals we hierna zullen zien in stelling 13.1.

DEFINITIE 13.8

Bisimulatie
Een bisimulatie tussen twee mogelijke-werelden modellen $M = \langle W_M, R_M, V_M \rangle$ en $N = \langle W_N, R_N, V_N \rangle$ is een niet-lege relatie $\leftrightarrows$ tussen W_M en W_N met de volgende eigenschappen voor werelden $w \in W_N$ en $v \in W_N$:
i als $w \leftrightarrows v$ dan $V_{M,w}(p) = V_{N,v}(p)$ voor elke propositieletter p;
ii als $w \leftrightarrows v$ en wR_Mw' dan is er een $v' \in W_N$ zodat vR_Nv' en $w' \leftrightarrows v'$;
iii als $w \leftrightarrows v$ en vR_Nv' dan is er een $w' \in W_M$ zodat wR_Mw' en $w' \leftrightarrows v'$.

Voorbeeld 13.6

De relatie $\leftrightarrows = \{(w_0, v_0), (w_0, v_1), (w_0, v_2), (w_0, v_3), ...\}$ is een bisimulatie tussen de twee modellen uit voorbeeld 13.5. Aan de eerste voorwaarde is voldaan omdat alle werelden dezelfde waardering $V(p) = 1$ hebben. De tweede voorwaarde controleren we voor w_0 en een willekeurige wereld uit het rechtermodel: v_i. Er geldt dat $w_0 \leftrightarrows v_i$ en vanuit w_0 is alleen w_0 zelf toegankelijk (w_0Rw_0). Omdat v_iRv_{i+1} en $w_0 \leftrightarrows v_{i+1}$ is aan de tweede voorwaarde voldaan. De derde voorwaarde is met een vergelijkbare redenering te controleren.

Bisimulatie is een voldoende voorwaarde voor modale equivalentie.

STELLING 13.1

Als er een bisimulatie $\leftrightarrows$ is tussen de mogelijke-werelden modellen M en N, zodat voor de werelden w uit M en v uit N geldt dat $w \leftrightarrows v$, dan zijn w en v modaal equivalent.

Bewijs

Met inductie naar de opbouw van φ bewijzen we, dat voor elk tweetal werelden $w \in M$ en $v \in N$ geldt dat als $w \leftrightarrows v$, dan $w \vDash \varphi \Leftrightarrow v \vDash \varphi$.

Basisstap

Voor propositieletters volgt dit uit de definitie van het begrip bisimulatie.

Inductiestap

Veronderstel dat de bewering opgaat voor formules φ en ψ.
De behandeling van de propositielogische gevallen laten we over aan de lezer. Omdat $\Box\varphi$ logisch equivalent is met $\neg\Diamond\neg\varphi$ kunnen we volstaan met het geval $\Diamond\varphi$. (We kiezen in dit geval niet het geval $\Box\varphi$, omdat het bewijs zo iets eenvoudiger is.)
Stel dat $w \leftrightarrows v$ en $w \vDash \Diamond\varphi$. Dat betekent dat $w' \vDash \varphi$ voor zekere w' met wR_Mw'. Uit de tweede voorwaarde van de definitie van $\leftrightarrows$ volgt dat er een v' is zodat $w' \leftrightarrows v'$, en vR_Nv'. Uit $w' \leftrightarrows v'$ volgt met de inductiehypothese dat $v' \vDash \varphi$, en dus $v \vDash \Diamond\varphi$. Net zo bewijzen we dat uit $v \vDash \Diamond\varphi$ volgt dat $w \vDash \Diamond\varphi$ met behulp van de derde voorwaarde uit definitie 13.8. □

Het omgekeerde is niet algemeen geldig: als w en v modaal equivalent zijn, dan is er niet noodzakelijk een bisimulatie te vinden zodanig dat w

$\underline{\leftrightarrow} v$. Deze bisimulatie is wel te vinden in het geval w en v tot een *eindig* model behoren.

STELLING 13.2

Laten M en N *eindige* mogelijke-wereldenmodellen zijn met daarin twee modaal equivalente werelden w en v. Dan bestaat er een bisimulatie tussen M en N zodat $w \underline{\leftrightarrow} v$.

Bewijs

Zie opgave 13.8.

13.5 DE STANDAARDVERTALING

Er is een verband tussen modale operatoren en kwantoren: semantisch correspondeert een formule $\Box\varphi$ met een universele bewering over werelden en een een formule $\Diamond\varphi$ met een existentiële bewering. Dit verband kunnen we expliciet maken door middel van een vertaling van modale formules in predikaatlogische formules.

DEFINITIE 13.9

Standaardvertaling
De standaardvertaling S die aan een modale formule φ een predikaatlogische formule $S(\varphi)$ toevoegt, wordt inductief gedefinieerd door:

a $S(p) = Px$

b $S(\neg\varphi) = \neg S(\varphi)$
$S(\varphi \wedge \psi) = S(\varphi) \wedge S(\psi)$
evenzo voor de overige propositionele connectieven

c $S(\Box\varphi) = \forall y\,(Rxy \rightarrow [y/x]S(\varphi))$
$S(\Diamond\varphi) = \exists y\,(Rxy \wedge [y/x]S(\varphi))$

De standaardvertaling introduceert dus een vrije variabele x en vertaalt elke propositieletter in een predikaat, waarbij we Px kunnen lezen als propositieletter p is waar in wereld x. In feite is de standaardvertaling een soort formele notatie van de waarheidsdefinitie (definitie 13.4) voor modale logica.

Voorbeeld 13.7

De standaardvertaling van de formule $\Box p \rightarrow p$ (vergelijk voorbeeld 13.9) berekenen we als volgt:
$S(\Box p \rightarrow p)$
$= S(\Box p) \rightarrow S(p)$
$= \forall y\,(Rxy \rightarrow [y/x]S(p)) \rightarrow Px$
$= \forall y\,(Rxy \rightarrow Py) \rightarrow Px$

Om het verband tussen een modale formule en de standaardvertaling op semantisch niveau te kunnen geven, definiëren we nu bij een

mogelijke-wereldenmodel M een predikaatlogisch model M_{PL}. De manier waarop dit gebeurt ligt voor de hand: het model M_{PL} heeft als domein de werelden van M. Voor elke propositieletter p die voorkomt in M definiëren we een unair predikaat P zodanig dat $M, w \models p \Leftrightarrow M_{PL} \models P(w)$. Ten slotte definiëren we een binair predikaat R dat overeenkomt met de toegankelijkheidsrelatie op M. Dat de standaardvertaling correspondeert met de waarheidsdefinitie blijkt uit de volgende bewering, waarvan we het bewijs achterwege laten.

BEWERING 13.2

$M, w \models \varphi \Leftrightarrow M_{PL} \models [w/x]S(\varphi)$

Uit bewering 13.2 volgt dat predikaatlogische formules die equivalent zijn met de standaardvertaling van een modale formule, invariant zijn onder bisimulatie.

13.6 AFLEIDINGEN

Axiomatisch afleiden

In de modale logica kunnen ook afleidingen gemaakt worden, als alternatieve beschrijving van geldig redeneren. Eén bekend voorbeeld van zo'n afleidingssysteem is het systeem K, dat bekendstaat als de 'minimale modale logica'. Dit bevat de volgende axioma's:

K1 alle instanties van propositionele tautologieën
K2 $\Box(\varphi \rightarrow \psi) \rightarrow (\Box\varphi \rightarrow \Box\psi)$

Naast deze axioma's bevat K twee afleidingsregels:

N	Uit φ volgt $\Box\varphi$	(vernoodzakelijking, 'Necessitation')
MP	Uit φ en $\varphi \rightarrow \psi$ volgt ψ	(Modus Ponens)

Regel N zegt, dat als een formule φ axiomatisch afleidbaar is, dan is ook $\Box\varphi$ afleidbaar. 'Formule φ is in K afleidbaar' noteren we als $\vdash_K \varphi$. Net als hiervoor bij propositielogica en predikaatlogica behoort afleiden met aannames ook tot de mogelijkheden, en we schrijven dan $\Sigma \vdash_K \varphi$ voor een afleiding van φ uit aannames Σ. Hierbij is van belang dat de regel N alleen *zonder* aannames toegepast kan worden (terwijl Modus Ponens gaat als voorheen: de vereniging van beide aannameverzamelingen nemen). In het bijzonder volgt uit $p \vdash_K p$ dus *niet* met de regel N dat $p \vdash_K \Box p$. We hebben inderdaad dat $p \nvdash \Box p$.

Voorbeeld 13.8

Dit is bijvoorbeeld een stelling in het systeem K:

$\Box(p \wedge q) \rightarrow \Box p$

Een afleiding die dit bewijst:

a	$(p \wedge q) \rightarrow p$	K1
b	$((p \wedge q) \rightarrow p)$	N, a
c	$\Box((p \wedge q) \rightarrow p) \rightarrow (\Box(p \wedge q) \rightarrow \Box p)$	K2
d	$\Box(p \wedge q) \rightarrow \Box p$	MP, b, c

Tussen 'geldig gevolg' en 'axiomatische afleidbaarheid in K' bestaat het volgende verband:

STELLING 13.3

Volledigheid
Als Σ een formuleverzameling is en φ een formule, dan geldt:
$\Sigma \vdash_K \varphi \Leftrightarrow \Sigma \vDash \varphi$

Om deze stelling te bewijzen kunnen we gebruikmaken van dezelfde technieken die we toepasten in het bewijs van de volledigheidsstelling van de propositie- en predikaatlogica. We laten het bewijs hier achterwege.

In de vervolghoofdstukken zullen we het systeem K uitbreiden met axioma's die karakteristiek zijn voor bepaalde toepassingen van modale logica. Een aantal van die axioma's komen in een ander kader al wel in de volgende paragraaf aan bod.

13.7 AXIOMA'S EN FRAMES

In deze paragraaf onderzoeken we de relatie tussen axioma's en structurele eigenschappen van modellen. De motivatie hiervoor is tweeledig. Enerzijds hebben we situaties waarin we aan modellen extra eisen opleggen. Denk bijvoorbeeld aan een tijdslogica, waarbij de werelden toestanden in de tijd representeren en de toegangelijkheidsrelatie de relatie 'later dan' is. De toegangelijkheidsrelatie is in dit geval een (partiële) ordening. Zijn er nu axioma's te vinden die corresponderen met de eigenschappen van een partiële ordening? Omgekeerd kunnen we bij een bepaald axioma ook op zoek gaan naar een bijbehorende eigenschap van de modellen. In kennislogica wordt aangenomen dat kennis waar is: $\Box \varphi \rightarrow \varphi$. Is er een eigenschap van de modellen te vinden die hiermee overeenkomt?

Bij onderzoek naar structurele eigenschappen van modellen kijken we in eerste instantie alleen naar de werelden en de toegankelijkheidsrelatie tussen deze werelden.

DEFINITIE 13.10

Frame
Een *frame* $F = \langle W, R \rangle$ bestaat uit:
a een niet–lege verzameling W van mogelijke werelden;
b een toegankelijkheidsrelatie R tussen mogelijke werelden in W.

De waarheidswaarde van een formule op een frame is als volgt:

DEFINITIE 13.11

Geldigheid op een frame
Laat $F = \langle W, R \rangle$ een frame zijn en φ een formule:

$F \models \varphi$ ('φ is geldig op het frame F')
$\Leftrightarrow$ voor elk mogelijke–wereldenmodel M op F geldt $M \models \varphi$

Het belang van de notie $F \models \varphi$ is het bepalen van puur structurele modale eigenschappen. Er zijn namelijk directe verbanden tussen formules die waar zijn op een frame $F = \langle W, R \rangle$ en de toegankelijkheidsrelatie R. K is wat dit betreft nog een 'neutraal' systeem, in die zin dat elke stelling van K op elk frame geldig is (vandaar de naam minimale modale logica).

Voorbeeld 13.9
Reflexief

Als eerste axioma(schema) bekijken we $\Box\varphi \rightarrow \varphi$. Hiermee correspondeert de structurele conditie *reflexiviteit*. Een bewijs hiervan:

Bewijs

$\Leftarrow$ Op een frame F in wereld w zal deze formule ongeacht de waardering waar zijn indien w toegankelijk voor zichzelf – dat wil zeggen reflexief – is (Rww).

$\Rightarrow$ Ook het omgekeerde geldt: stel $\neg Rww$. Beschouw de waardering die een atoom p de waarde 1 geeft in alle R–opvolgers van w doch niet in w zelf: op w geldt dan wel $\Box p$ maar niet p, hetgeen de waarheid van $\Box\varphi \rightarrow \varphi$ op het frame in w weerlegt.
Bij elkaar toont dit aan dat $\Box\varphi \rightarrow \varphi$ precies geldig is op de *reflexieve* frames, dat wil zeggen, frames met een reflexieve toegankelijkheidsrelatie. □

Een dergelijke correspondentie tussen een modale formule en een klasse van frames waarin de toegankelijkheidsrelatie R een bepaalde ordening heeft, wordt als volgt gedefinieerd.

DEFINITIE 13.12

Karakteriseren van frames
Een modale formule φ *karakteriseert* een klasse $\boldsymbol{F}$ van frames als voor elk frame F geldt: $F \models \varphi \Leftrightarrow F \in \boldsymbol{F}$.

De formule $\Box\varphi \rightarrow \varphi$ karakteriseert dus de klasse van de reflexieve frames.

Voorbeeld 13.10

De klasse van de *transitieve* frames wordt gekarakteriseerd door de modale formule $\Box\varphi \rightarrow \Box\Box\varphi$.

Bewijs

$\Rightarrow$ Op een transitief frame zijn R–opvolgers van R–opvolgers van een wereld w zelf weer R–opvolgers van w. Dus zal bij iedere waardering waarheid van $\Box\varphi$ in w reeds waarheid van $\Box\Box\varphi$ impliceren.

$\Leftarrow$ Stel omgekeerd dat F een niet–transitief frame is. Dan zijn er dus werelden w_1, w_2 en w_3 zodat Rw_1w_2 en Rw_2w_3 maar niet Rw_1w_3. Kies nu een waardering V zodat *slechts* voor de R–opvolgers w van w_1 geldt: $w \models p$ voor een atoom p. Dan geldt dus $w_1 \models \Box p$. Tevens geldt echter $w_1 \not\models \Box\Box p$, want $w_2 \not\models \Box p$, want $w_3 \not\models p$. En $w_3 \not\models p$ omdat w_3 niet een R–opvolger is van w_1. Dus $w_1 \not\models \Box\varphi \rightarrow \Box\Box\varphi$, en dus $F \not\models \Box\varphi \rightarrow \Box\Box\varphi$. □

Het karakteriseren van frames is vooral nuttig in allerlei toepassingen van modale logica, waar we ons vaak op speciale modelklassen concentreren.

De verbanden tussen de modale formules en de eigenschappen van frames lijken op het eerste gezicht misschien wat uit de lucht te komen vallen. In een aantal gevallen is het echter mogelijk om vanuit een axioma een frame-eigenschap af te leiden door de modale formule eerst te vertalen in een predikaatlogische formule met behulp van de standaardvertaling.

Voorbeeld 13.11

De voorwaarde dat $\Box p \rightarrow p$ een axioma is, correspondeert met de eis dat de standaardvertaling hiervan waar is voor elk predikaat P. In voorbeeld 13.7 berekenden we dat $\forall y\,(Rxy \rightarrow Py) \rightarrow Px$ de vertaling is van $\Box p \rightarrow p$. Om uit $\forall y\,(Rxy \rightarrow Py) \rightarrow Px$ een conclusie over R te kunnen trekken, zoeken we voor P een minimaal predikaat zodanig dat $\forall y\,(Rxy \rightarrow Py)$ universeel geldig is. Hieraan is voldaan als we Pu equivalent laten zijn met Rxu. Vullen we dit in de standaardvertaling in, dan vinden we $\forall y\,(Rxy \rightarrow Rxy) \rightarrow Rxx$, waaruit volgt dat Rxx, oftewel de relatie R is reflexief.

13.8 MODALE PREDIKAATLOGICA

Als $\Box$ en $\Diamond$ toegevoegd worden aan de taal van de predikaatlogica, dan ontstaat de *modale predikaatlogica*.

Voorbeeld 13.12

- $\Diamond \exists x\, Sx$ (lees: 'mogelijk is er een x zodat Sx')
- $\Box \forall x \Diamond (Sx \wedge \exists y \Diamond Rxy)$

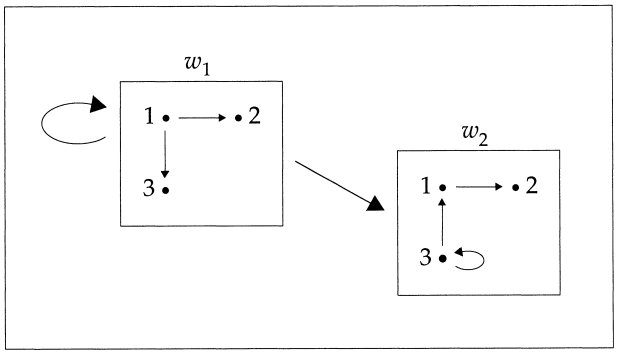
En wat concretere voorbeelden:

- $Pn \wedge \Diamond Pe$ 'Nederland is een euroland en Engeland had dit ook kunnen zijn.'
- $\forall x (\Diamond \neg Cx \rightarrow \Diamond \Box \neg Cx)$ ' Elk land dat ooit in de toekomst geen eurocenten gebruikt, zal op zeker moment de eurocent definitief afschaffen.'

De modale predikaatlogica is een natuurlijke uitbreiding van de modale propositielogica.
Het idee is voor de hand liggend. In de modale propositielogica werden met werelden in het toegankelijkheidspatroon waarderingen geassocieerd, in de modale predikaatlogica zijn dat predikaatlogische modellen in de zin van hoofdstuk 7. Elke wereld w correspondeert dus met een model $(\boldsymbol{D}, I)$. In principe kan het domein D per wereld verschillen. We geven twee voorbeelden om de semantiek van de modale predikaatlogica duidelijk te maken, zonder te streven naar totale precisie. In deze voorbeelden zullen we één vast domein aanhouden.

Voorbeeld 13.13

Beschouw het volgende model M:



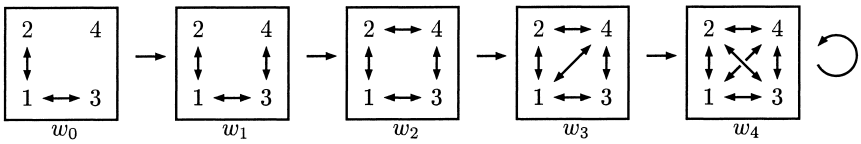
Er zijn twee mogelijke werelden die elk een predikaatlogisch model dragen. In w_1 en w_2 is sprake van een of andere relatie, zeg S, tussen de objecten 1, 2 en 3. Tussen de werelden zelf bestaat de toegankelijkheids-

relatie R. In dit model geldt dus Rw_1w_1 en Rw_1w_2.
Laten a_1, a_2 en a_3 nu constanten zijn met $I(a_i) = i$, voor $i = 1, 2, 3$. Dan geldt onder andere:

- $w_1 \models \exists x\, Sa_1x$ en $w_2 \models \exists x\, Sa_1x$, en dus $w_1 \models \Box\exists x\, Sa_1x$. Ook geldt $w_2 \models \Box\exists x\, Sa_1x$ (elke formule is noodzakelijk waar in w_2) en dus $M \models \Box\exists x\, Sa_1x$.
- $w_1 \models Sa_1a_3$, maar $w_2 \not\models Sa_1a_3$ en dus $w_1 \not\models \Box Sa_1a_3$. Wel geldt: $w_1 \models \Diamond Sa_1a_3$.
- $w_2 \models \forall x \exists y\, Syx$ dus $w_1 \models \Diamond\forall x \exists y\, Syx$, alsmede $w_1 \models \forall x \Diamond\exists y\, Syx$.
- $w_1 \models Sa_1a_2$ en $w_2 \models Sa_1a_2$ en dus $M \models Sa_1a_2$.

Voorbeeld 13.14

In dit voorbeeld modelleren we een groep individuen die elkaar leren kennen. Op tijdstip 0, afgebeeld in wereld w_0, kennen de personen 1 en 2 en de personen 1 en 3 elkaar. In de daaropvolgende werelden zijn er steeds twee personen die elkaar leren kennen. De toegankelijkheidsrelatie tussen de werelden wordt gevormd door de relatie later dan: $w_iRw_j \Leftrightarrow i < j$. In de figuur zijn alleen de pijlen tussen direct opvolgende werelden getekend.



Gaan we ervan uit dat twee personen die elkaar kennen, elkaar blijven kennen, dan geldt voor dit model $M \models \forall x\, \forall y\, (Kxy \rightarrow \Box Kxy)$. In wereld w_5 kent iedereen elkaar, en dat blijft dan ook zo. Dit wordt beschreven door de formule $\Diamond\Box\forall x\, \forall y\, Kxy$. Dit model is in zoverre niet realistisch, dat er geen rekening mee is gehouden dat er individuen geboren worden of sterven. In een algemene waarheidsdefinitie voor modale predikaatlogica waarbij de domeinen van de werelden kunnen verschillen, moet er dus rekening mee gehouden worden dat een bedeling per wereld kan verschillen. We laten deze definitie hier achterwege.

Dit besluit ons overzicht van de algemene modale logica. Er zal duidelijk zijn geworden dat dit systeem zich praktisch laat ontwikkelen naar analogie met de propositie– en predikaatlogica. Dit laatste geldt ook voor zijn meta–eigenschappen: de modale logica heeft een modeltheorie en bewijstheorie waar men alle thema's van de hoofdstukken 5 en 11 weer kan terugvinden.

13.9 OPGAVEN

13.1 Geef drie lezingen van 'Fietsers zijn noodzakelijk tweebenig'.

13.2 Laat met behulp van een tegenvoorbeeld zien dat de volgende gevolgtrekkingen niet geldig zijn:
i $p, p \rightarrow q, \Box(q \rightarrow p) \,/\, \Box p$
ii $p \vee \neg p \,/\, \Box p \vee \Box \neg p$

13.3 Geef voor iedere wereld van het model uit voorbeeld 13.4 een formule die geldig is in die wereld en onwaar in alle andere werelden.

13.4 Vertaal $\Box p \rightarrow \Box\Box p$ met behulp van de standaardvertaling in een predikaatlogische formule. Kies vervolgens een geschikt predikaat voor P zodanig dat volgt dat een frame waarop $\Box\varphi \rightarrow \Box\Box\varphi$ algemeen geldig is, transitief is.

13.5 Leid axiomatisch af :
i $\Box(p \wedge q) \leftrightarrow (\Box p \wedge \Box q)$
ii $(\Diamond p \wedge \Box q) \rightarrow \Diamond(p \wedge q)$

13.6 De zin 'Er is noodzakelijk een premier' kan op twee manieren worden weergegeven: $\Box\exists x\, Px$ en $\exists x\, \Box Px$. De eerste formule geeft de zogenaamde 'de dicto'–lezing (dictum = bewering) en de tweede de 'de re'–lezing (res = object).
Dit verschil geldt ook tussen de combinaties $\Box\, \forall$ en $\forall\Box$.
Geef modellen die de verschillen illustreren tussen de vier mogelijke combinaties $\Box\exists$, $\exists\Box$, $\Box\, \forall$ en $\forall\Box$.

* 13.7 a Geef twee verschillende modellen bestaand uit vier werelden waarop de formule $\Diamond\, \Box p \rightarrow \Box\, \Diamond p$ waar is, en twee modellen waarop deze formule onwaar is.
b Welke klasse van frames wordt gekarakteriseerd door de formule $\Diamond\, \Box p \rightarrow \Box\, \Diamond p$?
c Een relatie R is *dicht* als: $(x, y) \in R \Rightarrow$ er is een w zodat $(x, w) \in R$ en $(w, y) \in R$. Geef een modale formule die dichtheid karakteriseert.

* 13.8 Bewijs stelling 13.2:
Zij M en N *eindige* mogelijke-wereldenmodellen met daarin twee modaal equivalente werelden w en v. Dan bestaat er een bisimulatie tussen M en N zodat $w \underline{\leftrightarrow} v$.

Blok V

Logica en programmeren

Logica en berekenbaarheid

Hoofdstuk 14

Logica en berekenbaarheid

14.1 INLEIDING

In hoofdstuk 1 werden diverse verbanden besproken tussen logica en informatica. Nu we het meer zuiver logische gedeelte van dit boek achter de rug hebben valt toch wel een taakverdeling te constateren. Logische systemen als de predikaatlogica of modale logica houden zich vooral bezig met 'beschrijven' van semantische situaties, terwijl de informatica in het algemeen meer dynamisch gericht is op 'sturen' van rekenprocessen. Dit is natuurlijk slechts een globale karakteristiek: in beide gebieden zijn er ook stromingen met de omgekeerde tendens, getuige de eerdere bespreking van constructieve intuïtionistische logica, die bewijzen en berekenen sterk koppelt, of de nog te behandelen declaratieve programmeertalen, waar details van het rekenproces juist zoveel mogelijk bij het programmeren worden uitgebannen. Niettemin is het begrip berekenbaarheid (of beslisbaarheid, wat hetzelfde betekent) op diverse plaatsen in het voorgaande aan de orde geweest, met name wanneer de vraag rees of een gegeven logische methode haar gestelde problemen effectief besliste. Wij hebben over zulke kwesties voornamelijk 'impressionistisch' gesproken. Het is de moeite waard om op dit punt in het boek, alvorens we enkele contacten met de informatica nader onderzoeken, vanuit een algemeen logisch perspectief enkele opmerkingen te maken over het begrip 'effectieve berekenbaarheid'.

Berekenbaarheid

Het begrip *berekenbaarheid* middels een algoritme (ook wel *effectieve* berekenbaarheid genoemd) is in de logica sinds de jaren dertig vanuit diverse gezichtspunten onderzocht. Eerst volgen twee benaderingen waarin de klasse van berekenbare functies op een dynamische of imperatieve manier wordt gedefinieerd: via registermachines en via gestructureerd programmeren. Daarna twee declaratieve of statische benaderingen: recursieve functies en berekenbaarheid via definities.

14.2 REGISTERMACHINES

Ten eerste is er de benadering via 'Turing–machines' of iets handzamer, 'registermachines', waar dicht bij een eenvoudig machinemodel simpele instructies worden gecodeerd om objecten in registers te manipuleren. Een registermachine bestaat uit een oneindig, maar aftelbaar aantal registers (of geheugenplaatsen), aangegeven door R_1, R_2, R_3, ... Door het oneindige aantal registers en de oneindige capaciteit van elk register is een registermachine dus een geïdealiseerd computergeheugen. In elk register kan een natuurlijk getal worden opgeslagen. De inhoud van register R_i wordt aangegeven met x_i. Met betrekking tot de inhoud van registers mogen nu vier verschillende opdrachten uitgevoerd worden. Deze worden in de volgende definitie gegeven.

DEFINITIE 14.1

Een *registermachine–programma* is een *eindige* lijst genummerde opdrachten. Elke opdracht in zo'n programma heeft een van de volgende vormen, met eronder de betekenis:

a $x_i := x_i + 1$; GOTO j
'tel bij de inhoud van register R_i 1 op en ga naar de opdracht die genummerd is met j'

b $x_i := x_i - 1$; GOTO j
'trek 1 af van de inhoud van register R_i, als deze inhoud niet 0 is, en ga vervolgens naar opdracht j'

c IF $x_i = 0$ GOTO j ELSE GOTO k
'als de inhoud van register R_i 0 is, ga dan naar opdracht j, anders naar opdracht k'

d HALT
'stop met rekenen'

Omdat een registermachine–programma een eindige lijst van opdrachten is, komt er slechts een eindig aantal variabelen in voor. Een programma gebruikt dan ook alleen het eindige aantal registers waarin die variabelen zijn opgeslagen.

Voorbeeld 14.1

Een basisprogramma is het kopiëren van het ene register in een ander: Dit programma telt de inhoud van R_i op bij de inhoud van R_j en laat R_i leeg achter:

0 IF $x_i = 0$ GOTO 3 ELSE GOTO 1
1 $x_i := x_i - 1$; GOTO 2
2 $x_j := x_j + 1$; GOTO 0
3 HALT

Met iets ingewikkelder instructies kan ook tussen registers gekopieerd worden met behoud van de oorspronkelijke waarde.

Subroutine

Programma's als 'overhevelen' of 'kopiëren' van een registerinhoud zijn op zich niet zo interessant, maar worden eerder als een *subroutine* gebruikt. Om niet telkens deze programma's helemaal uit te schrijven in een programma waar ze nodig zijn, mag in dat programma ook de subroutine $K_{i,j}$ aangeroepen worden. Deze subroutine kopieert de inhoud van R_i naar R_j en laat R_i onveranderd.

Voorbeeld 14.2

Vermenigvuldigen met registermachines
Het volgende programma berekent het product $x \cdot y$ voor gegeven x en y. We gaan ervan uit dat x in R_1 zit en y in R_2. De inhoud van de overige registers is 0. De waarde $x \cdot y$ komt in R_0. We gaan de inhoud van R_2, y dus, x keer kopiëren naar R_0. Dit gebeurt door R_1 zolang dat mogelijk is met 1 te verminderen, en elke keer dat dat gebeurt de subroutine $K_{2,0}$ aan te roepen.

0 IF $x_1 = 0$ GOTO 3 ELSE GOTO 1
1 $x_1 := x_1 - 1$; GOTO 2
2 $K_{2,0}$; GOTO 0
3 HALT

De subroutine $K_{2,0}$ wordt x_1 keer aangeroepen, en dit geeft het gewenste resultaat.

Machine–berekenbaar

We introduceren nu de klasse van alle functies van $\mathbb{N}$ naar $\mathbb{N}$ die op deze wijze berekenbaar zijn:

DEFINITIE 14.2

Machine–berekenbaar
Een registermachine–programma P *berekent* een n–plaatsige functie f als P bij invoer van natuurlijke getallen $x_1, ..., x_n$ steeds waarde $f(x_1, ..., x_n)$ in een afgesproken uitvoerregister geeft. Een functie f heet *machine-berekenbaar* als er een registermachine–programma is dat f berekent.

14.3 PROGRAMMEREN IN STIP

De geschiedenis van de informatica kent een trend van laag–niveau beschrijvingen van rekenprocessen naar steeds hoger–niveau beschrijvingen, waarmee de informatica ook steeds dichter bij de logica komt te staan. Instructies voor registermachines vormen een beschrijving van het rekenproces op het lage niveau van assembleertaal. Die beschrijving

kan uiterst ondoorzichtig zijn door de vele GOTO–spronginstructies. In de lijn van de ontwikkeling binnen de informatica naar meer overzichtelijke vormen van 'gestructureerd programmeren' introduceren we nu een eenvoudige hogere *programmeertaal*. In de nu volgende taal STIP ('geSTructureerd Imperatief Programmeren') zijn programma's inductief opgebouwd, net als formules in onze logische formalismen. Dit dwingt ons om gestructureerde programma's te schrijven. Overigens is STIP in geen enkel opzicht bedoeld als realistische programmeertaal. Het is een kapstok voor voorbeelden van gestructureerd programmeren.

Definitie van STIP

We nemen een predikaatlogische taal voor de rekenkunde, met individuele constante 0 en eenplaatsige functieletters S en P, 'onmiddellijke opvolger' respectievelijk 'onmiddellijke voorganger' (met stop in 0), waarmee we termen t als gebruikelijk kunnen vormen. Als predikaat nemen we alleen de identiteit.

DEFINITIE 14.3

Syntaxis van STIP voor de rekenkunde

H1 Als x een variabele is en t een term, dan is de toekenning $x := t$ een STIP–programma.

Als π_1 en π_2 STIP–programma's zijn, dan ook:

H2	$(\pi_1 ; \pi_2)$	('opvolging')
H3	IF ε THEN π_1 ELSE π_2	('voorwaardelijke keuze')
H4	WHILE ε DO π_1	('terwijl–lus')

Boolese uitdrukking

De conditie ε staat voor een kwantorvrije predikaatlogische formule (in de gekozen taal) die in een toestand kan worden geëvalueerd tot een waarheidswaarde. Zo'n conditie heet een 'Boolese uitdrukking'.

Non–terminatie

Programma's termineren niet noodzakelijk. De WHILE–constructie kan oneindig doorlopen wanneer aan de conditie ε telkens wordt voldaan.

Interpretatie

Deze programma's moeten als volgt geïnterpreteerd worden. De variabelen worden opgevat als namen voor registers. De basisprogramma's zijn toekenningen van waarden van termen t aan variabelen. Een bedeling voor alle variabelen is dus een momentane geheugentoestand, dat wil zeggen een volledige beschrijving van alle registerinhouden. In het volgende hoofdstuk zullen we deze interpretatie uitvoeriger formuleren.

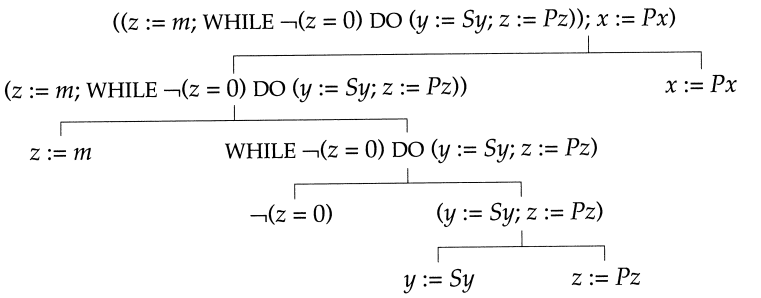
Voorbeeld 14.3

Vermenigvuldigen met STIP

Het volgende STIP–programma berekent voor getallen n en m hun product $n \cdot m$ in het y–register:

$(x := n ; (y := 0 ; \text{WHILE } \neg(x = 0) \text{ DO } ((z := m ; \text{WHILE } \neg(z = 0) \text{ DO } (y := Sy ; z := Pz)) ; x := Px)))$

De binnenste WHILE–lus telt m op bij de lopende waarde van y. Merk op dat dit programma een unieke syntactische constructieboom heeft. We geven een deel van de constructieboom:



Programma–berekenbaar

Weer is er een natuurlijke bijbehorende klasse van programmeerbare functies van $\mathbb{N}$ naar $\mathbb{N}$:

DEFINITIE 14.4

Programma–berekenbaar
Een STIP–programma π *berekent* een n–plaatsige functie f als π bij invoer van natuurlijke getallen $x_1, ..., x_n$ steeds de waarde $f(x_1, ..., x_n)$ geeft in een afgesproken uitvoervariabele.
Een functie f heet *programma–berekenbaar* als er een STIP–programma bestaat dat f berekent.

Op een informaticus die gewend is aan paginalange overzichten met voorgedefinieerde procedures in de definitie van een programmeertaal, komt STIP wellicht wat onschuldig over. In de taal STIP kunnen echter serieuze programma's geschreven worden, met ook in de praktijk veel voorkomende instructies in talen als Pascal en C.

14.4 RECURSIEVE FUNCTIES

De klasse van berekenbare functies kan ook 'statisch' of 'declaratief' wiskundig worden gedefinieerd zonder in te gaan op details van de feitelijke berekening. Eén resultaat van een dergelijke meer traditionele wijze van beschrijving van berekenbaarheid is de klasse der recursieve functies. Kenmerkend voor berekenbare functies als wiskundige objecten is hun recursieve karakter: berekening van functiewaarden grijpt terug op andere functiewaarden.

Recursieve functies

De volgende inductieve opbouw van de verzameling der *recursieve functies* weerspiegelt dit.

DEFINITIE 14.5

Recursieve functies

a De *constante functie N* met waarde 0.

b De *opvolgerfunctie S*.

c De *projectiefuncties* $\Pi^n_i(x_1, ..., x_n) = x_i$.

d Als H, G_1, ..., G_p recursieve functies zijn, waarbij H p–plaatsig is en elke G_i n–plaatsig, dan is de volgende n–plaatsige functie F ook recursief: $F(x_1, ..., x_n) = H(G_1(x_1, ..., x_n), ..., G_p(x_1, ..., x_n))$.
We zeggen dan dat F verkregen is door *compositie* van H en G_1, ..., G_n (ook wel: H met G_1, ..., G_p).

e Als H en G recursieve functies zijn, waarbij G n–plaatsig is en H $(n + 2)$–plaatsig, dan is de volgende $(n + 1)$–plaatsige functie F ook recursief:

$$F(0, x_1, ..., x_n) = G(x_1, ..., x_n)$$
$$F(Sy, x_1, ..., x_n) = H(y, F(y, x_1, ..., x_n), x_1, ..., x_n)$$

We zeggen nu dat F verkregen is door *primitieve recursie* op y.

Alvorens we de laatste constructieregel voor recursieve functies geven, eerst twee voorbeelden.

Voorbeeld 14.4

Optelling is recursief

Laat G de eenplaatsige basisfunctie Π^1_1 zijn. Laat H de compositie van S en Π^3_2 zijn, met als werking:

$$H(x, y, z) = S(\Pi^3_2(x, y, z)) = S(y) = y + 1$$

H is recursief volgens definitie 14.5d. Optelling definiëren we als volgt met primitieve recursie:

$$0 + x = x = \Pi^1_1(x) = G(x)$$
$$(Sy) + x = S(y + x) = S(\Pi^3_2(y, y + x, x)) = H(y, y + x, x)$$

Voorbeeld 14.5

Vermenigvuldigen is recursief

Op een vergelijkbare wijze als bij optelling is vermenigvuldiging primitief recursief te definiëren met $G = N$ (de constante functie met waarde nul) en $H(x, y, z) = \Pi^3_2(x, y, z) + \Pi^3_3(x, y, z)$.

De basisfuncties in a, b en c en de opbouwregels d en e leveren de klasse van de *primitief recursieve* functies. Om alle berekenbare functies te vangen is nog een opbouwregel nodig met een 'smaak' van oneindigheid:

f Als G recursief is, dan is de volgende functie F ook recursief:
$F(x_1, ..., x_n) = \mu y \,.\, (G(y, x_1, ..., x_n) = 0)$.
Hiermee wordt bedoeld: 'het kleinste getal y, indien dit bestaat, waarvoor de conditie $G(y, x_1, ..., x_n) = 0$ opgaat'. We zeggen dan dat F is verkregen door *minimalisatie*.

Voorbeeld 14.6

Stel $F(x) = \mu y \,.\, (y^2 - 3xy + 2x = 0)$.
Dan geldt: $F(0) = \mu y \,.\, (y^2 = 0) = 0$, $F(1) = \mu y \,.\, (y^2 - 3y + 2 = 0) = 1$.

Merk op dat een door minimalisatie verkregen functie niet overal gedefinieerd hoeft te zijn. Het is immers mogelijk dat er geen y is waarvoor $G(y, x_1, ..., x_n) = 0$. Zo geldt in voorbeeld 14.6 dat $F(2)$ niet gedefinieerd is. Dit is te vergelijken met een registerprogramma dat bij zekere invoer niet termineert en met een oneindig repeterende lus in een WHILE–programma.

Functioneel berekenbaar

Een recursieve functie noemen we *functioneel berekenbaar*.

14.5 BEREKENBAARHEID VIA DEFINITIES

Als laatste mogelijkheid om berekenbaarheid te beschrijven lopen we vooruit op het 'logisch programmeren' van hoofdstuk 16. Wat declaratief is kan simpel worden uitgelegd: de bedoeling is nu om rekenkundige functies te programmeren door hun definitie in predikaatlogica op te schrijven (dat wil zeggen: door ze te *declareren*), en wel zo dat het berekenen van functiewaarden neerkomt op het logisch afleiden van geldige gevolgen in de modellen van het declaratieve programma.
We illustreren dit weer aan de hand van vermenigvuldiging:

Voorbeeld 14.7

Declaratief vermenigvuldigen
Een logisch programma voor een vermenigvuldigingspredikaat:

$Maal(x, y, z)$: '$x \cdot y = z$'

schrijven we als volgt, met een hulppredikaat *Som* ($Som(x, y, z)$: '$x + y = z$') voor de optelling:

$\forall x \; Som(0, x, x)$
$\forall x \; \forall y \; \forall z \; (Som(x, y, z) \rightarrow Som(Sx, y, Sz))$
$\forall x \; Maal(0, x, 0)$
$\forall x \; \forall y \; \forall z \; ((Maal(x, y, z) \wedge Som(z, y, u)) \rightarrow Maal(Sx, y, u))$

De bewijsmethode voor resolutie die in hoofdstuk 16 wordt gepresenteerd, zal uit dit logisch programma de correcte drietallen $\langle x, y, x \cdot y \rangle$ afleiden.

Definitioneel berekenbaar

De op deze manier berekenbare rekenkundige functies noemen we de definitioneel berekenbare of kortweg *definieerbare* functies.

DEFINITIE 14.6

Definitioneel berekenbaar
Een n–plaatsige functie f heet *definitioneel berekenbaar* als er een logisch programma is voor een $(n + 1)$–plaatsig predikaat $P(x_1, ..., x_n, z)$, zodat voor waarden $x_1, ..., x_n$ slechts $z = f(x_1, ..., x_n)$ uit het programma afgeleid kan worden.

14.6 DE THESE VAN CHURCH

In het voorgaande zijn vier vormen van berekenbaarheid geïntroduceerd, die corresponderen met de voornaamste genres uit de moderne informatica, te weten machine–, programma–, functionele en definitionele berekenbaarheid.
Nu geldt het volgende opmerkelijke resultaat:

STELLING 14.1

De vier genoemde klassen van berekenbare functies vallen samen.

These van Church

Een bewijs valt buiten het bestek van dit boek. Dit samenvallen van uiteenlopende vormen van berekenbaarheid motiveert de zogenaamde 'These van Church', die zegt dat elke *intuïtief* berekenbare functie berekenbaar is in de hier omschreven technische zin. Dit is niet exact te bewijzen (daarom heet het ook een *these*), omdat er geen exacte beschrijving bestaat van intuïtieve berekenbaarheid, maar het duidt wel aan dat we waarschijnlijk het juiste algemene begrip te pakken hebben. Dit gevoel wordt nog versterkt doordat nog vele andere voorgestelde benaderingen van berekenbaarheid, waaronder ook niet–numeriek symbolische benaderingen, zoals Post–productiesystemen en representeerbaarheid in λ–calculus, ook weer door geschikte numerieke codering overeen blijken te komen met de vier samenvallende benaderingen van berekenbaarheid die we hier behandeld hebben. Het is dan ook dit begrip van berekenbaarheid dat aan dit boek ten grondslag ligt.

Berekenbaarheid en complexiteit

Een mogelijk misverstand dient hier wel terstond ontzenuwd te worden. Vanuit het licht der eeuwigheid, zou men kunnen zeggen, impliceert de voorgaande analyse dat alle bekende programmeervormen in de informatica op hetzelfde neerkomen. In de praktijk heeft deze principiële constatering echter weinig inhoud. Onder de oppervlakte van de algemene notie van berekenbaarheid schuilt namelijk een grote diversiteit aan hogere of lagere *complexiteit* van berekenbare functies, waarbij keuzen tussen verschillende programmeertalen, stijlen en representaties juist cruciaal zijn. Veel vraagstellingen in de informatica gaan over dit soort keuzen. En dus loont het ook de moeite om vanuit de logica de eigenaardigheden van verschillende stromingen binnen de informatica te bestuderen.

In dit boek zullen we daarom om te beginnen in hoofdstuk 15 een beschouwing wijden aan het imperatief programmeren in de STIP–taal, wat model kan staan voor een logische behandeling van imperatieve talen, zoals Pascal en C, in het algemeen. Vervolgens onderzoeken we in hoofdstuk 16 een stijl van declaratief programmeren, namelijk het logisch programmeren, waarvan PROLOG een bekende vertegenwoordiger is. Andere programmeerstijlen hebben we wegens plaatsgebrek buiten beschouwing gelaten. Voorbeelden hiervan zijn programmeerstijlen die bij het genoemde registermachine–programmeren aansluiten, zoals in de taal BASIC. En ook op de functionele programmeerstijl die op de wiskundige opzet der recursieve functies is gebaseerd, gaan we niet verder in. Denk bij dit laatste bijvoorbeeld aan LISP of MIRANDA of andere op de λ–calculus georiënteerde talen.
In hoofdstuk 17, het laatste hoofdstuk van dit blok, gaan we verder in op de complexiteit van logische en andere berekeningen.

14.7 OPGAVEN

14.1 Laat voor elk van de vier noties van berekenbaarheid zien hoe de faculteitsfunctie $n!$ berekend kan worden.

* 14.2 Druk compositie en minimalisatie uit in registermachine–programma's. Dat wil zeggen, als F verkregen is door compositie, geef dan een registermachine–programma dat F berekent.
Evenzo voor een functie F die verkregen is door minimalisatie.

Logica en imperatief programmeren

Hoofdstuk 15

Logica en imperatief programmeren

15.1 INLEIDING

De logische formalismen die eerder in dit boek aan de orde zijn geweest, worden in de informatica ook gebruikt bij het redeneren *over* imperatieve programma's en hun verwerking. Daarbij fungeert de gegeven semantiek van logische talen als inspiratie voor het opzetten van een semantiek voor programmeertalen. De eerste vraag daarbij is de volgende. Tot nu toe dienden logische formalismen voor de weergave van 'statische' beweringen over situaties (modellen). Hoe kunnen we ons gezichtspunt nu zo uitbreiden dat 'dynamische' programmeertekst, met opeenvolgende instructies om situaties juist te veranderen, toch ook binnen het logische kader komt te passen?
Men zou in eerste instantie kunnen denken dat de fysische activiteit binnen de computer zelf een realistische 'interpretatie' van een programma vormt ('de betekenis van een programma is wat het doet'). Dit gezichtspunt helpt echter niet bij het *ontwerp* van programma's of het toetsen van hun *correctheid*. Vandaar dat we meer abstracte, geïdealiseerde modellen hanteren om de betekenis van programma's weer te geven. Hierbij zijn overigens nog vele keuzes mogelijk, afhankelijk van het beoogde doel.
In dit hoofdstuk geven we een semantiek voor imperatieve programma's in termen van *overgangen van geheugentoestanden* van een rekenautomaat.

15.2 PROGRAMMA'S EN TOESTANDSOVERGANGEN

Beschouw om te beginnen een of ander model $M = (\boldsymbol{D}, I)$ voor een predikaatlogische taal. We denken voortaan over $\boldsymbol{D}$ als een *gegevensstructuur* waarop we kunnen rekenen. Predikaatlogische formules φ met vrije variabelen $x_1, \ldots, x_n$ kunnen vervolgens in deze situatie beschouwd worden als beweringen over momentane waarden van die variabelen, weergegeven in een *bedeling*. En aan dit laatste begrip valt nu een heel concrete computationele duiding te geven. Een bedeling was in het

voorgaande een functie die aan variabelen x, y, z, ... waarden $b(x)$, $b(y)$, $b(z)$, ... uit het domein D toekent, of in meer concrete termen, een 'vulling van de registers x, y, z, ... met gegevens'. Dat laatste is een goede omschrijving van een momentopname van een geheugentoestand in een rekenautomaat. Een rekenproces gestuurd door een programma verandert stapsgewijs waarden in registers, dat wil zeggen, de lopende bedeling.

Voorbeeld 15.1

Het eerdere STIP–programma voor vermenigvuldiging uit voorbeeld 14.3:

$(x := n\ ;\ (y := 0\ ;$ WHILE $\neg(x = 0)$ DO $((z := m\ ;$ WHILE $\neg(z = 0)$ DO $(y := Sy\ ;\ z := Pz))\ ;\ x := Px)))$

geeft de volgende overgangen te zien bij input $n = 2$ en $m = 2$ (het streepje staat hier voor een willekeurige waarde):

lopende bedeling b	$b(x)$	$b(y)$	$b(z)$
begintoestand b_0	–	–	–
b_1	2	–	–
b_2	2	0	–
b_3	2	0	2
b_4	2	1	2
b_5	2	1	1
b_6	2	2	1
b_7	2	2	0
b_8	1	2	0
b_9	1	2	2
b_{10}	1	3	2
b_{11}	1	3	1
b_{12}	1	4	1
b_{13}	1	4	0
eindtoestand b_{14}	0	4	0

De totale actie van een programma π bestaat dus uit vele achtereenvolgende 'lokale' veranderingen in de bedeling, en kan worden beschreven als: een bedeling b – de 'begintoestand' – is na een geslaagde uitvoering van π overgegaan in een andere bedeling e – de 'eindtoestand'. Anders gezegd: een programma bepaalt een overgangsrelatie tussen bedelingen.

Dit is overigens een zeer 'ruwe' beschrijving. Er wordt immers alleen naar de begin– en eindtoestand gekeken. Een rijkere semantiek ontstaat als ook andere aspecten in het spel komen, zoals de doorlopen tussenstadia ('traces'), gemaakte keuzes of gebruikte tijdspannes. Voor onze doeleinden hier volstaat echter het eenvoudigste beeld.

15.3 STIP–PROGRAMMA'S

Om de ideeën uit de vorige paragraaf concreter te kunnen uitwerken, keren we terug naar de STIP–programma's uit hoofdstuk 14, nu iets algemener omschreven.

DEFINITIE 15.1

Syntaxis van STIP
Net als in het vorige hoofdstuk, alleen nu voor predikaatlogische termen in het algemeen en niet slechts voor rekenkundige termen.

Interpretatie van STIP

STIP–programma's π worden nu van een betekenis voorzien middels de volgende inductieve toekenning van bijbehorende overgangsrelaties $T(\pi)$ in een model $M = (\boldsymbol{D}, I)$.

DEFINITIE 15.2

Semantiek van STIP–programma's

a $b_1\ T(x := t)\ b_2 \Leftrightarrow b_2 = b_1[x \mapsto V_{(\boldsymbol{D}I),b_1}(t)]$

b $b_1\ T(\pi_1\ ;\ \pi_2)\ b_2 \Leftrightarrow$ er is een bedeling b zodat $b_1\ T(\pi_1)\ b$ en $b\ T(\pi_2)\ b_2$

c $b_1\ T(\text{IF } \varepsilon \text{ THEN } \pi_1 \text{ ELSE } \pi_2)\ b_2 \Leftrightarrow (\boldsymbol{D}, I), b_1 \models \varepsilon$ en $b_1\ T(\pi_1)\ b_2$, of $(\boldsymbol{D}, I), b_1 \not\models \varepsilon$ en $b_1\ T(\pi_2)\ b_2$

d $b_1\ T(\text{WHILE } \varepsilon \text{ DO } \pi)\ b_2 \Leftrightarrow$ er is een *eindige* rij bedelingen $b_1 = b^1, \ldots, b^n = b_2$ zodat voor alle $i < n$: $(\boldsymbol{D}, I), b^i \models \varepsilon$ en $b^i\ T(\pi)\ b^{i+1}$ en $(\boldsymbol{D}, I), b_2 \not\models \varepsilon$

Voorbeeld 15.2

Semantiek van STIP
Laat $\pi = (\text{IF } x < y \text{ THEN } z := y - x \text{ ELSE } z := x - y)$. Laat verder b_1 en b_2 twee bedelingen zijn waarvoor geldt: $b_1(x) = 5$; $b_1(y) = 4$; $b_2(x) = 5$; $b_2(y) = 4$; $b_2(z) = 1$. We werken op $\boldsymbol{D} = \langle \mathbb{N}, <, - \rangle$ met voor de hand liggende I. Nu geldt:
i $(\boldsymbol{D}, I), b_1 \not\models x < y$
Voor $\pi_2 = (z := x - y)$ geldt:
ii $b_2 = b_1[z \mapsto V_{(\boldsymbol{D}I),b_1}(x - y)]$
En dit betekent volgens definitie 15.2a: $b_1\ T(\pi_2)\ b_2$.
Uit i en $b_1\ T(\pi_2)\ b_2$ volgt met 15.2c: $b_1\ T(\pi)\ b_2$.

Determinisme

Een effect van deze definitie is dat STIP–programma's *deterministisch* zijn: vanuit elke bedeling is via $T(\pi)$ hoogstens één andere bedeling bereikbaar. In het algemeen kan men zo'n semantiek voor overgangsrelaties overigens ook voor *indeterministische* programma's geven.

Partiële bereikbaarheid

Wanneer het programma niet termineert, is zelfs helemaal geen andere bedeling bereikbaar. De $T(\pi)$–relatie is dan partieel.

15.4 DE HOARE–CALCULUS VOOR CORRECTHEID

De Hoare–calculus (genoemd naar een van de grondleggers, Hoare) is wellicht de bekendste toepassing van deze semantiek en heeft betrekking op zogenaamde *correctheidsbeweringen*. Dit zijn beweringen van de vorm:

$\{\varphi\}\ \pi\{\psi\}$

De betekenis hiervan is: als voor verwerking van π de formule φ geldt in de begintoestand in een model M, dan geldt ψ na verwerking van π in de eindtoestand. De formule φ wordt de 'preconditie' genoemd, ψ de 'postconditie'. Formeel:

DEFINITIE 15.3

Correctheidsbewering
$M \models \{\varphi\}\ \pi\{\psi\}$ als voor alle b_1, b_2 met $b_1\ T(\pi)\ b_2$ geldt:
als $M, b_1 \models \varphi$, dan $M, b_2 \models \psi$.

Voorbeeld 15.3

Correctheidsbeweringen
Beschouw de volgende correctheidsbeweringen:
a $\{x = 3\}\ y := x \cdot x\ \{y = 9\}$
b $\{x = z\}\ y := x \cdot x\ \{y = z \cdot z\}$
c $\{x = 1\}\ y := x \cdot x\ \{y = 2\}$
De beweringen a en b zijn juist, c niet.

Zij π het programma $(y := S0\ ;$ WHILE $\neg(x = 0)$ DO $(y := y \cdot x\ ;\ x := Px))$.
Intuïtief valt niet moeilijk in te zien dat dit de functie *faculteit* berekent.
Beschouw nu de volgende correctheidsbeweringen:
d $\{\text{TRUE}\}\ \pi\{y = x!\}$
e $\{x = z\}\ \pi\{y = z!\}$
Bewering d is onjuist omdat de oude waarde van x in het rekenproces is verdwenen. Bewering e is juist.

Voor lange complexe programma's is het rekengedrag, weerspiegeld in correctheidsbeweringen, niet altijd doorzichtig, zodat systematische

controle urgent wordt. Er bestaan daarom diverse *bewijssystemen* om systematisch bij programma's behorende correctheidsbeweringen te bewijzen, die uitdrukken dat het programma onder zekere voorwaarden (preconditie) precies een gewenst resultaat (postconditie) bereikt.

We geven hier de zogenaamde *Hoare–axiomatiek* voor STIP–programma's, die op natuurlijke wijze de inductieve structuur van onze programmeertaal volgt:

DEFINITIE 15.4

Hoare–axiomatiek

H1 $\{[t/x]\varphi\}\ x := t\ \{\varphi\}$, mits t vrij voor x in φ

H2 uit $\{\varphi\}\ \pi_1\ \{\psi\}$ en $\{\psi\}\ \pi_2\ \{\chi\}$ volgt $\{\varphi\}\ (\pi_1\ ;\ \pi_2)\ \{\chi\}$

H3 uit $\{\varphi \wedge \varepsilon\}\ \pi_1\ \{\psi\}$ en $\{\varphi \wedge \neg\varepsilon\}\ \pi_2\ \{\psi\}$ volgt $\{\varphi\}$ IF εTHEN π_1 ELSE $\pi_2\ \{\psi\}$

H4 uit $\{\varphi \wedge \varepsilon\}\ \pi\ \{\varphi\}$ volgt $\{\varphi\}$ WHILE ε DO $\pi\ \{\neg\varepsilon \wedge \varphi\}$

H5 als $\varphi \rightarrow \varphi'$ en $\psi' \rightarrow \psi$ waar zijn op $M = (\boldsymbol{D}, I)$ dan geldt:
uit $\{\varphi'\}\ \pi\ \{\psi'\}$ volgt $\{\varphi\}\ \pi\ \{\psi\}$

De eerste drie regels zijn tamelijk voor de hand liggend. De vierde introduceert een nieuw idee: de zogenaamde '*lus–invariant* φ', een bewering die bij elke doorgang voor π vanuit een ε–toestand blijft opgaan. Vinden van geschikte lus–invarianten is een kunst bij het programmeren.
De vijfde regel zegt dat de preconditie in een correctheidsbewering versterkt en de postconditie verzwakt mag worden.

Voorbeeld 15.4

Een correctheidsbewijs

Beschouw het volgende programma π dat voor een willekeurig natuurlijk getal x de waarde $x + (x-1) + (x-2) + \ldots + 1$ berekent:

$y := 0\ ;\ z := x\ ;$ WHILE $\neg(z = 0)$ DO $(y := y + z\ ;\ z := z - 1)$

We willen de volgende correctheidsbewering bewijzen:

$\{\text{TRUE}\}\ \pi\ \{y = x \cdot (x+1)/2\}$

Om deze postconditie te bereiken nemen we voor de lus–invariant φ in H4 de formule $y + z \cdot (z+1)/2 = x \cdot (x+1)/2$. Immers, als op een bepaald moment $z = 0$ (dit is $\neg\varepsilon$) geldt, dan stopt de lus en $\varphi \wedge \neg\varepsilon$ impliceert dan $y = x \cdot (x+1)/2$, precies wat we willen.
Dat deze lus–invariant voldoet, blijkt uit de volgende twee toepassingen van H1:

$\{y + z \cdot (z+1)/2 = x \cdot (x+1)/2\}\ y := y + z\ \{y + (z-1) \cdot z/2 = x \cdot (x+1)/2\}$

$\{y + (z-1) \cdot z/2 = x \cdot (x+1)/2\}\ z := z - 1\ \{y + z \cdot (z+1)/2 = x \cdot (x+1)/2\}$

H2 geeft nu:

$\{\varphi\}\ (y := y + z\ ;\ z := z - 1)\ \{\varphi\}$

En H5 levert:

$\{\varphi \wedge \neg(z = 0)\}\ (y := y + z\ ;\ z := z - 1)\ \{\varphi\}$

Uit H4 volgt nu de WHILE–lus:

$\{\varphi\}$ (WHILE $\neg(z = 0)$ DO $y := y + z\ ;\ z := z - 1)\ \{\varphi \wedge z = 0\}$

Nu is nog twee keer H1 nodig:

$\{\text{TRUE}\}\ y := 0\ \{y = 0\}$
$\{y = 0\}\ z := x\ \{y + z \cdot (z+1)/2 = x \cdot (x+1)/2\}$

Drie keer toepassen van H2 levert nu de uiteindelijke correctheids-bewering. □

We vermelden nog enkele meer theoretische resultaten.

Correct

De Hoare–calculus is semantisch correct: alle voor een model bewijsbare correctheidsbeweringen zijn in dat model ook waar. Overigens is de overeenkomst in terminologie tussen semantisch *correct* ('sound') en *correctheids*bewering ('correct') een toevalligheid van het Nederlands.

Volledig

Omgekeerd geldt een soort volledigheidsresultaat: op voldoende 'rijke' gegevensstructuren is elke ware correctheidsbewering Hoare-bewijsbaar.

15.5 DYNAMISCHE LOGICA

Naast correctheid zijn er andere belangrijke eigenschappen van programmaverwerking, die tot nu toe buiten beeld bleven. Voorbeelden hiervan zijn 'terminatie', 'determinisme' en 'equivalentie' van programma's. Willen we ook deze verdisconteren, dan blijkt een rijker logisch systeem nodig dan de Hoare–calculus. Een goede kandidaat voor zo'n generalisering is een toepassing van de modale logica die *dynamische logica* heet. Anders dan in de Hoare–axiomatiek, kunnen we

in dynamische logica terminatie, determinisme en equivalentie van programma's beschrijven.

De dynamische logica is ontstaan vanuit de informatica. In deze logica worden mogelijke werelden geïnterpreteerd als geheugentoestanden van een computer. In zo'n toestand hebben alle variabelen een waarde, zodat deze is te vergelijken met een bedeling. Maar we gaan nu over van het *predikaatlogisch* perspectief dat we hanteerden bij de STIP-programma's naar een *modaal–logisch* perspectief.

Voorbeeld 15.5

Laat b een toestand zijn waarin $x = 0$, $y = 1$ en $z = 4$. De propositieletters p, q en r staan voor respectievelijk '$x = 0$', '$y < 0$' en '$z > 5$'. Dan geldt onder andere:

- $b \vDash p$
- $b \nvDash p \rightarrow q$
- $b \vDash p \vee r$

De overgangsrelatie $T(\pi)$ tussen geheugentoestanden die door een programma π wordt bepaald, kunnen we zien als een modale *toegankelijkheidsrelatie*. Deze wordt als volgt gedefinieerd:

$b\ T(\pi)\ e \Leftrightarrow$ vanuit begintoestand b is er een succesvolle uitvoering van programma π die eindigt in eindtoestand e

De operator $\Box$ behoeft nu enige aanpassing. Immers, wat toegankelijk is hangt nu af van het programma. Daarom moet bij $\Box$ worden aangegeven over welk programma het gaat. In plaats van $\Box$ noteren we dan $[\pi]$. Dit levert:

$b \vDash [\pi]\varphi \Leftrightarrow$ voor elke toestand e met $b\ T(\pi)\ e$ geldt $e \vDash \varphi$

Ook $\Diamond$ wordt programma–afhankelijk (notatie: $\langle\pi\rangle$):

$b \vDash \langle\pi\rangle\varphi \Leftrightarrow$ er is een toestand e zodat $b\ T(\pi)\ e$ en $e \vDash \varphi$

In plaats van $b\ T(\pi)\ e$ hadden we dus ook $R_\pi be$ kunnen schrijven. Het mogelijke–wereldenmodel waarbinnen we interpreteren is hier impliciet. Dit bestaat echter eenvoudigweg uit alle mogelijke geheugentoestanden van een computer. Met behulp van de dynamische operatoren kunnen diverse beweringen over programmagedrag geformuleerd worden.

Correctheidsbewering

De *correctheidsbeweringen* die we hiervoor reeds hebben geïntroduceerd, zijn semantisch wellicht de belangrijkste beweringsvorm in de informatica. In de dynamische logica heeft een correctheidsbewering de volgende vorm:

$$\varphi \rightarrow [\pi]\psi$$

Hierin wordt uitgedrukt dat π bij een bepaalde invoerconditie φ altijd een uitvoer geeft die voldoet aan de conditie ψ.

Determinisme

Beschouw de volgende formule:

$$\langle\pi\rangle\varphi \rightarrow [\pi]\varphi$$

Deze formule interpreteren we als volgt: als er *een* toestand e is zodat $b\ T(\pi)\ e$ en $e \vDash \varphi$, dan geldt voor *elke* toestand e' met $b\ T(\pi)\ e'$ dat $e' \vDash \varphi$. Omdat φ een willekeurige formule is, drukt $\langle\pi\rangle\varphi \rightarrow [\pi]\varphi$ uit dat π *deterministisch* is.

Terminatie

Dit laat op zich nog open dat in sommige toestanden in het geheel geen succesvolle berekening van π mogelijk is. Wanneer dit wel zo is, noemen we dit *terminatie* en dat is ook met een dynamische formule te beschrijven:

$$\langle\pi\rangle\text{TRUE}$$

Hierin staat TRUE voor de 'ware' bewering, bijvoorbeeld 1 = 1. Als $\langle\pi\rangle$TRUE waar is in zekere toestand b, dan betekent dit dat er ten minste een toestand e is waarin we terechtkomen als π wordt uitgevoerd. Maar dan is π kennelijk gestopt.

Equivalentie

We kunnen ook in een modale formule uitdrukken dat twee programma's π_1 en π_2 *equivalent* zijn:

$$[\pi_1]\varphi \leftrightarrow [\pi_2]\varphi$$

Nog complexere uitspraken zijn ook voorstelbaar, met gestapelde modaliteiten, zoals bijvoorbeeld:

$$p \rightarrow [\text{WHILE } p \text{ DO } \pi_1]\langle\pi_2\rangle p$$

Betekenis: 'als in een toestand p geldt, dan is na elke verdere toestand waarin $\neg p$ is ontstaan door herhalen van π_1, via uitvoeren van π_2 weer een p–toestand te bereiken'. Overigens is dit natuurlijk ook een – iets concretere – correctheidsbewering.

Om al dit soort beweringen te kunnen bewijzen is weer een calculus nodig waarvan de regels op een natuurlijke manier aansluiten bij de aanwezige programmastructuur. Het voordeel van de dynamische logica is nu dat we ons daarbij kunnen richten op bekende bewijssystemen uit de modale logica.

Reguliere programma's

Ter wille van de elegantie is het dan echter wel nodig over te schakelen op een iets ruimere klasse programmaconstructies dan tot nu toe in STIP is gebruikt. Daartoe definiëren we inductief de zogenaamde *reguliere programma's*:

DEFINITIE 15.5

Reguliere programma's

Basisprogramma's

a $a, b, c, \ldots$ 'atomaire acties'

b $\varphi?$ 'test of een bewering φ waar is'

Programmaopbouw

Als π_1, π_2, π reguliere programma's zijn, dan ook:

c $(\pi_1 ; \pi_2)$ 'opvolging'

d $(\pi_1 \cup \pi_2)$ 'keuze'

e π^* 'iteratie'

Het programma $\varphi?$ test of φ waar is. Zo ja, dan wordt doorgegaan met de volgende opdracht: zo nee, dan wordt afgebroken.
Het programma $(\pi_1 \cup \pi_2)$ maakt een willekeurige keuze tussen π_1 en π_2 en voert het gekozen programma uit.
Het programma π^* kiest een of ander eindig natuurlijk getal n en voert vervolgens n keer π uit.

Formules van de dynamische logica

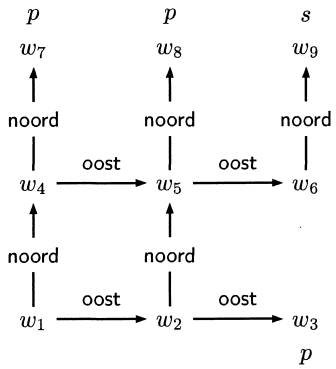
De klasse der *formules* van onze dynamische logica ontstaat nu net als in hoofdstuk 13. Er wordt begonnen met atomaire proposities $p, q, r, \ldots$ en vervolgens opgebouwd via de propositionele connectieven $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$ alsmede de programmamodaliteiten $[\pi]$ en $\langle\pi\rangle$ voor elk regulier programma π.

Vervlechting van formules en programma's

Merk op dat dit tot interactie leidt tussen de (logische) 'beweringen' en de (programma) 'instructies' in het systeem: modale proposities worden gevormd met behulp van programma's, maar ook omgekeerd vormen proposities programma's via de test op het basisprogramma. Zo'n interactie komt in meer geavanceerde programmeerformalismen wel vaker voor.

Voorbeeld 15.6

We illustreren de uitdrukkingskracht van de dynamische logica aan het 'schateiland'–voorbeeld uit hoofdstuk 13 over modale logica. Om te beginnen maken we de situatie nog iets gevaarlijker dan deze al was. In voorbeeld 13.4 kon je de piraten nog bevechten en daarna de schat bereiken. Nu daarentegen, betekent piraten tegenkomen de onverbiddelijke dood. Vanuit de werelden met piraten zijn daarom geen andere meer bereikbaar. De situatie is dus als volgt:



De atomen zijn p, voor 'aanwezigheid van piraten', en s, voor 'locatie van de schat', net als in het eerdere voorbeeld. De atomaire acties zijn n, voor 'naar het noorden gaan', en o, voor 'naar het oosten gaan'. Deze worden uiteraard geïnterpreteerd als de voor de hand liggende overgangen $T(n) = \mathit{noord}$ en $T(o) = \mathit{oost}$ in de figuur. Dit is dus anders dan hiervoor, waar er slechts van een enkele toegankelijkheidsrelatie sprake was.

Op dit model geldt bijvoorbeeld dat $[o\,;o\,;o]\neg\text{TRUE}$. Drie keer naar het oosten lopen kan nooit. Of met andere woorden: dit programma termineert niet. Tevens geldt dat $[n\,;o]\varphi \leftrightarrow [o;n]\varphi$. Of je nu eerst noord en dan oost gaat, of andersom, maakt niet uit. Met andere woorden: deze programma's zijn hier *equivalent*. Of je onderweg piraten tegenkomt is hierbij niet van belang: de bewering zegt dat *als* je eerst naar het noorden en daarna naar het oosten had kunnen gaan, *dan* had het ook andersom gekund. De atomaire acties zijn deterministisch: $\langle o\rangle\varphi \rightarrow [o]\varphi$ en tevens $\langle n\rangle\varphi \rightarrow [n]\varphi$. Als je piraten tegenkomt ben je dood (en kun je dus niet verder): $[p?\,;(n \cup o)]\neg\text{TRUE}$. Vanuit het startpunt kan de schat bereikt worden: $w_1 \vDash \langle(n \cup o)^*\rangle s$. Dit lukt namelijk via het pad w_1, w_4, w_5,

w_6, w_9, waarmee de sequentie $n\ ;\ o\ ;\ o\ ;\ n$ correspondeert. Dit is een concretisering van $(n \cup o)^4$ waarbij de eerste keuze n is, de tweede o, enzovoorts. Hiermee is het dus een mogelijke uitvoering van $(n \cup o)^*$ (namelijk: 'kies vier keer'). Vanuit een wereld waar piraten zijn, is het niet mogelijk de schat te bereiken: $p \rightarrow \neg\langle(n \cup o)^*\rangle s$.

Deze voorbeelden dienen ook ter illustratie van de algemene logisch-dynamische semantiek, waarvan de op STIP toegespitste inperking nu volgt.

Inbedding van STIP in reguliere programma's

Om STIP–programma's te laten aansluiten bij de dynamische logica eisen we bovendien:
a atomaire acties zijn toekenningen $x := t$;
b voorwaarden in testprogramma's zijn kwantorvrije Boolese condities.

Hiermee gaan we over van bijvoorbeeld een propositieletter p die staat voor '$x = 0$' naar de voorwaarde $x = 0$. Dit komt tot uitdrukking in de semantiek.

DEFINITIE 15.6

Semantiek van reguliere programma's
De beoogde semantiek van reguliere programma's luidt dan als volgt, in modellen $M = (\boldsymbol{D}, I)$:

$$b_1\ T(x := t)\ b_2 \quad \Leftrightarrow \quad b_2 = b_1[x \mapsto V_{(\boldsymbol{D}I),b_1}(t)]$$
$$b\ T(\varepsilon?)\ b \quad \Leftrightarrow \quad M, b \vDash \varepsilon$$

De overgangsrelatie T voor de drie overige programmaoperaties kan inductief gegeven worden met behulp van standaardoperaties op binaire relaties:

$$T(\pi_1\ ;\ \pi_2) = T(\pi_1) \circ T(\pi_2) \qquad \text{'compositie'}$$
$$T(\pi_1 \cup \pi_2) = T(\pi_1) \cup T(\pi_2) \qquad \text{'vereniging'}$$
$$T(\pi^*) = T(\pi)^* \qquad \text{'transitieve en reflexieve afsluiting'}$$

Hierbij staat $T(\pi)^*$ voor $\bigcup_{n\geq 0} T(\pi)^n$, waarbij $T(\pi)^n = T(\pi^n)$ en π^n staat voor $(\pi\ ;\ \ldots\ ;\ \pi)$ (n maal herhaald). Het programma π^0 komt overeen met TRUE?, het testprogramma dat altijd slaagt.

Indeterminisme

Merk op dat T–relaties voor reguliere programma's niet functioneel hoeven zijn. Vanuit een gegeven toestand kunnen ze meerdere mogelijke voortzettingen hebben. Reguliere programma's modelleren

daarmee bepaalde aspecten van 'indeterminisme'.

Dat hier echt van een generalisering sprake is ten opzichte van STIP, zegt de volgende bewering:

BEWERING 15.1

Elk STIP–programma kan uitgedrukt worden als een regulier programma met dezelfde overgangsrelatie.

Bewijs

We bewijzen dit met inductie naar de opbouw van STIP–programma's.
Atomaire toekenningen
Voor de atomaire toekenningen is de bewering per definitie het geval.
Opvolging
Opvolging komt bij STIP– en bij reguliere programma's voor.
Voorwaardelijke keuze
IF ε THEN π_1 ELSE $\pi_2 = (\varepsilon? \,;\, \pi_1) \cup (\neg\varepsilon? \,;\, \pi_2)$
Terwijl–lus
WHILE ε DO $\pi = ((\varepsilon? \,;\, \pi)^* \,;\, \neg\varepsilon?)$ □

Wat hier wordt bewerkstelligd is overigens loutere gelijkheid van overgangsrelaties. Er blijven namelijk intuïtieve verschillen tussen onze manieren van begrijpen hoe de programma's aan weerszijden van de gelijktekens functioneren. Zo lijkt kiezen voor het verkeerde alternatief rechts in het geval van de voorwaardelijke keuze, te leiden tot afbreken van de verwerking: anders dan aan de linkerkant, waar van keuze geen sprake is. Maar dergelijke verschillen vallen weg zolang we slechts op uiteindelijke succesvolle toestandsovergangen letten.

DEFINITIE 15.7

Axiomatiek voor de dynamische logica
We presenteren nu een axiomatiek voor de propositionele dynamische logica:

Propositielogische axioma's

a Elke instantie van een propositionele tautologie is een axioma.
b Modus Ponens: uit φ en $\varphi \to \psi$ volgt ψ.

Axioma's voor minimale modale logica

c Distributie
Π1 $[\pi](\varphi \to \psi) \to ([\pi]\varphi \to [\pi]\psi)$
d Dwang: indien φ bewijsbaar is, dan ook $[\pi]\varphi$.

Logica voor programma's

e Uitleg van toekenningen: $[x := t]\varphi \leftrightarrow [t/x]\varphi$, mits t vrij is voor x in φ.
f Decompositieaxioma's voor test, sequentie en keuze:
Π2 $[\varepsilon?]\varphi \leftrightarrow (\varepsilon \to \varphi)$
Π3 $[\pi_1 \,;\, \pi_2]\varphi \leftrightarrow [\pi_1][\pi_2]\varphi$
Π4 $[\pi_1 \cup \pi_2]\varphi \leftrightarrow [\pi_1]\varphi \wedge [\pi_2]\varphi$
g Inductieaxioma's voor iteratie:

Π5 $[\pi^*]\varphi \leftrightarrow (\varphi \wedge [\pi][\pi^*]\varphi)$
Π6 $(\varphi \wedge [\pi^*](\varphi \rightarrow [\pi]\varphi)) \rightarrow [\pi^*]\varphi$

We geven nog een illustratie van de werking van deze calculus:

BEWERING 15.2

Alle principes van de Hoare–calculus zijn in propositionele dynamische logica afleidbaar.

Als voorbeeld leiden we de regel van voorwaardelijke keuze H3 af. We gebruiken de vertaling van STIP– naar reguliere programma's:

a $(\varphi \wedge \varepsilon) \rightarrow [\pi_1]\psi$ (aanname in H3)
b $\varphi \rightarrow (\varepsilon \rightarrow [\pi_1]\psi)$ a + propositielogica
c $\varphi \rightarrow [\varepsilon?][\pi_1]\psi$ b + Π2
d $\varphi \rightarrow [\varepsilon? ; \pi_1]\psi$ c + Π3
Op analoge wijze vinden we uit de andere aanname van H3 een propositioneel–dynamische afleiding van:
e $\varphi \rightarrow [\neg\varepsilon? ; \pi_2]\psi$
We combineren dan d en e propositioneel tot:
f $\varphi \rightarrow ([\varepsilon? ; \pi_1]\psi \wedge [\neg\varepsilon? ; \pi_2]\psi)$
Dit leidt met axioma Π4 tot de gewenste conclusie van H3:
g $\varphi \rightarrow [(\varepsilon? ; \pi_1) \cup (\neg\varepsilon? ; \pi_2)]\psi$
Immers, g geeft precies de dynamische versie van de correctheids-bewering voor de IF...THEN...ELSE–opdracht.

Reguliere programma's en dynamische logica worden in toenemende mate in de informatica gebruikt, ook voor andere dan numerieke toepassingen.

15.6 OPGAVEN

15.1 Druk met een correctheidsbewering op de natuurlijke getallen uit wat het volgende STIP–programma π doet:

$(((u := 0$; WHILE $u \cdot u < x$ DO $u := Su)$; $(z := 0$; WHILE $z + z < y$ DO $z := Sz))$; IF $u > z$ THEN $s := u$ ELSE $s := z)$

15.2 a Beschouw een nieuwe STIP–programmaconstructie $\pi_1 \cup \pi_2$.
Geef een semantische definitie van $\pi_1 \cup \pi_2$.
b Geef een Hoare–regel voor correctheidsbeweringen over $\pi_1 \cup \pi_2$.

* c Beschouw een nieuwe STIP–programmaconstructie $\pi_1 \cap \pi_2$. Hoe zou u deze constructie interpreteren? Probeer ook voor deze constructie een Hoare–regel te vinden.

15.3 Geef een afleiding van H2 in dynamische logica (zie bewering 15.2).

* 15.4 De actie van het STIP–programma $x := y + 1$ kan in de predikaatlogica worden weergegeven door een formule waarin naast x en y variabelen x' en y' voorkomen die de waarde van x respectievelijk y geven na afloop van het programma: $y' = y \wedge x' = y + 1$.
Noem een dergelijke bij een programma horende formule een *transitie-predikaat*.
Laat zien hoe voor elk STIP–programma π waarin de variabelen $x_1, ..., x_n$ voorkomen, een transitiepredikaat kan worden gemaakt met variabelen $x_1, ..., x_n, x_1', ..., x_1'$.

* 15.5 Bewijs dat het faculteitsprogramma π uit voorbeeld 15.3 correct is.
Hint: begin met $\{x! = z!\}\ y := S0\ \{y \cdot x! = z!\}$ en gebruik de postconditie hiervan als preconditie bij de WHILE–lus.

* 15.6 Toon aan dat de WHILE–regel correct is (in de zin van correctheid zoals bij propositie– en predikaatlogica).

Logisch programmeren

Hoofdstuk 16

Logisch programmeren

16.1 INLEIDING

In het vorige hoofdstuk hebben we logische talen gebruikt als hulpmiddel om het gedrag van imperatieve programmeertalen te beschrijven. Het is echter ook mogelijk om bijvoorbeeld de predikaatlogica zelf als *declaratieve* programmeertaal te gebruiken. Deze programmeerstijl wordt *logisch programmeren* genoemd. Een van de bekendste voorbeelden van logisch programmeren is de taal PROLOG.
Er zijn verschillende argumenten ten gunste van logisch programmeren:
– het onderscheid tussen specificatie– en programmeertaal kan komen te vervallen;
– de parallellen tussen (correctheids)bewijzen en berekenen worden maximaal gebruikt ('rekenen = deductie');
– de gebruiker kan zich concentreren op de logische aspecten van een programmeerprobleem, en de precieze verwerkingsvolgorde aan de implementatie van het systeem overlaten ('algorithm = logic + control').
In dit hoofdstuk bespreken we een aantal aspecten van logisch programmeren in analogie met de eerdere logische systemen in dit boek. We beginnen met een voorbeeld.

Voorbeeld 16.1

Familierelaties
Julia is een vrouw en een van de ouders van Augustus. Vrouwelijke ouders heten moeders. Deze informatie kunnen we in predikaatlogica beschrijven als:

- $Ouder\_van(julia, augustus)$
- $Vrouw(julia)$
- $\forall x\, \forall y\, ((Ouder\_van(x, y) \wedge Vrouw(x)) \rightarrow Moeder\_van(x, y))$

Elk van deze formules behoort tot het universele fragment van de predikaatlogica. Dit gegevensbestand van familierelaties is een voorbeeld van een *logisch programma*. Een logisch programma kunnen we bevragen. Bijvoorbeeld: Is Julia de moeder van Augustus? Geldt: $Moeder\_van(julia, augustus)$?
Wie zijn de kinderen van Julia? Voor welke x geldt: $Ouder\_van(julia, x)$?

Wie zijn Julia's ouders? Voor welke x geldt: *Ouder_van*$(x, julia)$? Zo'n vraag heet een *doel* bij het programma. Antwoorden op deze vragen zijn te beschouwen als uitvoer van dit logische programma. Anderzijds corresponderen ze natuurlijk ook met logische gevolgen hiervan. In dit hoofdstuk onderzoeken we de precieze relatie tussen beide.

16.2 KENNISREPRESENTATIE MET LOGISCHE PROGRAMMA'S

In deze paragraaf onderzoeken we de taal waarin logische programma's worden geformuleerd, zodat we kunnen begrijpen wat met zulke programma's valt te 'zeggen'. In latere paragrafen wordt ingegaan op de vraag hoe zo'n programma 'rekent' en wat die berekening 'betekent'. We beginnen met enige nieuwe termen en notaties zoals die gangbaar zijn in de literatuur over logisch programmeren.

DEFINITIE 16.1

Clausule
Een *literal* is een predikaatlogisch atoom (*positieve literal*) of de negatie van een atoom (*negatieve literal*). Een *clausule* is een disjunctie van literals.

Stel $A_1, ..., A_k, B_1, ..., B_l$ zijn atomen, dan is $A_1 \vee ... \vee A_k \vee \neg B_1 \vee ... \vee \neg B_l$ een clausule. Het is gebruikelijk een clausule te schrijven in zogenaamde *programmanotatie*, namelijk als

$$A_1, ..., A_k \leftarrow B_1, ..., B_l$$

Dit wordt gelezen als 'A_1 of ... of A_k, als B_1 en ... en B_l'. Hierbij moeten we bedenken dat $(B_1 \wedge ... \wedge B_l) \rightarrow (A_1 \vee ... \vee A_k)$ logisch equivalent is met $A_1 \vee ... \vee A_k \vee \neg B_1 \vee ... \vee \neg B_l$. De procedurele strekking van de programmanotatie zal later nog aan de orde komen.

Een andere conventie is dat we clausules universeel gekwantificeerd lezen. Als $x_1, ..., x_n$ de in de voorgaande clausule voorkomende vrije variabelen zijn, dan wordt deze dus gelezen als

$$\forall x_1 ... \forall x_n (A_1 \vee ... \vee A_k \vee \neg B_1 \vee ... \vee \neg B_l)$$

Voorbeeld 16.2

Clausules
Twee voorbeelden van clausules zijn

- $Rxz \leftarrow Rxy, Ryz$
- $Rxy, Ryx, x = y \leftarrow$

In feite lezen we hier dus

- $\forall x\, \forall y\, \forall z\, ((Rxy \wedge Ryz) \rightarrow Rxz)$
- $\forall x\, \forall x\, (Rxy \vee Ryx \vee x = y)$

Deze clausules drukken respectievelijk *transitiviteit* en *lineariteit* van een binaire relatie R uit.

Universele formules

Clausules corresponderen met het universele fragment van de predikaatlogica uit hoofdstuk 8:

BEWERING 16.1

Elke *universele* predikaatlogische zin is logisch equivalent met een conjunctie van clausules.

Bewijs

Een universele zin had de vorm $\forall x_1 \ldots \forall x_n\, \varphi$, waarbij φ een kwantorvrije formule is. Dit laatste deel kunnen we nu propositielogisch equivalent herschrijven tot een conjunctie van disjuncties van atomen en hun negaties. Daartoe gebruiken we een methode uit hoofdstuk 2: $\neg\varphi$ heeft een *disjunctieve normaalvorm* ψ. Uit $\neg\psi$ werken we de negatie middels de wetten van De Morgan naar binnen (en dubbele negaties strepen we weg). Het resultaat ξ heeft de beschreven conjunctieve vorm en is equivalent met φ. Vervolgens distribueren we de universele kwantoren $\forall x_1 \ldots \forall x_n$ over deze conjunctie ξ met behulp van het geldige principe $\forall x\, (\alpha \wedge \beta) \leftrightarrow (\forall x\, \alpha \wedge \forall x\, \beta)$. Wat resulteert is een conjunctie van clausules. □

Horn–clausules

In de praktijk blijkt nog een verdere inperking van clausules nuttig, waarbij we hoogstens één positieve literal toestaan (zie hoofdstuk 8), dat wil zeggen, hoogstens één literal aan de linkerkant van de pijl $\leftarrow$:

DEFINITIE 16.2

Horn-clausule
Een *Horn–clausule* is een clausule met *ten hoogste* één positieve literal:

$$A \leftarrow B_1, \ldots, B_l$$

Hierbij heet A de *kop* en $B_1, \ldots, B_l$ de *staart*. In voorbeeld 16.2 was het transitiviteitsaxioma een Horn–clausule en het lineariteitsaxioma niet.

DEFINITIE 16.3

Logisch programma
Een *programmaclausule* is een Horn–clausule met *precies* één positieve literal. We maken hierbij onderscheid tussen feiten en regels. Een *feit* is een programmaclausule met een lege staart: $A \leftarrow$. Anders noemen we het een *regel*. Een *logisch programma* is een eindige verzameling programmaclausules.

Heel wat genres informatie vallen uit te drukken in de vorm van een logisch programma.

Voorbeeld 16.3

Familierelaties
Een logisch programma voor de informatie in voorbeeld 16.1 is:
- $Ouder\_van(julia, augustus) \leftarrow$
- $Vrouw(julia) \leftarrow$
- $Moeder\_van(x, y) \leftarrow Ouder\_van(x, y), Vrouw(x)$

Voorbeeld 16.4

Numerieke informatie
Het gedrag van rekenkundige operaties kan vaak als logisch programma beschreven worden, zoals we reeds zagen bij de 'definieerbare functies' van hoofdstuk 14. Het vermenigvuldigingspredikaat $Maal(x, y, z)$: '$x \cdot y = z$', beschreven we met de volgende recursie, tezamen met een hulppredikaat *Som* voor de optelling en met gebruikmaking van een 'voorgeprogrammeerd' functiesymbool voor opvolging. We herhalen deze exercitie nu in programmanotatie:
- $Som(0, x, x) \leftarrow$
- $Som(Sx, y, Sz) \leftarrow Som(x, y, z)$
- $Maal(0, x, 0) \leftarrow$
- $Maal(Sx, y, u)) \leftarrow Maal(x, y, z), Som(z, y, u)$

Voorbeeld 16.5

Niet–numerieke bewerkingen
Ook allerlei operaties op lijsten kunnen als logisch programma worden beschreven. Als we beschikken over een domein van symbolen en lijsten, een binaire operatie • die een symbool a vooraan een lijst x toevoegt en zo een nieuwe lijst $\bullet(a, x)$ maakt, en de 'lege lijst' [ ] als uitverkoren object, dan kunnen we lidmaatschap van een lijst als volgt omschrijven:
- $Lid(x, \bullet(x, y)) \leftarrow$
- $Lid(x, \bullet(z, y)) \leftarrow Lid(x, y)$

Concatenatie of aaneenschakeling van rijtjes ('aan elkaar plakken') kan worden omschreven via:
- $Plak([\,], x, x) \leftarrow$
- $Plak(\bullet(u, x), y, \bullet(u, z)) \leftarrow Plak(x, y, z)$

Bevragen van gegevensbestanden

Wat deze voorbeelden hebben laten zien, is hoe gegevens compact als logisch programma vallen op te schrijven. Een tweede belangrijke kwestie is dan natuurlijk hoe de voornaamste *vragen* eruitzien die we naar aanleiding van zulke gegevens zouden willen stellen. Ook hier blijken eenvoudige predikaatlogische vormen uit het Horn–fragment te volstaan:

DEFINITIE 16.4

Doel
Een *doel* is een Horn–clausule zonder kop: $\leftarrow B_1, ..., B_l$. De literals $B_1, ..., B_l$ heten de *subdoelen*. Een doel fungeert als een samengestelde vraag aan het programma. De *lege clausule* is een doel zonder subdoelen; met andere woorden: een Horn–clausule met kop noch staart. Hiervoor gebruiken we de notatie $\square$. De lege clausule fungeert als een STOP-instructie voor de programmaverwerking die correspondeert met het beantwoorden van een vraag.

Voorbeeld 16.6

Bevragen van gegevensbestanden
Meerdere vragen aan het familieprogramma van voorbeeld 16.1 werden al in de inleiding genoemd. Iets preciezer gaat het daar dus om de doelen, respectievelijk:
- $\leftarrow Moeder\_van(julia, augustus)$
- $\leftarrow Ouder\_van(julia, x)$
- $\leftarrow Ouder\_van(x, julia)$

Vragen aansluitend bij het numerieke voorbeeld 16.4 zijn:
Wat is het product van 2 en 3? Voor welke x geldt: $Maal(SS0, SSS0, x)$?
Wat zijn de delers van 5? Voor welke x en y geldt: $Maal(x, y, SSSSS0)$?
We kunnen dus ook vragen naar waarden voor een aantal variabelen tegelijk. En evenzo kan gevraagd worden naar waarden die aan een combinatie van eisen voldoen:
Voor welke paren getallen is zowel som als product gelijk aan 4?
Voor welke x en y geldt: $Som(x, y, SSSS0) \wedge Maal(x, y, SSSS0)$?
Deze gecombineerde vraag correspondeert dus met het doel
- $\leftarrow Som(x, y, SSSS0), Maal(x, y, SSSS0)$

16.3 BEWIJZEN ALS PROGRAMMAVERWERKING

Hoe 'werkt' nu een logisch programma? Om dit te kunnen inzien zijn twee dingen van belang. Ten eerste moeten we verband leggen tussen het beantwoorden van een vraag aan een logisch programma, en de notie van logisch gevolg. Ten tweede moeten we de invoer en uitvoer van zo'n programma relateren aan een vraag en een antwoord. We gaan eerst in op het verband met logisch gevolg.

Er zijn twee soorten vragen: vragen die we lijken te kunnen beantwoorden met ja of nee (Is Julia de moeder van Augustus?), en vragen die we kunnen beantwoorden door al dan niet objecten te noemen die

aan de gestelde vraag voldoen (Voor welke x geldt: $Ouder\_van(julia, x)$?). Wat houdt een antwoord op een vraag in?

JA/NEE–vragen

In het geval van een JA/NEE–vraag φ? proberen we φ af te leiden uit het programma Π. Lukt dat, dan geven we het antwoord JA, lukt dat niet, dan volgt het antwoord NEE. Hieraan zien we een eigenaardigheid van logisch programmeren. Bij de relatie 'semantisch gevolg' uit hoofdstuk 7 zijn er in principe immers *drie* mogelijke uitkomsten:

φ volgt uit Π	ja
$\neg\varphi$ volgt uit Π	nee
noch φ noch $\neg\varphi$ volgt uit Π	geen mening/onbekend.

De laatste twee gevallen worden echter bij logisch programmeren, en in het bijzonder bijvoorbeeld in PROLOG, tezamen genomen. Dit heet 'negation as failure' en betekent dus: 'het lukt niet om φ uit Π af te leiden'.

WELKE–vragen

In het geval van een WELKE–vraag 'voor welke x φ?' proberen we uit de gegevens speciale gevallen t van x te vinden, liefst zo algemeen mogelijk, waarvoor we $[t/x]\varphi$ kunnen bewijzen. Iets anders geformuleerd: we trachten de *existentiële* bewering $\exists x\ \varphi$ te bewijzen door geschikte getuigen t te produceren. De gevallen waarvoor dat lukt, geven we dan als antwoord.

Bewijsmethode

Nu is dan aan de orde welke bewijsmethode hier geëigend is. Laat Π een logisch programma zijn en $(D_1 \wedge ... \wedge D_l)$? een vraag aan dit programma. De vraag is nu of er waarden voor de variabelen in het doel zijn, zodat dit uit Π afleidbaar wordt. Met andere woorden, zodat
$\Pi \models \exists x_1 ... \exists x_n\ (D_1 \wedge ... \wedge D_l)$
Dit is logisch equivalent met
$\Pi, \neg\exists x_1 ... \exists x_n\ (D_1 \wedge ... \wedge D_l) \models \bot$
en met
$\Pi, \forall x_1 ... \forall x_n\ (\neg D_1 \vee ... \vee \neg D_l) \models \bot$
Het teken '$\bot$' is het falsum, de altijd onware bewering, zie hoofdstuk 4. Het verband tussen $\bot$ en $\square$ komt later aan bod. De programmanotatie voor $\forall x_1 ... \forall x_n\ (\neg D_1 \vee ... \vee \neg D_l)$ is $\leftarrow D_1, ..., D_l$. De vraag $(D_1 \wedge ... \wedge D_l)$? komt dus overeen met het doel $\leftarrow D_1, ..., D_l$, en dit doel zien we als de invoer bij het programma Π. We zijn nu aangekomen bij een speciaal geval van de algemene vraag of een verzameling clausules Π' (namelijk $\Pi' = \Pi \cup \{\leftarrow D_1, ..., D_l\}$) weerlegbaar is:
$\Pi' \models \bot$

Omdat we opereren binnen het predikaatlogische fragment van clausules kunnen we $\Pi' \models \bot$ systematisch onderzoeken met een bewijsmethode die *resolutie* heet. We behandelen nu eerst propositionele resolutie, daarna algemene resolutie, en ten slotte het speciale geval van SLD-resolutie, dat alleen op Horn-clausules van toepassing is. Tot slot blijkt dan, dat de antwoorden op een gegeven vraag aan een logisch programma overeenkomen met de substituties die nodig zijn om zo'n resolutiebewijs 'sluitend' te maken.

16.4 PROPOSITIONELE RESOLUTIE

DEFINITIE 16.5

Propositionele resolutie
Gegeven zijn twee clausules $\boldsymbol{A} \leftarrow \boldsymbol{B}$ en $\boldsymbol{C} \leftarrow \boldsymbol{D}$, waarbij $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$, en $\boldsymbol{D}$ rijtjes atomen zijn. Stel een atoom A komt zowel voor in $\boldsymbol{A}$ als in $\boldsymbol{D}$. Laat $\boldsymbol{A}'$ het resultaat zijn van het verwijderen van een aantal voorkomens van A uit $\boldsymbol{A}$, en $\boldsymbol{D}'$ het resultaat van het verwijderen van een aantal voorkomens van A uit $\boldsymbol{D}$. Dan volgt uit $\boldsymbol{A} \leftarrow \boldsymbol{B}$ en $\boldsymbol{C} \leftarrow \boldsymbol{D}$ met propositionele resolutie dat $\boldsymbol{A}', \boldsymbol{C} \leftarrow \boldsymbol{B}, \boldsymbol{D}'$.

'Normaal gesproken' denken we hierbij aan precies één voorkomen van A in $\boldsymbol{A}$ en van A in $\boldsymbol{D}$, maar zoals we hierna zullen zien is die beperktere formulering niet voldoende voor de volledigheid van propositionele resolutie. Overigens is in deze definitie een atoom een propositieletter, maar 'predikaatlogisch atoom zonder variabelen', zoals in voorbeeld 16.7 hierna, staan we ook toe.

BEWERING 16.2

Propositionele resolutie is semantisch correct.

Bewijs

We bewijzen dat $\boldsymbol{A}', \boldsymbol{C} \leftarrow \boldsymbol{B}, \boldsymbol{D}'$ een logisch gevolg is van $\boldsymbol{A} \leftarrow \boldsymbol{B}$ en $\boldsymbol{C} \leftarrow \boldsymbol{D}$. Met andere woorden: $\{\boldsymbol{A} \leftarrow \boldsymbol{B}, \boldsymbol{C} \leftarrow \boldsymbol{D}\} \models \boldsymbol{A}', \boldsymbol{C} \leftarrow \boldsymbol{B}, \boldsymbol{D}'$. Stel dat A. Daaruit en uit $\boldsymbol{C} \leftarrow \boldsymbol{D}$ volgt $\boldsymbol{C} \leftarrow \boldsymbol{D}'$. Uit $\boldsymbol{C} \leftarrow \boldsymbol{D}'$ volgt (de langere disjunctie) $\boldsymbol{A}', \boldsymbol{C} \leftarrow \boldsymbol{B}, \boldsymbol{D}'$. Stel nu dat $\neg A$. Dan volgt daar op vergelijkbare wijze uit dat $\boldsymbol{A}' \leftarrow \boldsymbol{B}$ en dus $\boldsymbol{A}', \boldsymbol{C} \leftarrow \boldsymbol{B}, \boldsymbol{D}'$. □

Het resultaat van een resolutiestap noemen we de *resolvent*. De resolvent kan zelf weer de invoer kan zijn van een volgende resolutiestap, enzovoort. Een *resolutiebewijs* is een rijtje van nul, of één, of meer resolutiestappen, en als φ het resultaat is van de laatste stap in een resolutiebewijs met aannames uit een logisch programma Π, dan schrijven we net als bij andere bewijsmethoden dat $\Pi \vdash \varphi$. Een *weerlegging* is een resolutiebewijs van de lege clausule □. Een weerlegging $\Pi \vdash \square$ komt overeen

met de inconsistentie van die verzameling clausules: $\Pi \models \bot$. Bedenk namelijk dat de laatste resolutiestap de vorm moet hebben 'uit $A \leftarrow$ en $\leftarrow A$ volgt $\square$'. De conjunctie van de verzameling $\{A \leftarrow, \leftarrow A\}$ is $A \wedge \neg A$, en dit is equivalent met $\bot$.

Voorbeeld 16.7

We bewijzen dat Julia de moeder van Augustus is in een weerleggingsbewijs van drie resolutiestappen. In plaats van de regel voor familierelaties nemen we een geschikte instantie hiervan.

1 $\leftarrow \mathit{Moeder\_van}(\mathit{julia}, \mathit{augustus})$
2 $\mathit{Moeder\_van}(\mathit{julia}, \mathit{augustus}) \leftarrow \mathit{Ouder\_van}(\mathit{julia}, \mathit{augustus}), \mathit{Vrouw}(\mathit{julia})$
3 $\leftarrow \mathit{Ouder\_van}(\mathit{julia}, \mathit{augustus}), \mathit{Vrouw}(\mathit{julia})$
4 $\mathit{Vrouw}(\mathit{julia}) \leftarrow$
5 $\leftarrow \mathit{Ouder\_van}(\mathit{julia}, \mathit{augustus})$
6 $\mathit{Ouder\_van}(\mathit{julia}, \mathit{augustus}) \leftarrow$
7 $\square$

Regel 3 volgt uit 1 en 2 met resolutie, regel 5 uit 3 en 4, en 7 uit 5 en 6.

Volledigheid

Zonder bewijs vermelden we het volgende volledigheidsresultaat:

BEWERING 16.3

Voor propositioneel afleiden van atomaire conclusies uit logische programma's volstaat propositionele resolutie.

Voor de volledigheid is essentieel dat we *meerdere* voorkomens van een atoom tegelijk kunnen verwijderen. Als dat maar voor één positief en één negatief voorkomen mag, kunnen we bijvoorbeeld uit (de inconsistente verzameling) p, $p \leftarrow$ en $\leftarrow p$, p met resolutie niet afleiden dat $\square$, maar alleen dat $p \leftarrow p$!

16.5 UNIFICATIE EN ALGEMENE RESOLUTIE

Resolutie is een plezierige methode, omdat we telkens alleen maar hoeven te letten op het mogelijke 'wegschrappen' van tegengestelde paren in tweetallen clausules. Dit geeft ons een sterke controle op het bewijsproces. Niettemin ligt de situatie in de praktijk ingewikkelder, omdat we bij de opzet tot nu toe de beginfase van substituties nog niet onder controle hebben gebracht. Daarom is in de informatica een predikaatlogische variant van de resolutieregel bedacht die propositionele resolutie combineert met substitutie. Alvorens deze regel te kunnen formuleren, moeten we het syntactische begrip *unificatie* invoeren, dat teruggrijpt op de predikaatlogische grammatica.

Unificatie

Unificatie is het 'gelijk maken' van uitdrukkingen door middel van geschikte substituties voor de erin voorkomende variabelen: een soort van 'pattern matching' dus. Dit kan tegelijkertijd voor al die variabelen, en dat vergt een nieuwe definitie van substitutie. Met 'uitdrukkingen' worden bedoeld: termen en atomen.

DEFINITIE 16.6

Gelijktijdige substitutie
Een gelijktijdige substitutie van termen $t_1, ..., t_n$ voor respectievelijk variabelen $x_1, ..., x_n$ geven we aan met $[t_1/x_1, ..., t_n/x_n]$. Uitdrukking $[t_1/x_1, ..., t_n/x_n]A$ is de uitdrukking (term of atoom) die ontstaat door alle voorkomens van x_1 in A te vervangen door t_1, ..., en tegelijkertijd alle voorkomens van x_n in A te vervangen door t_n. (Zie ook definitie 6.5.) Als θ een (gelijktijdige) substitutie is en A een uitdrukking, dan noteren we met θA de uitdrukking die ontstaat door het uitvoeren van θ op A.

DEFINITIE 16.7

Unificatie
Twee uitdrukkingen A en B heten *unificeerbaar* als er een (gelijktijdige) substitutie θ bestaat waarvoor $\theta A = \theta B$. We noemen θ een unificatie.

Voorbeeld 16.8

- $P(x, f(y))$ en $P(a, f(g(z)))$ zijn unificeerbaar via $\theta = [a/x, g(z)/y]$.
- $Q(a, g(x, a), f(y))$ en $Q(a, g(f(b), a), x)$ zijn unificeerbaar via $\theta = [f(b)/x, b/y]$.
- $R(h(x), c)$ en $R(f(a), y)$ zijn niet unificeerbaar: er is geen substitutie voor x die $h(x)$ en $f(a)$ gelijk kan maken.
- $Moeder\_van(julia, augustus)$ en $Moeder\_van(x, y)$ zijn unificeerbaar via $\theta = [julia/x, augustus/y]$.

Er valt altijd effectief te bepalen of twee uitdrukkingen unificeerbaar zijn. Daartoe bestaan diverse 'unificatiealgoritmen'. Als twee uitdrukkingen unificeerbaar zijn, kunnen we zelfs steeds een *meest algemene unificatie* vinden, waarvan alle andere unificaties speciale gevallen zijn. Bijvoorbeeld, in het eerste geval hiervoor zou de substitutie $[a/x, g(b)/y, b/z]$ ook hebben gewerkt, maar deze is een speciaal geval van de gegeven meest algemene oplossing.

Ter bepaling van een meest algemene unificatie van twee uitdrukkingen geven we nu het unificatiealgoritme van Martelli-Montanari. Twee atomen of termen t en u unificeren als het volgende algoritme termineert op invoer die we schrijven als $[t/u]$, waarbij we de uitdrukking t/u gemakshalve een vergelijking noemen. Het gaat er immers om, t en u

'gelijk' te maken (en aangezien u geen variabele hoeft te zijn, kunnen we $[t/u]$ immers geen substitutie noemen). In het algoritme is f (en g) normaal gesproken een functie, maar als t en u atomen zijn, is f een predikaat. Verder mag $n = 0$ ook: f kan ook een constante zijn.

DEFINITIE 16.8

Unificatiealgoritme
Begin met invoer $Eq := [t/u]$. Deze variabele Eq noemen we 'de vergelijkingen'. Kies een vergelijking uit Eq die een van de volgende zes (elkaar uitsluitende) vormen heeft (x is een variabele). Voer de bijbehorende actie uit. Kies nu opnieuw zo'n vergelijking uit Eq, enzovoorts.

- Als deze de vorm $f(t_1, ..., t_n)/f(u_1, ..., u_n)$ heeft, vervang deze dan in Eq door $t_1/u_1, ..., t_n/u_n$;
- Als deze de vorm $f(t_1, ..., t_n)/g(u_1, ..., u_m)$ heeft, en $f \neq g$ of $m \neq n$, dan faalt het algoritme;
- Als deze de vorm x/x heeft, verwijder deze dan uit Eq;
- Als deze de vorm $x/f(t_1, ..., t_n)$ heeft, vervang deze dan in Eq door $f(t_1, ..., t_n)/x$;
- Als deze de vorm $f(t_1, ..., t_n)/x$ heeft en x komt niet voor in $f(t_1, ..., t_n)$ maar wel elders in Eq, voer dan de substitutie $[f(t_1, ..., t_n)/x]$ overal elders in Eq uit;
- Als deze de vorm $f(t_1, ..., t_n)/x$ heeft en x komt voor in $f(t_1, ..., t_n)$, dan faalt het algoritme.

Bij 'falen' unificeren t en u niet, maar als daarentegen geen keuze meer kan worden gemaakt, is Eq de meest algemene unificatie van t en u.

Voorbeeld 16.9

We passen het unificatiealgoritme toe op het tweede onderdeel van voorbeeld 16.8. We beginnen met invoer $Eq := [Q(a, g(x, a), f(y))/Q(a, g(f(b), a), x)]$.

$[Q(a, g(x, a), f(y))/Q(a, g(f(b), a), x)]$
wordt $[a/a, g(x, a)/g(f(b), a), f(y)/x]$
wordt $[g(x, a)/g(f(b), a), f(y)/x]$
wordt $[x/f(b), a/a, f(y)/x]$
wordt $[f(b)/x, a/a, f(y)/x]$
wordt $[f(b)/x, f(y)/x]$
wordt $[f(b)/x, f(y)/f(b)]$
wordt $[f(b)/x, y/b]$
wordt $[f(b)/x, b/y]$

Unificatiealgoritmen zijn eenvoudig te generaliseren voor het unificeren van meer dan twee uitdrukkingen. Dit laten we achterwege.

Algemene resolutie voor clausules

Nu kunnen we de algemene resolutieregel formuleren op predikaatlogische clausules:

DEFINITIE 16.9

Algemene resolutie
Gegeven zijn twee clausules $\boldsymbol{A} \leftarrow \boldsymbol{B}$ en $\boldsymbol{C} \leftarrow \boldsymbol{D}$. Stel $A_1, ..., A_k \in \boldsymbol{A}$ en $D_1, ..., D_l \in \boldsymbol{D}$, en laat θ een meest algemene unificatie zijn van $\{A_1, ..., A_k, D_1, ..., D_l\}$. Schrijf $\boldsymbol{A}'$ voor $\boldsymbol{A}$ zonder $A_1, ..., A_k$ en $\boldsymbol{D}'$ voor $\boldsymbol{D}$ zonder $D_1, ..., D_l$. Dan volgt uit $\boldsymbol{A} \leftarrow \boldsymbol{B}$ en $\boldsymbol{C} \leftarrow \boldsymbol{D}$ met algemene resolutie dat $\theta(\boldsymbol{A}', \boldsymbol{C} \leftarrow \boldsymbol{B}, \boldsymbol{D}')$.

Net als in het propositionele geval, is de conclusie dus de vereniging van de resterende literals, maar nu met daarop de unificerende substitutie overal uitgevoerd. Wederom kiezen we 'normaal gesproken' één (positief) atoom $A \in \boldsymbol{A}$ en één (negatief) atoom $D \in \boldsymbol{D}$.

Voorbeeld 16.10

Uit $Ay \leftarrow Af(x)$, $Rh(x)y$ en $Rzf(u)$, $Bz \leftarrow$ volgt met algemene resolutie dat $Af(u)$, $Bh(x) \leftarrow Af(x)$, met meest algemene unificatie $\theta = [h(x)/z, f(u)/y]$.

Voorbeeld 16.11

Van voorbeeld 16.7 kunnen we een weerlegging met algemene resolutie maken, door de eerste resolutiestap daarin als volgt aan te passen:

1 $\leftarrow Moeder\_van(julia, augustus)$
2 $Moeder\_van(x, y) \leftarrow Ouder\_van(x, y), Vrouw(x)$
3 $\leftarrow Ouder\_van(julia, augustus), Vrouw(julia)$ $\quad \theta = [julia/x, augustus/y]$

BEWERING 16.4

Algemene resolutie is semantisch correct.

Bewijs

Gebruik dat voor willekeurige atomen geldt dat $A \models \theta A$. Verder gaat het bewijs als bij propositionele resolutie. □

Op de kwestie van volledigheid komen we in een latere paragraaf nog terug. Unificatie en algemene resolutie, waarvan hier alleen de hoofdideeën uiteen zijn gezet, zijn ook buiten de grenzen van logisch programmeren een veel voorkomende computationele methode geworden.

16.6 SLD-RESOLUTIE

Voor Horn–clausules bestaat een variant van algemene resolutie die SLD–resolutie heet. Hierop is ook de programmaverwerking van PROLOG gebaseerd.

DEFINITIE 16.10

SLD–resolutie
Laat $\leftarrow \boldsymbol{D} = \leftarrow D_1, ..., D, ..., D_l$ een doel zijn en Π een logisch programma. Stel voor een programmaclausule $A \leftarrow \boldsymbol{C} \in \Pi$ is er een meest algemene unificatie θ zodat $\theta D = \theta A$. Dan volgt uit $\leftarrow \boldsymbol{D}$ en $A \leftarrow \boldsymbol{C}$ met SLD-resolutie dat $\theta(\leftarrow D_1, ..., \boldsymbol{C}, ..., D_l)$.

Met andere woorden: we vervangen D in $\boldsymbol{D}$ door $\boldsymbol{C}$, en voeren de unificatie θ uit. SLD-resolutie is dus een speciaal geval van algemene resolutie. Een stapsgewijze afleiding van $\square$ met SLD–resolutie kunnen we als volgt sturen:

Eén stap

Uit het doel $\leftarrow \boldsymbol{D}$ selecteren we een subdoel D. Dit trachten we te unificeren met de kop van een programmaregel, zodat we een resolutiestap kunnen maken.
Lukt unificatie, dan krijgen we een nieuwe lijst doelen waaraan de condities uit de zojuist aangeroepen regel zijn toegevoegd, waarin D is weggelaten, en waarop de unificerende substitutie is toegepast.

De overige stappen

Weer moet dan een subdoel worden geselecteerd en worden gezocht naar een toepasselijke programmaclausule, enzovoorts. Merk op dat bij unificatie met een feit het aantal subdoelen van het doel afneemt.

Berekende antwoordsubstitutie

In de loop van dit proces treden allerlei substituties op voor de variabelen die voorkomen in het oorspronkelijke doel $\leftarrow \boldsymbol{D}$. Hierbij moeten we zorgvuldig in het oog houden dat bij die substituties niet *ten onrechte* variabelen gebonden of met elkaar geïdentificeerd worden. Om dat te voorkomen kiezen we bij iedere aanroep van een regel uit het logische programma 'verse' variabelen.
Het cumulatieve effect van alle substituties is een zogenaamde 'berekende antwoordsubstitutie' θ, waarvoor geldt $\Pi \models \theta\boldsymbol{D}$. De substitutie θ bevat de informatie die het antwoord is op de vraag $\boldsymbol{D}$?, namelijk de waarden voor de variabelen die in het doel $\leftarrow \boldsymbol{D}$ voorkomen. Vandaar de naam *antwoord*substitutie. Deze informatie is de uitvoer van het programma.

We leggen nu een en ander uit aan de hand van een gedetailleerd voorbeeld.

Voorbeeld 16.12

Eenvoudig rekenen
Beschouw het eerdere optellingsprogramma:
a $Som(0, x, x) \leftarrow$
b $Som(Sx, y, Sz) \leftarrow Som(x, y, z)$

Laat het volgende doel gegeven zijn. Dit correspondeert met de vraag wat het verschil is van 1 en 0:

c $\leftarrow Som(x, 0, S0)$

Dit doel is in ieder geval niet te unificeren met a. Men zou nu kunnen denken dat $Som(Sx, y, Sz)$ in programmaregel b en $Som(x, 0, S0)$ evenmin te unificeren zijn, omdat Sx en x niet gelijk te maken zijn. Maar de voorkomens van x in b en c hebben niets met elkaar te maken: beide zijn immers gebonden variabelen in universeel gekwantificeerde beweringen. Daarom kiezen we een alfabetische variant voor b, en vervangen de variabelen x, y en z in b door respectievelijk x_1, y_1 en z_1:

b' $Som(Sx_1, y_1, Sz_1) \leftarrow Som(x_1, y_1, z_1)$

Via de substitutie $[Sx_1/x, 0/y_1, 0/z_1]$ levert resolutie op c en b' het nieuwe doel:

d $\leftarrow Som(x_1, 0, 0)$

Deze clausule is vervolgens te unificeren met programmaregel a. Ook hiervoor kiezen we een alfabetische variant, namelijk

a' $Som(0, x_2, x_2) \leftarrow$

Via de substitutie $[0/x_1, 0/x_2]$ leidt dit met resolutie tot de conclusie □. Het antwoord op de vraag 'voor welke x geldt $x + 0 = 1$?' wordt nu gevonden door de compositie van beide substituties te berekenen. We hebben dat $[0/x_1, 0/x_2][Sx_1/x, 0/y_1, 0/z_1] = [S0/x, 0/y_1, 0/ z_1, 0/x_2]$. Alleen variabele x kwam in het oorspronkelijke doel voor. Het antwoord op deze vraag aan het programma is dus $[S0/x]$ ('$x = 1$').

We vatten de hele weerlegging nog eens als volgt samen:

1	$\leftarrow Som(x, 0, S0)$	doel
2	$Som(Sx_1, y_1, Sz_1) \leftarrow Som(x_1, y_1, z_1)$	regel b

3	$\leftarrow Som(x_1, 0, 0)$	uit 1 en 2 met $[Sx_1/x, 0/y_1, 0/z_1]$
4	$Som(0, x_2, x_2) \leftarrow$	regel a
5	$\square$	uit 3 en 4 met $[0/x_1, 0/x_2]$

Hoe eenvoudig ook, het voorgaande voorbeeld heeft laten zien hoe het resolutieproces cumulatief de informatie kan leveren die we voor de beantwoording van vragen in meer interessante praktijksituaties nodig hebben.

Rekenregels en zoekregels

Bij resolutiebewijzen doet zich tweemaal een keuzemoment voor:
- Welk subdoel uit een lopende lijst is als eerste te behandelen? Dit vergt een afspraak voor een zogenaamde *rekenregel.*
- Welke programmaclausule wensen we met het gekozen doel te unificeren (er kunnen er immers meerdere van toepassing zijn)? Dit vergt een zogenaamde *zoekregel.*

Strategie bij resolutiebewijzen

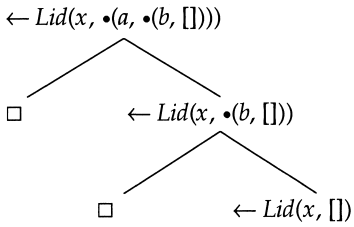
Bij een concrete implementatie zal hier een keuze moeten worden gemaakt. De meeste versies van PROLOG bijvoorbeeld, kiezen als rekenregel 'selecteer het meest linkse doel' en als zoekregel 'kies de eerste toepasbare programmaregel'. In het algemeen is de keuze van de rekenregel niet kritiek. Men zal hierdoor althans in principe geen geslaagde afleidingen over het hoofd zien. De keuze van de zoekregel is daarentegen uiterst belangrijk, willen we werkelijk alle mogelijke afleidingen nagaan. Deze kwesties vallen echter buiten het bestek van dit boek.

Weerleggingsboom

Afhankelijk van de keuze voor reken– en zoekregel zal men verschillende antwoordsubstituties kunnen krijgen. Een zogenaamde *weerleggingsboom* geeft een overzicht van alle resolutieafleidingen bij een gegeven doel, logisch programma en rekenregel. De wortel van de boom is het doel. Een kind van een gegeven knoop is het doel dat ontstaat door toepassing van resolutie op die knoop en een regel uit het logische programma. De alternatieven voor de zoekregel zijn in de weerleggingsboom zichtbaar als een vertakking. Een tak die in $\square$ eindigt, komt overeen met een weerlegging van het doel, met andere woorden: een antwoord op de vraag. Zo'n tak noemen we een 'succes'-tak. De weerleggingsboom bevat alle weerleggingen van het doel. Alle andere takken noemen we falende takken. Dit kunnen zowel eindige – 'doodlopende' – takken als oneindig doorlopende takken zijn.

Voorbeeld 16.13

Weerleggingsboom
Beschouw het eerdere programma voor het lidmaatschapspredikaat en vraag naar alle elementen van de lijst •(*a*, •(*b*, [])):



Aangezien het doel maar één subdoel heeft, is de rekenregel niet relevant. De meest linkse tak roept de eerste programmaregel aan, met berekende antwoordsubstitutie $[a/x]$. Dit is dus het antwoord dat gevonden wordt als de zoekregel is 'kies de eerste toepasbare programmaregel'. De rechtertak roept eerst de tweede programmaregel aan. Dit is dus het antwoord dat gevonden wordt als de zoekregel is 'kies de laatste toepasbare programmaregel'. Dan weer naar links aftakken naar de eerste programmaregel levert een afleiding van □ met berekende antwoordsubstitutie $[b/x]$. Geheel doorlopen naar rechts is een doodlopende tak: het doel *Lid*(x, [ ]) is niet te unificeren met een programmaclausule.

Dit is slechts een eenvoudig voorbeeld. Iets complexere notaties (waarbij we de verbindingen van de knopen labelen met de unificaties) zijn nodig wanneer bij een resolutieproces variabelen geleidelijk geïnstantieerd worden, zoals bij het *Som*–programma in voorbeeld 16.12.

16.7 PROLOG

Zoals gezegd is PROLOG een implementatie van een logische programmeertaal, waarbij de rekenregel is 'selecteer het meest linkse doel' en de zoekregel 'kies de eerste toepasbare programmaregel'. Nu kunnen we in zo'n geval wel eens in een doodlopende tak van de weerleggingsboom terechtkomen. PROLOG heeft ook een mechanisme om uit zo'n doodlopende tak te komen: backtracking. Dit komt erop neer dat voor de laatste keuze die volgens de zoekregel gemaakt is, een alternatief gezocht wordt. In de weerleggingsboom is er dan sprake van,

in het Engels, 'depth-first search'. We leggen dit verder niet in detail uit, maar volstaan met wat voorbeelden van PROLOG–programma's:

Voorbeeld 16.14

PROLOG–programma's
PROLOG–programma's voor de logische programma's van de voorbeelden 16.3, 16.4 en 16.5, zijn:

```
ouder_van(julia, augustus).
vrouw(julia).
moeder_van(X, Y) :- ouder_van(X, Y), vrouw(X).

som(0, X, X).
som(s(X), Y, s(Z)) :- som(X, Y, Z).

maal(0, X, 0).
maal(s(X), Y, U)) :- maal(X, Y, Z), som(Z, Y, U).

lid(X, punt(X, Y)).
lid(X, punt(Z, Y)) :- lid(X, Y).

plak(leeg, X, X).
plak(punt(U, X), Y, punt(U, Z)) :- plak(X, Y, Z).
```

De programma's voor `lid` en `plak` zijn hetzelfde als die voor de zogenaamde ingebouwde predikaten `member` en `append` in PROLOG, waarbij '`.`' de rol van `punt` vervult en '`[]`' de rol van de lege lijst `leeg`.

Een vraag aan zo'n PROLOG–programma komt natuurlijk overeen met een doel bij het logische programma. Zo komt het doel $\leftarrow Som(x, 0, S0)$, met bijbehorende berekende antwoordsubstitutie $[S0/x]$, overeen met de volgende combinatie van vraag en antwoord:

```
?- som(X,0,s(0)).

X = s(0) ;

Yes
?-
```

Hiermee besluiten we deze korte uitweiding over PROLOG.

16.8 SEMANTIEK VAN LOGISCHE PROGRAMMA'S

Een eerste aspect van Horn–clausules hebben we nu leren kennen. Ze staan een overzichtelijke resolutiebewijsmethode toe, die onderweg informatie levert over gewenste antwoorden. Een nog niet besproken aspect is de betekenis of semantiek van onze programma's. Wanneer we een logisch programma schrijven, hebben we doorgaans een situatie in gedachten waarin het waar is.

Herbrand–modellen

De betekenis van een verzameling Σ van formules in de standaardlogica werd bepaald door de klasse van alle modellen van Σ, en dat kunnen er heel veel zijn. Bij een logisch programma Π hebben we echter meestal één specifiek model in gedachten, bijvoorbeeld, bij de eerdere numerieke programma's de natuurlijke getallen. Zolang we met Horn–clausules werken, kunnen we een dergelijk 'voorkeursmodel' inderdaad aangeven.
Het domein van dit model wordt als volgt gedefinieerd:

DEFINITIE 16.11

Herbrand–universum
Het Herbrand–universum H_Π van een logisch programma Π bestaat uit alle gesloten termen uit de taal van Π.

Voorbeeld 16.15

- Als $\Pi = \{R(0, x, x) \leftarrow, R(Sy, x, Sz) \leftarrow R(y, x, z)\}$, dan bevat de taal van Π de constante 0 en de functie S. H_Π is dan $\{0, S0, SS0, ...\}$.
- Als een taal geen individuele constanten bevat, dan zijn er ook geen gesloten termen. In dat geval kiezen we een nieuwe constante a als 'voorgift'. Met $\Pi = \{M(fx) \leftarrow, N(x) \leftarrow M(gx), N(ffx) \leftarrow\}$ (hier zijn f en g functieletters) wordt H_Π dan $\{a, fa, ga, ffa, fga, gga, gfa, ...\}$.

Op het Herbrand–universum definiëren we nu:

DEFINITIE 16.12

Kleinste Herbrand–model
Het kleinste Herbrand–model $\mu(\Pi)$ voor een logisch programma Π wordt als volgt gedefinieerd:

a Het domein van $\mu(\Pi)$ is het Herbrand–universum H_Π.
b1 $I(a) = a$, voor elke individuele constante a.
b2 Als f een n–plaatsig functieteken is en $t_1, ..., t_n$ termen in H_Π, dan $I(f)(t_1, ..., t_n) = f(t_1, ..., t_n)$.
c De interpretatie der predikaatletters P is 'zo klein mogelijk': $I(P)$ = de verzameling van veeltallen termen $t_1, ..., t_n$ waarvoor $\Pi \models P(t_1, ..., t_n)$.

Dergelijke kleinste Herbrand–modellen bestaan altijd voor logische programma's, en ze zijn zelfs *uniek*.

Herbrand–modellen

Behalve het kleinste Herbrand–model kan een programma Π vele uiteenlopende modellen op zijn Herbrand–universum hebben, waaronder doorgaans vele 'onbedoelde'. De interpretatie van predikaatletters wordt dan ruimer genomen. Zulke modellen noemen we in het algemeen Herbrand–modellen.

Voorbeeld 16.16

Het *Som*–programma van voorbeeld 16.12 heeft vele Herbrand-modellen:

- Het grootste Herbrand–model maakt *Som* waar voor *alle* drietallen van termen uit het Herbrand–universum {0, *S*0, *SS*0, ...}, met andere woorden: alle (x, y, z) met x, y en z natuurlijke getallen.
- Het kleinste Herbrand–model $\mu(\Pi)$ heeft precies de standaard-interpretatie voor *Som* als optelling. Met andere woorden: alle (x, y, z) met $x + y = z$.
- Een van de vele modellen daar tussenin bevat tevens het equivalent van $1 + 1 = 3$, dus $(S0, S0, SSS0) \in I(Som)$. Omdat het model moet zijn van de recursieve regel van het *Som*–programma, *moeten* in de interpretatie van *Som* in dit model tevens de equivalenten zitten van $2 + 1 = 4$, $3 + 1 = 5$, $4 + 1 = 6$, enzovoorts. We kunnen dus niet geheel willekeurig maar wat drietallen aan de interpretatie van *Som* toevoegen!

Het volgende resultaat (zonder bewijs) rechtvaardigt het gebruik van kleinste Herbrand–modellen en van de resolutiebewijsprocedure:

STELLING 16.1

Zij Π een logisch programma en φ een (existentieel gelezen) atoom of conjunctie van atomen. Dan zijn de volgende beweringen equivalent:

a $\Pi \models \varphi$, in de gewone predikaatlogica
b $\mu(\Pi) \models \varphi$, volgens de waarheidsdefinitie
c $\Pi \vdash \varphi$ met SLD–resolutie.

De complexere notie van 'geldig gevolg' in a is dus equivalent met de eenvoudiger notie van 'model checking' in b, in dit geval waarheid in het kleinste Herbrand-model. Een gevolg van deze stelling is dat de theorie van een logisch programma Π de existentie–eigenschap (zie hoofdstuk 12) heeft: voor iedere geldige existentiële bewering is er een specifieke invulling van termen voor de kwantorvariabelen die reeds uit Π volgt. Dit verklaart dat berekenbare antwoordsubstituties altijd bestaan.

De stelling geldt dan ook niet voor predikaatlogische formules in het algemeen. Twee voorbeelden:

Voorbeeld 16.17

De verzameling $\Sigma = \{Pa, \exists x\, \neg Px\}$ heeft wel een model, maar geen kleinste Herbrand–model. Het Herbrand–universum van Σ is $\{a\}$, want a is de enige gesloten term in de taal van Σ. Maar op dit domein kan Σ nooit waar gemaakt worden.

Voorbeeld 16.18

Evenzo heeft een disjunctie als $Pa \vee Qa$ geen kleinste Herbrand–model. In dit geval is namelijk sprake van twee 'kleinste' modellen wat betreft de interpretatie van predikaten: één waarin Pa geldt en Qa niet, en één waarin Qa geldt en Pa niet. Op grond daarvan mogen dus Qa respectievelijk Pa geen deel uitmaken van een kleinste Herbrand–model. Maar als noch Pa noch Qa mogen gelden, dan gaat hun disjunctie evenmin op.

Er valt zelfs te bewijzen dat het fragment der Horn–clausules het ruimste is dat een algemene garantie biedt voor het bestaan van dergelijke minimale modellen ('Stelling van Mahr en Makovsky').

Wanneer we voor Π wel een logisch programma nemen, geldt stelling 16.1 evenmin als we voor φ niet een existentieel gelezen atoom of conjunctie nemen (dus niet een doel), maar een universele formule, bijvoorbeeld een programmaregel. Het is goed mogelijk dat dan meer waar is in $\mu(\Pi)$ dan strikt logisch uit Π volgt. Bijvoorbeeld, in het standaardmodel voor ons *Som*–programma is de optelling *commutatief*:

$\forall x\, \forall y\, \forall z\, (Som(x, y, z) \rightarrow Som(y, x, z))$

Maar deze laatste zin is geen standaard logisch gevolg van de *Som*-principes: de laatste hebben modellen, zelfs op het Herbrand–universum, waar $I(Som)$ niet commutatief is.

Voorbeeld 16.19

Het laatste Herbrand–model voor *Som* uit voorbeeld 16.16 bevat wel het equivalent van 2 + 1 = 4 (want 1 + 1 = 3 zit erin) maar niet dat van 1 + 2 = 4 (want 0 + 2 = 3 zit er niet in). Daarin is de optelling dus niet commutatief!

We zien hier dus het volgende: wanneer we ons beperken tot werken met minimale Herbrand–modellen, dan volgen er consequenties uit programma's die er niet semantisch uit volgen.

16.9 SKOLEM–VORMEN EN ALGEMENE RESOLUTIE

Er zijn ook aspecten aan het werken met Horn–clausules die zich laten generaliseren tot ruimere fragmenten van de predikaatlogica, resolutie bijvoorbeeld. Voor Horn–clausules levert resolutie, toegepast op een doel en een programmaclausule, weer een doel op. Zoals we reeds zagen, was deze bewijsmethode echter ook van toepassing op *willekeurige* clausules. Algemene resolutie is semantisch volledig in de volgende zin (zonder bewijs):

STELLING 16.2

Weerleggingsvolledigheid van resolutie
Voor elke verzameling clausules S geldt: $S \models \bot \Leftrightarrow S \vdash \bot$ met algemene resolutie.

Algemene resolutie speelt een belangrijke rol als mechanisme voor het automatisch bewijzen van stellingen. Deze bewijsregel werd eigenlijk ook in dit laatste gebied ontdekt: logisch programmeren is historisch een latere toepassing. De mogelijkheden van algemene resolutie zijn vooral zo groot, omdat *elke* predikaatlogische gevolgtrekking effectief valt te reduceren tot de weerleggingsvorm. Een schets van hoe dat stapsgewijs mogelijk is, vormt het laatste thema in dit hoofdstuk.

1 Laat Σ een eindige verzameling predikaatlogische formules zijn, σ de conjunctie van alle formules in Σ, en ψ een formule. De vraag of $\Sigma \models \psi$, is equivalent met de vraag of $\sigma \wedge \neg\psi \models \bot$. Dus kunnen we ons beperken tot de voorgaande weerleggingsvragen van de vorm $\varphi \models \bot$ voor willekeurige predikaatlogische formules φ.
2 Vervolgens gaan we φ herschrijven tot een verzameling clausules.
2a Uit hoofdstuk 8 weten we om te beginnen dat elke zin φ equivalent is met een formule in *prenexvorm* $Q_1x_1 \ldots Q_nx_n\,\chi$, waarbij $Q_1, \ldots, Q_n$ kwantoren $\forall$ of $\exists$ zijn en χ een kwantorvrije formule is waarin precies de variabelen $x_1, \ldots, x_n$ voorkomen.
2b Om resolutie te kunnen toepassen, willen we de $\exists$–kwantoren in de prefix op een of andere manier verwijderen. Dit lukt door het introduceren van zogeheten *Skolem–functies*. Kwantorcombinaties drukken namelijk net als functies typisch afhankelijkheden uit.

Voorbeeld 16.20

Eenplaatsige Skolem–functie
Beschouw bijvoorbeeld de formule $\forall x\, \exists y\, Rxy$. Stel nu dat f een 'keuzefunctie' is die bij elke x een y vindt zodat Rxy. Dan is de gegeven formule dus te reduceren tot $\forall x\, Rxf(x)$. Zo'n functie f is nu een Skolem–functie.

Voorbeeld 16.21

Meerplaatsige Skolem–functie
Meer in het algemeen kunnen we met behulp van Skolem–functies expliciete afhankelijkheden in kwantorpatronen tot uitdrukking brengen. Met behulp van een eenplaatsige Skolem–functie f is de formule

$$\forall x\, \exists y\, \forall z\, \exists u\, ((Rxy \wedge Sxyz) \rightarrow Tu)$$

te reduceren tot de vorm

$$\forall x\, \forall z\, \exists u\, ((Rxf(x) \wedge Sxf(x)z) \rightarrow Tu)$$

Immers, y was alleen afhankelijk van x. De volgende existentiële u is echter afhankelijk van zowel x als z, hetgeen we verantwoorden met een tweeplaatsige Skolem–functie g:

$$\forall x\, \forall z\, ((Rxf(x) \wedge Sxf(x)z) \rightarrow Tg(x, z))$$

Voorbeeld 16.22

Nulplaatsige Skolem–functie
In het geval dat een existentiële kwantor voorop staat, zoals in $\exists x\, \forall y\, Rxy$, kiezen we een nulplaatsige functie ofwel een individuele constante, zeg $k : \forall y\, Rky$.

Aldus kan elke formule φ in prenexvorm vervangen worden door een zuiver universele formule φ^* met een geschikt aantal Skolem–functies $f_1, ..., f_k$. De verhouding tussen deze twee laat zich omschrijven in *tweede–orde* logica (vergelijk hoofdstuk 12), waarin kwantificatie over functies immers mogelijk is:

φ is logisch equivalent met $\exists f_1 ... \exists f_k\, \varphi^*$

Niettemin valt eenvoudig in te zien dat voor weerleggingsvragen de volgende gewone eerste–orde predikaatlogische equivalentie opgaat:

$$\varphi \vDash \bot \Leftrightarrow \varphi^* \vDash \bot$$

En daarmee zijn we, zoals beloofd, aangekomen bij een weerleggingsprobleem binnen het universele fragment van de predikaatlogica.

16.10 OPGAVEN

16.1 Bepaal voor elk van de volgende paren uitdrukkingen of ze unificeerbaar zijn en zo ja, geef dan de meest algemene substitutie:
i $Pf(x)h(y)$, $Pf(g(y))y$
ii $Qf(h(x))y$, $Qf(y)h(a)$
iii $Rg(h(a, x))h(a, g(x))$, $Rg(y)h(y, g(y))$

16.2 Beschouw de volgende verzameling clausules (universeel gekwantificeerd gelezen):

$$\{\neg Cza \vee \neg Bz, Dwu \vee \neg Avuf(w) \vee Cvw, Bb, Axf(x)f(y)\}$$

Bepaal door middel van algemene resolutie voor welke termen t en s geldt dat Dts afleidbaar is uit deze verzameling.

16.3 a Geef clausules in programmanotatie voor $Maal(x, y, z)$ ('$x \cdot y = z$') en $Fac(x, y)$ ('$x! = y$').
b Geef weerleggingsbomen voor de volgende vragen en leid hieruit de antwoorden af:
i $Maal(SS0, x, SSSS0)$
ii $Fac(x, S0)$
iii $Fac(SS0, x)$

16.4 Zet de volgende formules om in universele clausules (met behulp van equivalenties en eventueel Skolem–functies). Geef aan welke zinnen Horn–zinnen zijn.
i $\neg\forall x\ \forall y\ Rxy \vee \forall z Az$
ii $\forall x\ (Ax \rightarrow Bx) \rightarrow (\forall x\ Ax \rightarrow \forall x\ Bx)$
iii $\forall x\ (\forall y\ (Ryx \rightarrow Ay) \rightarrow Ax) \rightarrow \forall x\ Ax$
iv $\forall y\ \forall z\ (((Ryc \rightarrow Ay) \rightarrow Ac) \rightarrow Az)$

* 16.5 Bewijs dat propositionele resolutie voor redeneren met clausules volledig is.

Logica en computationele complexiteit

Hoofdstuk 17

Logica en computationele complexiteit

17.1 INLEIDING

Redeneren en rekenen liggen dicht bij elkaar, zoals is gebleken in de voorgaande hoofdstukken over programmeerstijlen. Deze twintigste-eeuwse contacten tussen logica en informatica gaan op zich weer terug op een oudere historische traditie van redeneermachines. Zo werken de digitale circuits van een moderne computer in wezen volgens de negentiende-eeuwse propositielogica van George Boole. Maar naast dit *rekenaspect* er is nog een tweede kant aan het contact tussen logica en informatica. De logische talen in dit boek geven manieren om diverse soorten van informatie exact uit te drukken: propositioneel, met kwantoren, modaal, temporeel en anderszins. Daarmee wordt die informatie voor berekening vatbaar. *Representatie* met logica zal meer centraal staan in ons volgende deel, over contacten met de kunstmatige intelligentie. In dit hoofdstuk staat de *berekening* centraal.

In de informatica gaat het om het juiste samenspel tussen de geschikte weergave of representatie van informatie (in getalstructuren, gegevens-banken, of andere vormen) en efficiënte berekening met die gegevens, met andere woorden om de interactie tussen, in het Engels: 'representation + computation'. Ook in het ontwerpen van logische systemen gaat het om een balans tussen logische uitdrukkingskracht en rekencomplexiteit. In het algemeen geldt: hoe rijker de taal, hoe moeilijker het rekenproces voor de centrale logische taken. De bedoeling van dit korte hoofdstuk is om iets meer inzicht te geven in deze balans, die vooral typerend is voor logische systemen die ontwikkeld zijn in de buurt van de informatica.

In paragraaf 17.2 introduceren we de relevante logische taken: bewijzen, vervullen, evalueren en vergelijken. In 17.3 en 17.4 preciseren we hoe moeilijk deze in het algemeen zijn uit te voeren. In de paragrafen daarna geven we een concreet overzicht van de complexiteit van standaard logische systemen.

17.2 BEWIJZEN, VERVULLEN, EVALUEREN EN VERGELIJKEN

In dit boek zijn heel uiteenlopende logische taken besproken. Het berekenen van een waarheidswaarde van een formule in een gegeven model is zo'n taak, maar ook het vinden van een model voor een gegeven stel formules, of weer anders: het nagaan of twee gegeven modellen een nuttige semantische invariantie vertonen, zoals een modale bisimulatie. We zetten een aantal van deze taken nog eens iets uitvoeriger op een rij. Om te beginnen hebben we *bewijstaken*.

Bewijstaken (theorem proving)

Veel theoretische en praktische problemen komen erop neer dat bepaald moet worden of een bewering uit een andere volgt. We kunnen dit zien bij:

- *logisch programmeren* (hoofdstuk 16): het afleiden van antwoorden op vragen uit een bestand van feiten en regels;
- *predikaatlogische theorieën* (hoofdstuk 10): het automatisch vinden van wiskundige stellingen die volgen uit gegeven axioma's;
- *expertsystemen* (valt onder hoofdstuk 19, hierna): bijvoorbeeld een rechter of dokter leidt een conclusie af uit gegeven informatie;
- *taalkundige zinsontleding* (hoofdstuk 20, hierna): de constructie van een bewijs dat een reeks symbolen volgens de regels der grammatica is te produceren.

Dit soort logische geldigheidsvragen wordt vaak gesteld binnen het kader van een bewijssysteem en de gangbare term hiervoor is *theorem proving*. Aan de complexiteit van bewijzen besteden we wegens ruimtegebrek geen aandacht in dit boek. Er zijn echter ook vele puur semantische logische taken.

Vervulbaarheidstaken (satisfiability)

In gewone conversatie 'bewijzen' we zelden formeel. Het gaat er dan meer om, consistente informatie te verstrekken. Dit 'consistency management' heeft als logische kernvraag of een gegeven verzameling beweringen een model heeft. Dit is de zogenaamde *vervulbaarheid*, in het Engels: *satisfiability*, van die verzameling. *Vervulbaar* is dus een ander woord voor het begrip *semantisch consistent* uit hoofdstuk 3. Dezelfde vraag doet zich in een ander jasje voor in *ontwerp*, bijvoorbeeld het ontwerp van een digitaal circuit dat moet voldoen aan vooraf gegeven Boolese specificaties, of dat van een procesdiagram voor een machine die moet voldoen aan vooraf gegeven modaal-logische eisen.

Evaluatietaken (model checking)

In wezen eenvoudiger dan vervullingstaken, maar ook zeer nuttig, zijn *evaluatietaken*. Hoe bepalen we de waarheidswaarde van een formule in

een semantisch model? Dit is de controle of een gegeven structuur voldoet aan zekere eisen die we van tevoren hebben gesteld. In het Engels: *model checking*. Het checken van specificaties voor een gegeven proces is van deze aard, evenals het bepalen van eigenschappen van een Booles circuit.

Logische taken kunnen natuurlijk ook tezamen voorkomen in eenzelfde situatie. Zo moet in de rechtszaal de aanklager bewijzen dat de beklaagde schuldig is: een bewijstaak voor een conclusie uit de voorliggende gegevens. Maar de verdediger heeft slechts de vervulbaarheidstaak om een scenario te schetsen dat past bij die gegevens, maar waarin zijn cliënt onschuldig is. Ook evaluatietaken kunnen voorkomen, namelijk bij het vaststellen van het toelaatbare bewijsmateriaal.

Vergelijkingstaken (model comparison)

Ten slotte zijn er nog heel andere genres taken. Heel belangrijk zijn bijvoorbeeld *vergelijkingstaken*. Wanneer zijn twee Boolese circuits equivalent, of twee procesdiagrammen, of twee modellen voor de predikaatlogica – in die zin dat ze dezelfde formules van de relevante taal waar maken? Dit soort informatie is cruciaal bij het vereenvoudigen van voorgestelde machines of informatieprocessen. Een typisch voorbeeld is het testen van modale modellen op bisimulatie, zoals in hoofdstuk 13 besproken is.

17.3 HOE MOEILIJK ZIJN LOGISCHE TAKEN ?

Onze volgende vraag betreft de moeilijkheid van deze logische taken. Een evaluatietaak in de propositielogica, dat wil zeggen het uitrekenen van een waarheidswaarde in een waarheidstabel, lijkt op het eerste gezicht eenvoudig, en gelukkig is dat ook inderdaad zo. Een rij in een waarheidstabel komt overeen met een propositielogische waardering. In die rij vullen we één voor één de waarheidswaarden in voor alle subformules van een gegeven formule. Dit kost ons evenveel 'tijd' (stappen) als het aantal subformules van die formule. En het aantal subformules is weer ongeveer gelijk aan de lengte van die formule: het aantal symbolen. Men zegt wel dat deze evaluatietaak in de *orde van grootte* is van de lengte van de formule. Evaluatie in de propositielogica kost dus zogenaamde *lineaire* tijd, gemeten in de lengte van de invoerformule.

Daarentegen is het is veel lastiger om te bepalen of een formule een model heeft (*satisfiability*), omdat we nu in het ergste geval alle rijen in de waarheidstabel zullen moeten nagaan. Dit is een *exponentieel* aantal,

namelijk alle combinaties van nullen en enen voor de propositieletters in de formule. De tableaumethode uit hoofdstuk 3 is misschien wat eenvoudiger, maar hoeveel eenvoudiger? We komen hier nog op terug.

Soortgelijke vragen rijzen bij de rijkere taal van de predikaatlogica. Hoe moeilijk is het bijvoorbeeld om voor een formule $\forall x\, \exists y\, Rxy$ met twee kwantoren te bepalen of deze waar is in een gegeven model van n objecten? Op het eerste gezicht is het antwoord hierop: dit duurt n^2 stappen. In het ergste geval moeten we namelijk alle (n) objecten één voor één voor x invullen, en voor elk daarvan weer alle (n) objecten proberen als mogelijke y. In het algemeen verwachten we dus *exponentiële* tijd voor de evaluatie van een formule met kwantoren, als we het model en die formule als invoerparameters nemen: namelijk het aantal objecten van het model tot de macht van de grootste 'nesting' van kwantoren in de formule. Maar ook hier kunnen verrassingen optreden die tot lagere complexiteit leiden!

Voorbeeld 17.1

Een 'beroemdheid' is iemand die niemand anders kent, maar wel door ieder ander wordt gekend. Hoe bepalen we zo snel mogelijk of een groep een beroemdheid heeft, en wie dat is? Het gaat hier om de vraag of in een gegeven model de predikaatlogische formule $\exists x\, \forall y\, (y \neq x \rightarrow (Kyx \wedge \neg Kxy))$ waar is. Omdat er een stapeling van twee kwantoren is, zou men verwachten dat dit kwadratische tijd kost. Het kan echter *lineair*! We geven daartoe aan iedereen een rood balletje, voor 'potentiële beroemdheid'. Nu herhalen we de volgende procedure:
Zolang er nog minstens twee mensen met rode balletjes zijn, pak er dan twee, zeg x en y, en kijk of x y kent. Zo ja, neem dan x het rode balletje af, zo nee, y. De unieke overblijver met een balletje wordt nu nog even getest op de eigenschap beroemdheid te zijn. Dit hele proces loopt hoogstens drie keer door het domein: namelijk twee keer om de overblijver met het rode balletje te bepalen (twee objecten kiezen, net zo vaak als het aantal objecten), en één keer om voor alle anderen te testen of zij die overblijver kennen en de overblijver hen niet kent. Daarmee is de complexiteit lineair. (Uit: *Programming, the derivation of algorithms*, zie literatuurlijst.)

Van de complexiteit van vergelijkingstaken geven we pas hierna een concreet voorbeeld. We formuleren nog eens de drie besproken typische taken van een logische taal:

Model checking

Gegeven een model M, een wereld w, en een formule φ, geldt $M, w \vDash \varphi$?

Satisfiability

Gegeven een formule φ, bestaan er M en w in M, zodat $M,w \models \varphi$?

Model comparison

Gegeven eindige modellen M en N, maken deze dezelfde formules φ waar?

Als we de precieze rekencomplexiteit van al deze taken kennen, dan hebben we een redelijk 'profiel' van het computationele gedrag van zo'n logisch systeem. We zullen deze kwesties in dit hoofdstuk langslopen voor de belangrijkste talen van dit boek, overigens zonder naar grote precisie of volledigheid te streven. Maar voordat we dat kunnen doen, moeten we natuurlijk wel iets preciezer zijn over wat we bedoelen met rekencomplexiteit! Dit is op zich al een belangrijk vakspecialisme in de praktische en theoretische informatica.

17.4 COMPUTATIONELE COMPLEXITEIT

Een taak is effectief opgelost als je er een algoritme voor kunt schrijven, misschien als programma in een of andere programmeertaal. Preciezer, bij, bijvoorbeeld, een JA/NEE–vraag '*P*?' wordt de bewering *P* als invoer aan een mechanische procedure gegeven die na een eindig aantal stappen steeds het juiste antwoord geeft. Een gedetailleerd wiskundig model voor zulke procedures zijn de *registermachines* uit hoofdstuk 14, die via getalsmatige codering ieder concreet symbolisch probleem aankunnen. In de voorbeelden hierna blijven we op een informeler, maar wel precies, niveau.

Heel veel taken in de informatica, zoals het bevragen van gegevensbanken, of zoektaken in de kunstmatige intelligentie, komen neer op het bepalen van de 'bereikbaarheid' van een doel uit een beginpunt:

Voorbeeld 17.2

Graph reachability (GR)
Gegeven is een eindige gerichte graaf G en twee knopen s ('beginknoop') en t ('eindknoop'). Is er een pad van s naar t? Deze vraag is te beantwoorden met het volgende algoritme:
R wordt de verzameling rode knopen. Kleur om te beginnen alleen s rood. B wordt de verzameling blauwe knopen. Deze is in eerste instantie leeg. Herhaal nu de volgende procedure: pak een rode knoop uit R, verf deze blauw (voeg deze aan B toe), en vervang die knoop in R door al zijn (directe) opvolgers in de graaf, 'tenzij je ze al gehad hebt', dat wil zeggen: tenzij ze al rood of blauw zijn. Met andere woorden: verf alle niet gekleurde opvolgers rood. Als er niet zulke opvolgers zijn, dan

krimpt R dus. Stop als R leeg is. Als t nu blauw is, dan is t bereikbaar uit s, en anders niet.
Dit algoritme is correct en eindigt in kwadratische tijd ten opzichte van het aantal knopen van de graaf. Dit kunnen we als volgt inzien. Stel n is het aantal knopen in de graaf. Ten hoogste iedere knoop verkleurt een keer van rood naar blauw, ieder van die knopen heeft ten hoogste alle andere knopen als opvolger, en aan het eind moeten we nog even de verzameling B doorlopen, die ten hoogste alle knopen bevat. In totaal zijn dit hooguit $n \cdot (n-1) + n$ stappen. Omdat voor grote n lineaire factoren verwaarloosbaar zijn ten opzichte van kwadraten, en n en $n-1$ dan bijna hetzelfde zijn, zeggen we toch dat de orde van grootte n^2 is.

De kunst van efficiënt programmeren loopt via dergelijke slimme representaties met lage complexiteit. Vergelijk dit ook met voorbeeld 17.1 van 'vind de beroemdheid'.

Een ander genre problemen lijkt een sprong in de complexiteit te vergen. Een standaardvoorbeeld is het zogenaamde handelsreizigers-probleem (*Travelling Salesman*):

Voorbeeld 17.3

Travelling Salesman (TS)
Wat is de kortste rondreis door een gegeven graaf G die alle knopen één keer aandoet?
Om te beginnen moeten we zo'n rondreis vinden. We maken een mogelijke route en testen dan of dit een rondreis is. Tezamen neemt dit in het ergste geval meer dan polynomiaal veel tijd, en het vinden van de *kortste rondreis* lijkt nog complexer. Bijvoorbeeld, als alle knopen met elkaar verbonden zijn, kun je vanuit een beginpunt $n-1$ andere punten bereiken, daarvandaan nog $n-2$ andere, enzovoorts. Er zijn dan dus $(n-1)!$ verschillende rondreizen, en dat is op den duur nog erger dan a^n, voor welke a dan ook (dus zeker erger dan polynomiaal). Laten we, voordat we verder op dit probleem ingaan, onze complexiteitsbegrippen nu iets preciezer omschrijven.

Graden van groei

We tellen het aantal rekenstappen (*tijdscomplexiteit*) of het aantal geheugenplaatsen (*ruimtecomplexiteit*) van een gegeven procedure, met de invoerlengte x als parameter. Hierbij krijgen we onder meer de volgende groeipatronen, die we overal in de wiskunde en informatica zien terugkomen, en ook in toepassingen zoals discussies over economische groei:

P	polynomiale tijd
NP	niet-deterministische polynomiale tijd
Pspace	polynomiale ruimte
Exptime	exponentiële tijd

Bijvoorbeeld $a \cdot x + b$ is *lineair* en $a \cdot x^3$ *derdegraads*, en beide zijn polynomen, dus zitten in **P**. Exponentiële tijd ziet er ten minste uit als $a \cdot 2^x$. Maar het kan nog erger dan **Exptime**, zoals bijvoorbeeld in $x!$ en in 2^{2^x}.

NP

'In **NP** zitten' betekent het volgende. We schrijven eerst een kandidaat-oplossing van de vraag op (en die neemt polynomiale ruimte in beslag ten opzichte van de invoer), en daarna testen we of die kandidaat voldoet als oplossing (en dit vergt polynomiale tijd in de lengte van de invoer).

Voorbeeld 17.4

Graph Reachability, GR, is in **P**, omdat voor een gegeven graaf G de procedure altijd eindigt na hooguit een aantal stappen van een *kwadratische* orde van grootte (namelijk $n \cdot (n-1) + n$).

Voorbeeld 17.5

Travelling Salesman, *TS*, is toch niet zo erg als exponentieel, maar zit in de lagere klasse **NP**. In *TS* is een kandidaat-oplossing een eindige verzameling van paren knopen uit de graaf (er zijn er ten hoogste n^2), en testen of dit een rondreis is, en wat de lengte is, kan ook polynomiaal. In plaats van 'we schrijven een kandidaat-oplossing op' kan men ook denken 'we kiezen een kandidaat-oplossing', oftewel 'bepaal niet-deterministisch een kandidaat-oplossing'. Vandaar **NP**. (Ook het bepalen van de *kortste* rondreis blijkt in **NP** te zitten. We leggen dit niet uit.)

Een logische analogie kan helpen deze begrippen nog iets beter te begrijpen. Hiervoor zagen we al dat propositielogische evaluatie in **P** zit. Testen op propositielogische vervulbaarheid leek complexer en zit in **NP**. Dit kunnen we als volgt inzien: we kunnen vaststellen of er een model is door eerst een geschikte valuatie op te schrijven – dit kan lineair in de lengte van de formule en dit is onze niet-deterministische keuze – en als deze inderdaad de formule waar maakt, kunnen we dat vervolgens in lineaire tijd vaststellen.

Veel andere logische taken vallen eveneens onder **P** of **NP**. Bijvoorbeeld, nagaan of een atomair feit volgt uit een zuiver Booles Prolog-programma is in **P**. Maar het vinden van propositionele Prolog-conclusies in het

algemeen, is ingewikkelder. Veel ontleedalgoritmen voor talen werken met derdegraads tijdscomplexiteit, en leven dus in **P**.
Waar je precies in **P** (of elders) uitkomt, is in de praktijk van het programmeren van groot belang. Een algoritme van lineaire tijd is prettiger dan een van kwadratische tijd, enzovoorts. De kunst van het programmeren bestaat doorgaans uit het geschikt kiezen van datastructuren en instructies, om zo laag mogelijk in rekencomplexiteit uit te komen.

De genoemde complexiteitsklassen liggen in een hiërarchie van stijgende moeilijkheid. Dit is de zogenaamde complexiteitshiërarchie. De volgende inclusies gelden:

Complexiteitshiërarchie

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{Pspace} \subseteq \mathbf{Exptime}$$

Problemen in **P** gelden als doenlijk (in het Engels: 'tractable') en alles daarbuiten als ondoenlijk ('intractable'), hetgeen overigens niemand verhindert allerlei speciale gevallen van **NP**-problemen toch op de computer op te lossen. Het is eenvoudig in te zien dat de inclusies gelden: polynomiale tijd **P** is bevat in polynomiale ruimte **Pspace**, omdat je in die tijd niet meer dan polynomiaal veel geheugenplaatsen kan bezoeken. En **P** is bevat in **NP**, omdat *geen* keuze een speciaal geval is van *wel* keuze.

Of het ook *echte* inclusies zijn, is maar ten dele bekend. Alleen $\mathbf{P} \subseteq \mathbf{Exptime}$ is bewezen. Het hangt af van een fameus open *probleem*, waarvoor onlangs nog een miljoenenprijs werd uitgeloofd:

Geldt $\mathbf{P} = \mathbf{NP}$?

We weten dus nog steeds niet, zoveel jaar na Boole, en na de komst van de computer, of propositionele evaluatie (**P**) nu *echt* eenvoudiger is dan propositionele vervulbaarheid (**NP**)! Hiermee is het simpelste logische systeem van dit boek nog steeds een brandhaard van informaticavragen. Het heersende vermoeden is, dat het antwoord op deze vraag NEE zal blijken te zijn.

Onbeslisbaarheid

Het kan dus, zoals we al zagen, nog veel erger qua complexiteit. En sommige natuurlijke problemen zijn helemaal niet met een mechanische methode op te lossen: deze heten *onbeslisbaar* (zie paragraaf 5.7). Wie zou niet graag willen kunnen vermijden dat zijn computer vastloopt:

Halting Problem

Bepaal bij een willekeurig programma en willekeurige invoer, of het programma voor die invoer termineert.

Feit

Het *Halting Problem* is onbeslisbaar.

Dit feit werd bewezen door Turing in 1936. Er zijn dus geen wondermiddelen die automatisch ongewenst gedrag van onze computers detecteren. Nog zo'n onbeslisbare vraag is bijvoorbeeld die naar de equivalentie van twee gegeven programma's, zeg in de taal STIP van hoofdstuk 14: geven ze op alle invoer dezelfde waarden? De oudste voorbeelden van onbeslisbare problemen komen uit de logica: we zagen bijvoorbeeld al dat het geldigheidsprobleem voor de predikaatlogica onbeslisbaar is (in paragraaf 9.4). De algemene moraal is deze: niet elk eenvoudig formuleerbaar probleem inzake redeneren, rekenen of informatieverwerking heeft een mechanische oplossing!

Ook hier merken we weer op dat de praktijk enig optimisme wettigt. Zelfs onbeslisbare problemen kunnen systematisch worden aangepakt, blijkens methodes als semantische tableaus voor de predikaatlogica. In dit verband maken we nog twee opmerkingen. Ten eerste geven onze voorbeelden in dit hoofdstuk slechts *bovengrenzen*: hoeveel tijd- of ruimtestappen hebben we *hoogstens* nodig, in het *ergste* geval? Om een probleem precies te plaatsen in de complexiteitshiërarchie moeten we ook een ondergrens geven: hoeveel stappen kan de oplossing *minstens* vergen, wat is de *minimale complexiteit*? Wij laten dat hier terzijde. Bovendien wordt het gedrag van een algoritme in de praktijk bepaald door het gedrag op *willekeurige* invoer, zodat we eerder zouden moeten spreken over een *gemiddelde complexiteit*. Ook dat begrip laten we hier terzijde.

We lopen nu nog langs onze voornaamste logische systemen, om de balans tussen hun uitdrukkingskracht en hun complexiteitsprofiel te illustreren.

17.5 COMPLEXITEIT VAN PROPOSITIELOGICA

De antwoorden voor evaluatie en vervulbaarheid volgen direct uit onze eerdere analyse. Verder komt modelvergelijking van twee eindige waarderingen neer op controle dat ze dezelfde formules waar maken, hetgeen natuurlijk slechts het geval is als ze gelijk zijn. Dit vergt slechts een lineaire controle in hun grootte. Het resultaat van dit alles kunnen we als volgt samenvatten. We gebruiken nu verder de standaard Engels-

talige terminologie, dat wil zeggen, *model checking* voor evaluatie (van een formule in een gegeven model), *SAT* (*satisfiability*) voor vervulbaarheid (van een verzameling formules), en *model comparison* voor vergelijking van modellen:

Complexiteitsprofiel propositielogica

Model checking	**P**
SAT	**NP**
Model comparison	**P**

Reductie

Overigens kunnen we ook andersom denken. Propositielogische vragen kunnen worden gekoppeld aan rekencomplexiteit, maar we kunnen dit soort logische problemen ook gebruiken om het rekenen aan andere problemen na te bootsen. Hiertoe maken we gebruik van zogenaamde *reducties*, vertalingen van het ene probleem in het andere, die zelf zo eenvoudig zijn dat ze geen extra rekencomplexiteit introduceren.

Voorbeeld 17.6

Graph Reachability is even moeilijk als propositielogische evaluatie.

Voorbeeld 17.7

Travelling Salesman is even moeilijk als propositielogische vervulbaarheid.

Het bewijs van zulke resultaten bestaat uit twee reducties: bewijzen dat het ene probleem moeilijker is dan het andere, en andersom. Het idee voor de richting van *TS* naar *SAT* is in grote lijnen als volgt. Voor een gegeven graaf G met n knopen voeren we propositieletters $p_{i,j}$ in, voor: 'de i-de knoop op het pad heet j'. Vervolgens beschrijven we alle eisen aan het pad in propositionele formules, bijvoorbeeld:

$p_{i,1} \vee \ldots \vee p_{i,n}$	bij elke knoop op het pad hoort ten minste een knoop uit de graaf (i willekeurig)
$\neg(p_{i,j} \wedge p_{i,k})$	bij elke knoop op het pad hoort ten hoogste een knoop uit de graaf (i willekeurig, $j \neq k$ willekeurig)
$\neg(p_{k,i} \wedge p_{k+1,j})$	de volgende knoop op het pad correspondeert met een directe opvolger in de graaf (i, j, k willekeurig, $\neg\, iRj$)

De verzameling van deze formules (er zijn er nog meer) is vervulbaar dan en slechts dan als de gegeven graaf een rondreis heeft.

In feite geldt zelfs iets veel sterkers dan in de voorbeelden 17.6 en 17.7. Elk probleem dat ligt in de complexiteitsklasse **NP** is efficiënt te vertalen in een propositioneel vervulbaarheidsprobleem ('Stelling van Cook'). En mede gegeven de onzekerheid of **P** = **NP**, is het eigenlijk zelfs andersom:

je bewijst dat een probleem 'ten minste in **NP** zit' (in het Engels: *NP-hard* is) door te laten zien dat een willekeurig **NP**–probleem ernaar vertaalt (met gebruikmaking van resultaten als de genoemde stelling). En eenzelfde reduceerbaarheid geldt voor problemen in **P** ten opzichte van propositielogische evaluatie. Propositielogica lijkt een haast triviaal stukje redeneren binnen de logica. Maar het bevat dus met enige fantasievolle vertaling al vele kernthema's uit de informatica en de kunstmatige intelligentie!

17.6 COMPLEXITEIT VAN PREDIKAATLOGICA

Voor een diepere analyse van systeemgedrag (zoals bij registermachines, hoofdstuk 14), van wiskundig redeneren, of van gewone natuurlijke taal (hoofdstuk 20), hebben we de rijkere taal nodig van de predikaatlogica. Hier is het complexiteitsprofiel voor dit 'werksysteem' van het boek:

Complexiteitsprofiel predikaatlogica

Model checking	**Pspace**
SAT	onbeslisbaar
Model comparison	**NP? P?**

Model checking voor dit systeem vereist polynomiale geheugenruimte, gemeten in de lengte van de invoerformule en de grootte van het model waarop wordt geëvalueerd. Dit kan men inzien door een zorgvuldige analyse van de werking van de waarheidsdefinitie (paragraaf 7.3).

Vervulbaarheid is echter veel ingewikkelder, en breekt naar boven uit de complexiteitshiërarchie, omdat dit probleem equivalent is met testen van geldigheid, zoals we weten van werken met semantische tableaus (zie de de paragrafen 3.3 en 9.4 – een formule is vervulbaar, als zijn negatie niet universeel geldig is, dat wil zeggen: een open tableau heeft). Vervulbaarheid is onbeslisbaar voor de predikaatlogica.

Ten slotte is de modelvergelijkingstaak nog een open probleem. Twee eindige modellen maken dezelfde predikaatlogische formules waar dan en slechts dan als ze *isomorf* zijn. Dit is een nog sterkere structurele notie dan de bisimulatie uit hoofdstuk 13; we geven geen details. Een test op isomorfie tussen twee gegeven eindige modellen (het zogenaamde 'graph isomorphism'–probleem) *kan* worden gedaan in **NP**–tijd. Het is een open vraag of dit probleem ook in polynomiale tijd kan!

17.7 COMPLEXITEIT VAN MODALE LOGICA

Propositie- en predikaatlogica waren er al lang voor de opkomst van de informatica. Ten slotte kijken we naar een logische taal die juist sterk samenhangt met motiveringen uit de informatica zelf: de modale logica. De modale logica staat qua uitdrukkingskracht in tussen de propositie- en de predikaatlogica. Deze tussenpositie vinden we ook terug in het complexiteitsprofiel van de modale logica:

Complexiteitsprofiel modale logica

Model checking	**P**
SAT	**Pspace**
Model comparison	**P**

Modale model checking zit in **P**. De procedure die dit voor elkaar krijgt werkt op slimme manier met de waarheidsdefinitie voor modale formules φ (zie paragraaf 13.3). Gegeven een model M en een formule φ, bepaal 'bottom-up' van elke subformule ψ van φ de verzameling van alle werelden in M die ψ waarmaken, en markeer de uitkomsten op die werelden. Dit kost een aantal ronden, namelijk een aantal gelijk aan het aantal subformules, en dit correspondeert met de lengte van φ. Per ronde hoeft per wereld hoogstens één keer het model te worden afgezocht, en wel indien de betreffende subformule met een modale operator begint; de Boolese gevallen zijn simpeler. Totaal gebruiken we dus een orde van grootte van $|\varphi| \cdot |M|^2$ stappen (voor iedere subformule – $|\varphi|$ is de lengte van de formule – , per wereld – $|M|$ is het aantal werelden van het model – , hooguit één keer het hele model – $|M|$).

Het algoritme voor het construeren van een modaal tableau, waarmee we nagaan of een formule φ vervulbaar is, is **Pspace**. Een volledig uitgeschreven tableau voor φ kan exponentieel groot zijn gemeten in $|\varphi|$. Dit komt door regels die een gegeven tak splitsen. Maar het bestaan van een open tableau voor een formule kan men echter ook *recursief* testen, namelijk met 'backtracking' (net zoals bij de weerleggingsbomen voor een doel in PROLOG, zie paragraaf 16.7), waarbij de nog te verrichten taken met 'pointers' worden gemarkeerd, zodat in het geheugen op zijn hoogst één enkele tak van boven naar beneden hoeft te worden opgeslagen. Op die manier kan men vervulbaarheid testen met polynomiaal gebruik van geheugenruimte, zonder ooit het hele model van de invoerformule te hoeven opschrijven! En hiermee is het **Pspace**.

Modellen die bisimilair zijn, zijn in de taal niet van elkaar te onderscheiden. Zie paragraaf 13.4. Nu lijkt het bestaan van een bisimulatie

een **NP**–probleem, net zoals dat van een *graph isomorphism*, de sterkere gelijkheidseis waarmee we in de predikaatlogica te maken hadden. Maar het 'lossere' karakter van bisimulaties staat een stapsgewijze procedure toe. De crux is om te werken van 'bovenaf', en dan te elimineren. Begin met *alle paren werelden* in de twee modellen kruiselings te verbinden, en laat dan in achtereenvolgende rondes al die paren afvallen die, met betrekking tot de aan het begin van deze ronde nog aanwezige 'kandidaat-verbindingen', ofwel niet aan de atomaire correspondentie–eis van bisimulatie voldoen, ofwel falen in een van de andere twee ('zigzag') eisen. Als we de stappen netjes tellen blijkt de bovengrens voor het aantal bezoeken van werelden in de modellen een zesdegraads polynoom. We zitten dus weer in **P**! Dit 'bisimulatie testen' kan zelfs heel efficiënt: als n de som is der groottes van de gegeven twee modellen, kan het in orde van grootte $n \cdot \log(n)$.

Deze complexiteitsuitkomsten hebben allerlei praktische consequenties. Zo wordt model checking met modale talen veel gebruikt voor de verificatie van ontwerpen voor processen. Evenzo wordt het testen op bisimulatie gebruikt in het vereenvoudigen van gegeven ontwerpen voor processen: het hiervoor geschetste algoritme kan namelijk ook worden gebruikt om de kleinste machines te vinden die tot op bisimulatie hetzelfde gedrag vertonen als een, wellicht ingewikkelder, 'prototype'.

Dynamische logica

Ten slotte kijken we nog even naar een uitbreiding van de modale logica, de *dynamische logica*. Deze werd in paragraaf 15.5 ingevoerd, ter beschrijving van programmagedrag, en bevatte ook modale operatoren voor de onbegrensde iteratie van programmauitvoering. Deze taal past niet helemaal in het rijtje tot nog toe, omdat de iteratie-modaliteit niet definieerbaar is in de predikaatlogica. Niettemin vermijdt het complexiteitsprofiel van de dynamische logica nog steeds de onbeslisbaarheid:

Complexiteitsprofiel dynamische logica

Model checking	**P**
SAT	**Exptime**
Model comparison	**P**

We leggen deze uitkomsten verder niet uit. Hiermee besluiten we de uitweiding over de complexiteit van logica's.

17.8 CONCLUSIES

Dit hoofdstuk liet een innig verband zien tussen logica en informatica.

Logische taken zijn rekentaken, met een uiteenlopend algoritmisch karakter dat zich uit in verschillende rekencomplexiteiten wanneer we ze implementeren. Deze complexiteit wordt meestal hoger (dus 'slechter') naarmate de uitdrukkingskracht van de logische taal groter (dus 'beter') is. Bij het modelleren van concrete problemen moeten we dus steeds het een tegen het ander afwegen.

Maar andersom zijn rekentaken in de informatica ook logische taken, zoals we zien door reducties van standaard rekenproblemen tot logische evaluatie– of vervulbaarheidsproblemen.

Overigens was onze behandeling slechts een schets, en dit hele gebied is in snelle ontwikkeling. Zo leidt de opkomst van gedistribueerd rekenen en communicatiemedia zoals het internet tot nieuwe interessante noties, bijvoorbeeld de 'communicatie-complexiteit' van gedistribueerde systemen en daarmee ook van logische systemen die er mee samenhangen, zoals de kennislogica die in hoofdstuk 19 aan bod komt.

17.9 OPGAVEN

17.1 Gegeven is een Prolog programma Π met propositionele clausules, en een doel $\leftarrow B$. Bepaal de complexiteit van de vraag of $\leftarrow B$ uit Π volgt.

17.2 Bekijk een modale deeltaal waarin de operatoren $\Box$ en $\Diamond$ nooit gestapeld voorkomen. Dus wel formules als $(\Box p \wedge q) \vee \neg\Diamond r$, maar nooit formules als $\Box\Diamond p$ en $\Box(p \wedge \Box q)$, enzovoorts. Deze restrictie komt niet zelden voor in concrete toepassingen. Wat wordt nu de complexiteit van model checking, satisfiability, en model comparison?

* 17.3 Laat zien dat het algoritme voor model comparison in de modale logica, zoals in paragraaf 17.7 uitgelegd, inderdaad op een zesdegraads polynoom uitkomt.

Blok VI

Logica en kunstmatige intelligentie

Kennisrepresentatie met tijdslogica

Hoofdstuk 18

Kennisrepresentatie met tijdslogica

18.1 INLEIDING

In het volgende blok van drie hoofdstukken houden we ons bezig met kunstmatige intelligentie. We vatten kunstmatige intelligentie op als de tak van informatica die zich bezighoudt met het mechaniseren van intelligente menselijke verrichtingen, zoveel mogelijk maar niet noodzakelijk op basis van analogie. Gegeven het werkterrein van de logica zijn er twee voor de hand liggende raakpunten tussen de logica en de kunstmatige intelligentie: redeneren en semantiek.

Redeneren

Redeneren is een van de meest centrale verschijnselen in intelligent gedrag. In de kunstmatige intelligentie is daarom veel gewerkt aan het implementeren van logische bewijssystemen. Bekende voorbeelden zijn het automatisch bewijzen van stellingen (waaruit logisch programmeren is voortgekomen) en de verschillende redeneermechanismen voor kennissystemen (expertsystemen). In dit blok zullen we enige standaard axiomatische systemen presenteren voor toegepaste modale logica's. Behalve overeenkomsten bestaan er echter ook grote verschillen tussen de wiskundig–logische redeneertrant zoals die in de vorige blokken van dit boek is uiteengezet, en de 'cognitieve vuistregels' waarmee mensen alledaagse problemen te lijf gaan. Die verschillen geven aanleiding tot logische systemen die afwijken van de standaardlogica (vergelijk hoofdstuk 12). Dit soort niet–standaard logica's is zelf weer met logische middelen te beschrijven. We besteden er enige aandacht aan in het laatste hoofdstuk van het boek, maar niet in dit blok.

Semantiek

Behalve een analyse van redeneren heeft de logica nog iets te bieden aan de kunstmatige intelligentie, namelijk een heldere *semantiek*. In de kunstmatige intelligentie wordt semantiek vaak 'kennisrepresentatie' genoemd: logische modellen coderen (representeren) conceptuele structuren (kennis) waarmee intelligente systemen opereren. Naarmate de relaties tussen een geprogrammeerd cognitief systeem en het probleem of de taak waarvoor het gecreëerd is ingewikkelder worden,

heeft men steeds meer baat bij een dergelijke logische beschrijving van dat probleem. Het thema kennisrepresentatie vertoont overeenkomsten met de semantiek van programmeren (zie hoofdstuk 15), maar verkrijgt in de kunstmatige intelligentie een eigen karakter door de speciale keuze van belangrijke kennisstructuren. We concentreren ons in dit blok vooral op dergelijke vernieuwende vormen van kennisrepresentatie. Drie belangrijke kennisstructuren uit de studie van actie, planning en communicatie zijn *tijd*, *kennis* en *natuurlijke taal*. In dit hoofdstuk zullen we beginnen met een gangbare representatie van tijd in modale logica. Dit is zeker niet de enige manier waarop tijd en logica in de kunstmatige intelligentie overlappen: in paragraaf 18.5 bespreken we kort enige gebieden die we hebben laten liggen. In hoofdstuk 19 zullen we ons concentreren op de representatie van kennis in modale logica. In hoofdstuk 20 ten slotte zal worden ingegaan op de representatie en machinale verwerking van natuurlijke taal. Het laatste is een traditioneel werkterrein van de kunstmatige intelligentie dat goed illustreert hoe de behandeling van een complex verschijnsel inzet van vele logische technieken tegelijk vereist.

Overigens is er vaak een vloeiende overgang tussen logische onderwerpen in kunstmatige intelligentie en verwante kwesties in de theoretische informatica.

18.2 TIJDSLOGICA

In hoofdstuk 13 werd de modale logica geïntroduceerd. Als we extra eisen opleggen aan de toegankelijkheidsrelatie, beperken we ons als het ware tot een speciale modelklasse. Daarmee krijgen de modale operatoren $\Box$ en $\Diamond$ dan direct een meer specifieke betekenis. Het kan ook voorkomen dat er ter wille van een goede beschrijving extra operatoren aan de modale taal toegevoegd worden. Tevens kunnen we ook een aparte betekenis aan de 'werelden' geven.
In hoofdstuk 15 hebben we hier al een voorbeeld van gezien: de werelden waren daar geheugentoestanden, de toegankelijkheid werd gerelateerd aan de uitvoering van een imperatief programma, en voor ieder programma π was er een aparte modale operator $[\pi]$.

Een andere toegepaste modale logica is de *tijdslogica*. In dit geval staan de werelden voor *tijdstippen*, en R voor de fundamentele temporele relatie *'eerder dan'* (notatie: $<$). Een andere keuze in de literatuur dan voor tijd*stippen* is die voor tijd*sintervallen* met daarbij passende relaties. Hierop komen we nog terug in paragraaf 18.6.

De modale operatoren F, G, P en H

De modale operatoren $\Box$ en $\Diamond$ krijgen nu de betekenis van 'voortaan' respectievelijk 'eens (in de toekomst)'. Beide operatoren 'kijken' dus naar de toekomst. In de tijdslogica gebruikt men veelal in plaats van $\Box$ de letter G ('Going to'), en in plaats van $\Diamond$ de letter F ('Future').
Om ook uitspraken over het verleden te kunnen doen, introduceren we twee extra operatoren, P en H. De operator P ('Past') wordt geïnterpreteerd als 'ooit (vroeger)' en H ('Has been') als 'tot nu toe'. Al deze operatoren vormen formules als eerder:

DEFINITIE 18.1

Temporele formule
Als φ een formule is, dan ook $G\varphi$, $F\varphi$, $H\varphi$ en $P\varphi$.

DEFINITIE 18.2

Waarheidsdefinitie voor temporele formules
a $t \models G\varphi \Leftrightarrow$ voor alle t' met $t < t'$ geldt $t' \models \varphi$
b $t \models F\varphi \Leftrightarrow$ er is een t' zodat $t < t'$ en $t' \models \varphi$
c $t \models H\varphi \Leftrightarrow$ voor alle t' met $t' < t$ geldt $t' \models \varphi$
d $t \models P\varphi \Leftrightarrow$ er is een t'zodat $t' < t$ en $t' \models \varphi$

In feite hebben we natuurlijk maar twee verschillende modale operatoren aan onze taal toegevoegd, namelijk G en H. De operatoren F en P, respectievelijk, kunnen hieruit gedefinieerd worden, net zoals $\Diamond$ uit $\Box$.

In de rest van deze paragraaf zullen we een aantal formules bespreken die kunnen dienen als axioma's voor een tijdslogica. Om te beginnen gold reeds op elk frame distributiviteit der modale operatoren, dus zeker ook op elk tijdsframe:

$G(\varphi \rightarrow \psi) \rightarrow (G\varphi \rightarrow G\psi)$
$H(\varphi \rightarrow \psi) \rightarrow (H\varphi \rightarrow H\psi)$

Het eerder (paragraaf 13.7) beschouwde $\Box \varphi \rightarrow \varphi$ is echter niet aannemelijk. Reflexiviteit van de toegankelijkheidsrelatie zou temporeel het ongewenste principe inhouden dat elk tijdstip aan zichzelf voorafgaat.

Transitiviteit

Wel plausibel is het principe $\Box \varphi \rightarrow \Box\Box\varphi$, aangezien 'eerder dan' inderdaad transitief is. Dus hebben we:

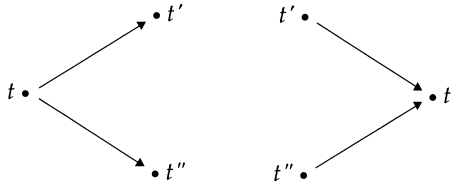
$G\varphi \rightarrow GG\varphi$
$H\varphi \rightarrow HH\varphi$

Lineariteit

Een andere zinvolle eis aan de tijd is dat zij niet kan 'splitsen'. Anders gezegd: van elk tweetal verschillende tijdstippen is een van beide eerder dan de andere. Deze *lineariteit* wordt uitgedrukt door de formules:

$F\varphi \rightarrow G(P\varphi \vee \varphi \vee F\varphi)$
$P\varphi \rightarrow H(P\varphi \vee \varphi \vee F\varphi)$

De eerste formule zegt dat de tijd niet kan splitsen naar de toekomst; de tweede zegt dat de tijd niet kan splitsen naar het verleden. De eerste maakt dat het linkerplaatje niet als onderdeel van een tijdsframe voorkomt, de tweede verbiedt het rechterplaatje:



Voorbeeld 18.1

Op een *begrensd* lineair tijdsframe is er een laatste tijdstip. Hierop geldt dus $FG\bot$: ooit in de toekomst is er een tijdstip (namelijk het laatste) waarna geldt dat *als* er een later tijdstip is, *dan* geldt daar (de absurde bewering) $\bot$. Uiteraard is $\bot$ altijd onwaar, maar de implicatie geldt omdat aan de conditie niet voldaan kan worden. Evenzo geldt $PH\bot$. Op een *onbegrensd* lineair tijdsframe geldt bijvoorbeeld het schema $PF\varphi \rightarrow FP\varphi$: alles wat in het verleden ooit het geval werd, zal ooit zelf in het verleden liggen. Ten opzichte van een tijdstip t in een lineair frame waarvoor $t \vDash PF\varphi$, kan het tijdstip t' dat φ waar moet maken om de geldigheid van dit schema te bewijzen, zowel eerder als later dan t zijn. Overigens is dit schema *geen* karakteristiek van lineariteit: $PF\varphi \rightarrow FP\varphi$ beschrijft eerder een soort confluentie van tijd (zoals in paragraaf 12.6).

Er kunnen nog meer eisen gesteld worden aan de temporele ordening, afhankelijk van de beoogde toepassing. Een voorbeeld is 'dichtheid' – $\forall x \forall y\ (Rxy \rightarrow \exists z\ (Rxz \wedge Rzy))$ – die correspondeert met de converse van het transitiviteitsaxioma: $GG\varphi \rightarrow G\varphi$.

18.3 PARALLELLE PROCESSEN

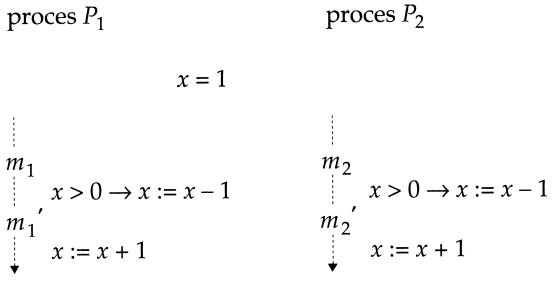
We gaan nu iets dieper in op een, uiteraard informatica-gerichte, toepassing van tijdslogica in kennisrepresentatie. De modale tijdslogica kan worden gebruikt in beweringen over het verloop van sequentiële of parallelle rekenprocessen. Dit is natuurlijk een thema binnen de

algemene informatica, maar ook binnen de kunstmatige intelligentie, bijvoorbeeld wanneer we planning beschrijven voor groepen processoren via een protocol.

Parallele processen

Een parallel proces Π bestaat uit n processen $P_1, ..., P_n$ die gemeenschappelijk toegang hebben tot k programmavariabelen $y_1, ..., y_k$. Wanneer T_i de verzameling van 'lokale' toestanden is waarin proces P_i kan verkeren, dan bestaat een 'globale' toestand T van het hele systeem uit een rijtje $\langle m_1, ..., m_n; a_1, ..., a_k \rangle$, waarbij $m_i \in T_i (1 \leq i \leq n)$ en a_j $(1 \leq j \leq k)$ een mogelijke waarde is van programmavariabele y_j. Een verwerkingsstap van het parallelle programma correspondeert met een stap van precies een van de deelprocessen of factoren P_i van Π. De globale toestand waarin Π verkeert ná een verwerkingsstap verschilt dus in precies één coördinaat m_i van de toestand vóór die stap (afgezien van wijzigingen in de programmavariabelen). De deelprocessen van Π beïnvloeden elkaars verloop via de gemeenschappelijke variabelen. Een voorbeeld van een proces en een aantal verwerkingsstappen daarin:

Voorbeeld 18.2



Deze gelabelde graaf beschrijft een deel van een programma Π dat bestaat uit twee processen P_1 en P_2. De formule $x > 0$ is een voorwaarde voor de overgang van toestand m_1 naar toestand m_1' en van m_2 naar m_2': wordt hieraan voldaan, dan kan P_1 naar de volgende toestand gaan. De toekenningen $x := x + 1$ en $x := x - 1$ aan de programmavariabelen worden uitgevoerd wanneer de desbetreffende overgang plaatsvindt. Wat voor beweringen kunnen we nu over dit proces doen? We kunnen bijvoorbeeld opmerken dat P_1 en P_2 zich niet tegelijkertijd in toestand m_1' respectievelijk m_2' kunnen bevinden. Stel dat we bijvoorbeeld proces P_1 binnenkomen via de beginwaarde $x = 1$. Aangezien geldt dat $x > 0$, gaat nu proces P_1 over van toestand m_1 naar toestand m_1'. De variabele x wordt dan op 0 gezet. Zolang P_1 in toestand m_1' blijft, verhindert dit laatste de overgang van m_2 naar m_2' in proces P_2. Pas wanneer P_1 de toestand m_1' verlaat, is de overgang van m_2 naar m_2' in

P_2 mogelijk, x wordt dan immers op 1 gezet zodat aan de voorwaarde $x > 0$ voldaan wordt.
De volgende reeks drietallen geeft dus een deel van een mogelijke uitvoering weer van dit concrete programma (neem aan dat proces 1 bij het verlaten van toestand m_1' een toestand m_1'' aanneemt):

$$..., \langle m_1, m_2; 1\rangle, \langle m_1', m_2; 0\rangle, \langle m_1'', m_2; 1\rangle, \langle m_1'', m_2'; 0\rangle, ...$$

Beweringen over het gedrag van een proces kunnen we weergeven in een modaal tijdslogisch formalisme. Vervolgens kunnen we met de semantiek daarvan redeneren over de werking van parallelle systemen.

18.4 PARALLELLE PROCESSEN: BESCHRIJVING IN TIJDSLOGICA

We zullen om te beginnen een tijdslogische structuur definiëren die kan dienen als representatie van programma–activiteit.

Uitvoering

Laat P een programma zijn en T de verzameling 'globale toestanden' van P. Een *uitvoering* s van P is een rij opeenvolgende elementen van T, met beginpunt maar zonder eindpunt (terminatie van een programma betekent dat de uitvoering wordt afgesloten met een oneindige rij lege stappen). Zo'n rij representeert een mogelijke verwerking van programma P in termen van de achtereenvolgens doorlopen toestanden. De verzameling van alle uitvoeringen noemen we S. Met $s(i)$ bedoelen we het i–de element van s, en met $s\,|i$ het staartstuk van s dat begint bij $s(i)$. De relatie $\leq$ (s' is een staartstuk van s, notatie $s \leq s'$) is een partiële orde op S. Een uitvoering van een proces kan men verbeelden als een 'mogelijke geschiedenis' voor het proces, bestaande uit een reeks tijd*stippen* i.
Merk nog op dat hoewel iedere $s \in S$ lineair is en niet splitst (iedere $s(i)$ heeft één opvolger $s(i + 1)$) het programma P niet deterministisch hoeft te zijn: voor twee uitvoeringen s en s' kan gelden $s(i) = s'(i)$, maar $s(i + 1) \neq s'(i + 1)$.

DEFINITIE 18.3

Uitvoeringsstructuur
Een *uitvoeringsstructuur* voor een programma P is een paar $F = \langle S, \leq\rangle$, waarbij S de verzameling van alle mogelijke uitvoeringen voor P is en $\leq$ de 'staartstuk'–ordening op S.

Vervolgens kiezen we een bij deze structuur passend tijdslogisch formalisme. Beschouw een propositionele taal waaraan zinsoperatoren G, X zijn toegevoegd. $G\varphi$ heeft de reeds bekende modale interpretatie 'φ is in alle toekomstige toestanden het geval', $X\varphi$ is een nieuwe modale

formule (X heet de neXt–operator), met als intuïtieve betekenis: 'φ is het geval in de volgende toestand'.

Waarheidsdefinitie

De waarheidsdefinitie voor deze taal werkt dan als volgt. Een waardering V beeldt iedere atomaire propositie p af op de verzameling van globale toestanden waarin p waar is. We nemen aan dat de taal atomen $p(i, j)$ bezit voor elke lokale toestand j waarin een deelproces P_i zich kan bevinden. Een 'uitvoeringsmodel' M is dan een paar (F, V), waarbij F een uitvoeringsstructuur is. Nu ontstaat de volgende inductie:

DEFINITIE 18.4

Waarheidsdefinitie

Voor atomaire beweringen kijken we naar het begin van een uitvoering:

- $M, s \models p \Leftrightarrow s(0) \in V(p)$

Propositionele connectieven hebben hun standaardinterpretatie.

Voor tijdslogische operatoren G en X definiëren we:

- $M, s \models G\varphi \Leftrightarrow$ voor alle s' met $s \leq s'$: $M, s' \models \varphi$
- $M, s \models X\varphi \Leftrightarrow M, s|1 \models \varphi$

Net als hiervoor definiëren we $F\varphi = \neg G\neg\varphi$, met als afgeleide betekenis 'voor minstens een toekomstig verloop'. In tegenstelling tot het voorgaande is de tijdsordening ($\leq$) hier dus reflexief.

Programmaeigenschappen

We zullen nu enkele voorbeelden geven van programmaeigenschappen die zijn uit te drukken in de tijdslogische taal:

Voorbeeld 18.3

Wederzijdse uitsluiting

$M, s \models G\neg(p(i, n) \wedge p(j, n'))$: in uitvoering s kan Π nooit gelijktijdig toegang krijgen tot toestand n van deelproces P_i en toestand n' van deelproces P_j.

In het bijzonder geldt deze eigenschap in voorbeeld 18.2:

$M, s \models G\neg(p(1, m_1') \wedge p(2, m_2'))$ voor alle s.

Voorbeeld 18.4

Toegankelijkheid

$M, s \models G(p(i, n) \rightarrow Fp(j, n'))$: als Π in uitvoering s ooit n bereikt met P_i, dan zal P_j daarna nog n' bereiken.

Ook correctheidsbeweringen (zie hoofdstuk 15) zijn tijdslogisch te formuleren. Laat m_0 de begintoestand beschrijven van programma Π en m_e de eindtoestand. Een correctheidsbewering zegt nu dat vertrek vanuit een globale toestand die aan preconditie φ voldoet, ons na verwerking van Π brengt naar een toestand die aan de postconditie ψ voldoet:

Correctheid
Voor alle s: $M, s \models (m_0 \wedge \varphi) \rightarrow G(m_e \rightarrow \psi)$.

Terminatie
Willen we ook terminatie garanderen van Π, dan moet tevens gelden:
Voor alle s: $M, s \models m_0 \rightarrow Fm_e$.

18.5 PARALLELLE PROCESSEN: EEN TEMPOREEL BEWIJSSYSTEEM

In de vorige paragraaf hebben we een logische semantiek gegeven voor temporeel gedrag in uitvoeringsstructuren. Geldig redeneren in deze situatie is volledig te beschrijven door het axiomatisch systeem DX. Behalve de afleidingsregels MP (Modus Ponens) en N, en de versies voor G en X van de modale axioma's K1 en K2 (zie hoofstuk 13), bevat DX de axioma's:

DEFINITIE 18.5

Het axiomatisch systeem DX
Functionaliteit van X
$X\neg\varphi \leftrightarrow \neg X\varphi$
Verband tussen G en X
$G\varphi \rightarrow XG\varphi$
Inductie
$G(\varphi \rightarrow X\varphi) \rightarrow (\varphi \rightarrow G\varphi)$
Reflexiviteit van G
$G\varphi \rightarrow \varphi$

In afleidingen van eigenschappen van programma's Π als geformuleerd in de vorige paragraaf, kan men dit systeem dus als 'motor' gebruiken. Als voorbeeld geven we een afleiding van $X\varphi \rightarrow F\varphi$ (als φ het volgende tijdstip geldt, dan geldt φ ooit in de toekomst):

Voorbeeld 18.5

De operator F komt in geen van de axioma's van DX voor. Herschrijf F dus als $\neg G\neg$. We dienen dan te bewijzen dat $X\varphi \rightarrow \neg G\neg\varphi$. Nu is het handiger om een negatie voor de operator X te hebben dan voor de operator G, gezien het functionaliteitsaxioma. Vandaar dat we beter eerst $G\neg\varphi \rightarrow \neg X\varphi$ kunnen trachten aan te tonen, waaruit dan met propositielogica het gevraagde volgt. Uit $G\neg\varphi \rightarrow \neg X\varphi$ volgt met behulp van functionaliteit $G\neg\varphi \rightarrow X\neg\varphi$. Om $G\neg\varphi \rightarrow X\neg\varphi$ aan te tonen, volstaat om te bewijzen dat $G\varphi \rightarrow X\varphi$:

1	$G\varphi \rightarrow \varphi$	reflexiviteit
2	$X(G\varphi \rightarrow \varphi)$	1, necessitatie voor X
3	$XG\varphi \rightarrow X\varphi$	2, distributie voor X

4	$G\varphi \rightarrow XG\varphi$	axioma $G\varphi \rightarrow XG\varphi$
5	$G\varphi \rightarrow X\varphi$	4, 3, propositielogica

Dit besluit onze bespreking over het gebruik van tijdslogica bij de beschrijving van parallelle processen.

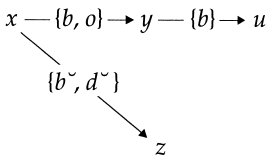
18.6 MEER OVER TIJD

In deze paragraaf besteden we kort aandacht aan andere voorbeelden van tijd in kennisrepresentatie.

Tijdsintervalnetwerken

Een bekende representatie van temporele informatie zijn zogenaamde 'temporele netwerken', waarin gebeurtenissen en feiten geïndexeerd worden met *tijdsintervallen*: de perioden waarin ze plaatsvinden. Dit is dus een alternatief voor tijdsrepresentatie met *tijdstippen*, zoals ook al in paragraaf 18.2 werd aangeduid. In zo'n netwerk kan de temporele structuur worden vastgelegd van zinnen als 'toen Elvis binnenkwam, was ik nog niet vertrokken', maar ook bijvoorbeeld de temporele ordening van deelprocessen in een industrieel proces. Hiermee kunnen vervolgens dynamisch–temporele gevolgtrekkingen worden beschreven die we maken over relaties tussen gebeurtenissen, of toestanden bij het begrijpen van een tekst of het oplossen van een planningsprobleem. Formeel vereist dit structuren als in het volgende voorbeeld.

Voorbeeld 18.6



De 'knopen' x, u, z in het netwerk representeren tijdsintervallen, die corresponderen met bepaalde gebeurtenissen. Tijdsintervallen zijn verbonden door een gerichte pijl met een 'label' die de aard van de temporele relatie uitdrukt, weergegeven in een of meer zogenaamde 'basislabels' zoals b, o en $d^{\smile}$ (van respectievelijk *b*efore, *o*verlap en de converse relatie – vandaar $d^{\smile}$ – van *d*uring). Bijvoorbeeld y — $\{b\}\rightarrow$ u drukt uit dat y geheel aan u voorafgaat (en y — $\{b^{\smile}\}\rightarrow$ u zou uitdrukken dat u geheel aan y voorafgaat). Dit netwerk kunnen we dus als volgt beschrijven:

- x gaat vooraf aan y of x overlapt met y
- y gaat vooraf aan u
- x omvat z of komt na z

Op een tijdsas kunnen we de hier opgevoerde relaties b, d en o tussen tijdsintervallen als volgt representeren:

notatie	*betekenis*	*grafische interpretatie*
$x\ b\ y$	(geheel) voorafgaan	— x — .... y ....
$x\ d\ y$	(echt) omvat worden	— x — ..... y .....
$x\ o\ y$	vooraan overlappen	— x — ..... y .....

Gegeven drie tijdsintervallen en twee intervalrelaties daartussen, kunnen we de derde relatie berekenen. Bijvoorbeeld, uit x — $\{b, o\}\rightarrow$ y en y — $\{b\}\rightarrow$ u volgt y — $\{b\}\rightarrow$ u.

Met dit soort temporele netwerken corresponderen ook formele structuren die met een predikaatlogische theorie beschreven kunnen worden. We geven geen details, maar merken alleen op dat dit niet triviaal is: stel dat x voor y, y voor z, en z voor x, dan 'lijkt' dit wel een netwerk maar het is evident dat deze informatie inconsistent is. Er moet dus meer, formeel, werk verricht worden.

Het gebruik van tijdsintervallen voor het oplossen van problemen in de kunstmatige intelligentie, en in het bijzonder voor productieplanning, is in de jaren tachtig onderzocht door James Allen.

Reactieve systemen

Model checking met temporele logica

Het gebruik van tijdslogica voor het beschrijven van processen (zoals ook in de vorige paragrafen) en programmaeigenschappen, en de ontwikkeling van verschillende computationele technieken hiervoor, is in de informatica geïntroduceerd door Amir Pnueli. Hij heeft voor zijn werk zelfs de Turing-award gekregen, de hoogste onderscheiding die een informaticus kan krijgen (uiteraard is er geen Nobelprijs voor de informatica!). Hij heeft met tijdslogica de specificatie en verificatie van *reactieve systemen* behandeld (dit zijn systemen waarvoor, kort gezegd, terminatie niet wenselijk is, zoals een koffieautomaat). Tevens heeft hij zich beziggehouden met de *model checking* (zie hoofdstuk 17) van temporele eigenschappen van eindige modellen.

Weer een hele andere, en niet modaal-logische, manier om tijd in logica te beschrijven is door middel van ofwel extra parameters bij predikaatlogische beschrijvingen, ofwel hoger–niveau beschrijvingen waarin we in een taal over de waarheid van beweringen op een gegeven tijdstip kunnen spreken. We lichten dit toe aan een in de kunstmatige intelligentie populair voorbeeld: een 'blokkenwereld' waarin verschillende blokken op en onder elkaar liggen, en waarin we configuraties willen kunnen veranderen.

Voorbeeld 18.7

In een gegeven systeemtoestand kan waar zijn dat blok a op blok b ligt, te noteren als $Bovenop(a, b)$. Om weer te geven dat dit alleen waar is op tijdstip t, kunnen we deze notatie uitbreiden tot $Bovenop(a, b, t)$. En in plaats daarvan kunnen we ook een niveau verspringen, en spreken van $Waar(Bovenop(a, b), t)$. In plaats van aan tijdstippen, kunnen we bij t ook denken aan *situaties*, die de toestand van de wereld op dat tijdstip beschrijven. Door het uitvoeren van acties gaat een gegeven situatie in een andere over. Stel, bijvoorbeeld, we pakken nu blok a op in de gegeven situatie t. De situatie t' die hierdoor ontstaat beschrijven we als $Resultaat(Pakop(a), t)$. We willen dan dat $\neg Waar(Bovenop(a, b), Resultaat(Pakop(a), t))$. Anderzijds willen we dat de blokken c en d, die, nemen we even aan, ook op elkaar lagen, in deze volgende situatie $Resultaat(Pakop(a), t)$ nog steeds op elkaar liggen. Maar door het expliciet maken van de tijd, is dit nu helaas niet meer vanzelfsprekend gegarandeerd! Dit staat in de kunstmatige intelligentie bekend als het *frame problem*. De rijkere kennisrepresentatie waarin tijd expliciet is, leidt dus zelfs tot nieuwe problemen waar we voorheen niet mee te maken hadden.

Frame problem

Situation calculus

Het berekenen van de logische beschrijving van door acties relateerde situaties, zoals in voorbeeld 18.7, staat bekend als *situation calculus*. Dit is werk van de bekende AI–onderzoeker John McCarthy (tevens de uitvinder van de functionele programmeertaal LISP, zie hoofdstuk 12).

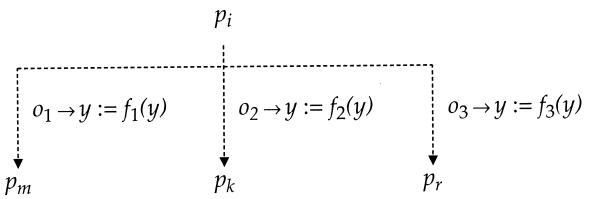
Hiermee is de rol van tijd in logica en kennisrepresentatie verre van uitputtend behandeld. Er zijn ook nog modale logica's waarin tijd en *kennis* gecombineerd worden, min of meer zoals tijd en programmaverwerking gecombineerd worden in de tijdslogica van processsen, besproken aan het begin van deze paragraaf. De logica van kennis is ter behandeling in het nu volgende hoofdstuk.

18.7 OPGAVEN

18.1 Laat zien dat $\langle \mathbb{N}, < \rangle \models H(H\varphi \rightarrow \varphi) \rightarrow H\varphi$.

* 18.2 Op het tijdsframe $\langle \mathbb{R}, < \rangle$ bestaan 15 verschillende 'tijden', dat wil zeggen, 15 niet–equivalente stapelingen van de operatoren F, P, G en H ('Hamblin's 15–tijden–stelling').
Bijvoorbeeld, PF ('ergens') is equivalent met FP en GH ('altijd') met HG. Maar ook is PFH te reduceren tot PH: als op een of ander moment geldt dat tot dan iets altijd het geval was, dan bestaat zeker in het verleden zo'n moment.
Bepaal, aan de hand van dergelijke equivalenties en reducties, de 15 tijden.

18.3 We gaan uit van een parallel proces Π bestaande uit twee deelprocessen P en Q. In onze tijdslogische taal beschikken we over atomaire formules voor alle toestanden p_i van P, q_i van Q en voor de overgangsvoorwaarden o_i die voorkomen in P of Q.
a Formuleer in deze taal de claim dat het onmogelijk is dat P in toestand p_i blijft steken als de overgangsvoorwaarde o_i vanuit p_i oneindig vaak vervuld wordt ('fair scheduling').
b Laat $Xy = f_i(y)$ uitdrukken dat de waarde van variabele y in de volgende toestand de waarde is van $f_i(y)$ in de huidige toestand. Formaliseer nu de bewering die opgaat op die momenten dat er een overgang gemaakt wordt van toestand p_i naar een van de volgende toestanden:



18.4 a Laat zien dat het functionaliteitsaxioma voor X in het DX–systeem opgaat dan en slechts dan als iedere toestand precies één directe opvolger heeft.
b Neem aan dat het systeem DX correct is met betrekking tot de semantiek gegeven op pagina 290. Laat nu zien dat $G(Xp \rightarrow p) \rightarrow (Fp \rightarrow Xp)$ niet bewezen kan worden in het systeem DX.
c Laat zien dat het inductieschema opgaat voor G als de opvolgerrelatie de transitieve afsluiting is van de directe opvolgerrelatie.

Multi–agentsystemen

Hoofdstuk 19

Multi–agentsystemen

19.1 INLEIDING

Een speciaal geval van kennisrepresentatie is de representatie van de kennis van individuele 'agents' in een informatiesysteem. Bij agents (actoren, kenners) kan men zich personen maar ook processoren voorstellen. De weergave van hun kennis is niet vanzelfsprekend, omdat we hierbij mede in overweging moeten nemen wat agents van elkaar weten en welke achtergrondkennis ze gemeenschappelijk hebben. Het wordt nog lastiger als we tevens de dynamiek van het systeem willen beschrijven: hoe de kennis van agents verandert als gevolg van aanvullende informatie, bijvoorbeeld in de vorm van acties van andere agents in het systeem. Een informatiesysteem dat uit meerdere agents bestaat is een 'multi–agentsysteem'. In dit hoofdstuk beschrijven we het gebruik van een bijzondere modale logica, namelijk kennislogica, voor het specificeren van multi–agentsystemen.

Paragraaf 19.2 geeft enige achtergrond over multi–agentsystemen, in paragraaf 19.3 introduceren we de kennislogica voor één agent en in paragraaf 19.4 die voor meer dan een agent. In paragraaf 19.5 breiden we de logica uit met operatoren voor groepen agents, en ten slotte geven we in paragraaf 19.6 een verdere uitbreiding van deze logica, namelijk met dynamische modale operatoren voor het beschrijven van informatieveranderingen.

19.2 MULTI–AGENTSYSTEMEN

Voorbeeld 19.1

Een multi–agentsysteem voor voetbal bestaat uit 22 'agents' of actoren. Een voetballer kan alleen spelers in zijn gezichtsveld waarnemen. Speler a neemt aan dat speler b, een tegenstander, zich achter hem bevindt, omdat dit in de vorige toestand van het spel, voordat a de bal toegespeeld kreeg, het geval was. Dit is een redelijke aanname, waarvan a weet dat ze ongefundeerd is. In werkelijkheid bleek b zich inmiddels elders op het veld te bevinden, buiten a's gezichtsveld. De spelsituatie

waarin b achter a is, is voor a niet te onderscheiden van de situatie waarin b niet achter hem is. Speler a ziet speler c – uit zijn eigen team – voor zich, en speelt c de bal toe om te verhinderen dat b de bal neemt. Wat blijkt: de toegespeelde speler krijgt de bal wel, alleen was het niet c maar d – gelukkig van hetzelfde team: a geloofde dat het c was, maar moet nu zijn kennis herzien. In een ander scenario mist speler c de bal, omdat a en c geen oogcontact hadden tijdens het schot van a: ze hadden geen gemeenschappelijke kennis van de situatie die tot dit schot leidde. Kennelijk zijn er allerlei zaken van logisch belang die we moeten kunnen formaliseren als we voetbal simuleren als 22 communicerende computerprocessen, in plaats van echte spelers. Dit soort 'virtual league' voetbal wordt ook echt bedreven, en er zijn zelfs, zoals de naam suggereert, competities in. Wat is een multi–agentsysteem?

Een multi–agentsysteem is een aantal communicerende computer– of *informatiesystemen* die *doelgerichte* en *autonome* interactie hebben met elkaar en met een welgedefinieerde *omgeving*. Vanuit logisch gezichtspunt is dit een veel te algemeen perspectief. In het bijzonder willen we ons hier niet bekommeren om op zich essentiële aspecten zoals hoe de feitelijke communicatie met de omgeving en met andere agents plaatsvindt. We beperken ons tot het representeren van abstracte architecturen voor multi–agentsystemen en het beschrijven van de kennis en acties van hun agents: de *logische specificatie* van multi–agentsystemen. Vanuit dit abstracte perspectief nemen we 'echte' agents van vlees en bloed weer in de beschrijving mee: vandaar dat ook kaartspelers en personen met stippen op het hoofd in de voorbeelden figureren, en voetballers.

Om te beginnen zijn er veel zaken van logische interesse waaraan we in dit hoofdstuk, en in dit boek, voorbijgaan. Zo is een 'echte' agent, of dit nu een processor of een individu is, beperkt in zijn redeneermogelijkheden: ook al weet hij dat p en dat $p \rightarrow q$, dat betekent nog niet dat hij ook q kan afleiden. Cognitieve architecturen zoals SOAR en ACT-R modelleren de beperkte redeneermogelijkheden van agents, en ook 'binnen' de logica wordt hier onderzoek naar verricht. Wij nemen daarentegen aan dat agents perfecte logici zijn: ze zijn op de hoogte van alle gevolgen van hun kennis. En we nemen zelfs aan, dat dit gemeenschappelijk bekend is bij die agents (gemeenschappelijke kennis is een vorm van 'en ze weten dit allemaal van elkaar', die pas in de loop van dit hoofdstuk duidelijk zal worden).

Een andere zaak van belang is dat kennis van agents zowel van voorlopige aard als feilbaar kan zijn. In het eerste geval redeneer je op basis van redelijke veronderstellingen die je bereid bent te herzien. De logica die dit bestudeert is de niet–monotone logica. Dit is een groeiend deelspecialisme waaraan we in dit boek voorbijgaan. In het tweede geval denk je iets te weten maar blijkt dit incorrect. Het zou dus preciezer zijn om te zeggen dat je het niet wist, maar *geloofde*. Het proces van herzien van incorrecte informatie heet kennisrevisie (in het Engels: belief revision). Ook hieraan gaan we in dit boek voorbij. Wat blijft nu nog over? De zogenaamde *ideale agent*.

Het volgende is een eenvoudig model voor de abstracte architectuur van een multi–agentsysteem dat bestaat uit ideale agents: we nemen aan dat iedere agent i een eindig aantal *lokale toestanden* kan aannemen, en voor de omgeving waarin het multi–agentsysteem gesitueerd is nemen we eveneens een eindig aantal omgevingstoestanden aan. De *globale toestand* van een systeem van n agents wordt dan beschreven door een rijtje $(l_1, l_2, ..., l_n, o)$. We spreken hier ook wel van een geïnterpreteerd of *distributief systeem*. Een redelijke aanname, die we ook in het volgende zullen zien terugkomen, is dat agents hun eigen toestand kennen. Dit betekent dat twee globale toestanden $(l_1, l_2, ..., l_n, o)$ en $(l'_1, l'_2, ..., l'_n, o')$ voor agent i niet van elkaar te onderscheiden zijn – en hier komt de toegankelijkheidsrelatie van de modale logica weer om de hoek kijken – als ze overeenkomen in de lokale toestand van die agent, dus als $l_i = l'_i$.

Zo'n globale toestand komt overeen met een *wereld* in de modale logica: van de bewering 'agent i heeft lokale toestand l_i in globale toestand l' maken we eenvoudigweg een atomaire propositie die in de wereld die correspondeert met l moet gelden. De kennis van de agents in die globale toestand komt overeen met een mogelijke–wereldenmodel dat uit dit soort werelden bestaat, met gebruikmaking van de hiervoor genoemde ononderscheidbaarheidsrelatie. En dat is het voornaamste onderwerp van dit hoofdstuk.

Los van de kennis die agents van elkaar hebben in een gegeven globale toestand, willen we ook de acties van agents beschrijven, bijvoorbeeld communicaties die onderdeel zijn van het uitvoeren van een protocol. Een actie van een agent in een multi–agentsysteem voert een globale toestand over in een nieuwe globale toestand. Op logisch niveau correspondeert zo'n actie met een relatie tussen mogelijke–werelden-modellen. De updates in de laatste paragraaf van dit hoofdstuk zijn voorbeelden van dergelijke acties.

19.3 KENNISLOGICA VOOR EEN AGENT

Kennislogica of epistemische logica is een toepassing van modale logica waarbij we aan 'werelden' denken als *feitelijke toestanden* van een persoon of processor, terwijl 'toegankelijkheid' zoiets wil zeggen als 'voorstelbaarheid' of 'ononderscheidbaarheid'. Hiermee krijgt $\Box\varphi$ de betekenis van 'ik weet dat φ', zodat voor de modale operator $\Box$ nu de letter K, van 'Know', wordt gebruikt. Voor $\Diamond$ is geen aparte letter algemeen gebruikelijk, maar de betekenis van $\Diamond\varphi$ is dus $\neg K\neg\varphi$. Dit staat voor 'ik weet niet dat φ niet zo is', met andere woorden: 'ik houd voor mogelijk dat φ' of 'ik kan me voorstellen dat φ'.

Er zijn twee mogelijke interpretaties voor de kennisoperator.

Interpretatie: toegang

Een eerste semantische interpretatie krijgt deze operator in termen van het computationele begrip 'toegang': de informatie waarover een gegevensbank of een processor in een zekere toestand van een reken-proces beschikt wordt bepaald door de verdere toestanden die in dat proces nog voor hem toegankelijk zijn.

Interpretatie: ononderscheidbaarheid

Een andere interpretatie legt de nadruk op het begrip 'ononderscheid-baarheid' of 'voorstelbaarheid'. 'Voorstelbaar' moeten we hier vrij nauw opvatten. Het betekent eerder 'voorstelbaar gegeven mijn beperkte waarneming en onvolledige kennis van de wereld om mij heen' dan 'in principe denkbeeldig in mijn fantasie'.

Voorbeeld 19.2

Ik kan me niet voorstellen dat mijn gebrilde gesprekspartner tegenover me geen bril op heeft: ik zie immers dat hij wel een bril draagt. Maar van de mij onbekende binnenkomer die achter mij met een luide roep "Hi, mate" het café binnenwandelt, kan ik me zowel voorstellen dat zij een bril draagt, als dat zij geen bril draagt.

De eigenschappen van kennis kunnen we bestuderen aan de hand van axioma's waar kennis 'idealiter' aan voldoet, en van de wijze waarop deze axioma's met frame–eigenschappen corresponderen. We kunnen vier van dergelijke eigenschappen onderscheiden:

Modale distributie

$$K(\varphi \to \psi) \to (K\varphi \to K\psi)$$

Thans zegt dit het volgende: 'de mij bekende gevolgen van mijn kennis weet ik ook'.

Waarheidsaxioma

$$K\varphi \to \varphi$$