Logica voor Informatici

Marc Denecker

5 september 2013

Bij de samenstelling van deze cursus werd gebruik gemaakt van verschillende andere cursussen: in de eerste plaats van de cursus van Prof. Jan Denef, die dit vak gaf tot in 2010, en ook van de cursus *CSC330 Logical Specifications* van Prof. Hector Levesque van de universiteit van Toronto.

De software LogicPalet die gebruikt wordt in deze cursus werd ontwikkeld door Jan Denef. Eén van de componenten van dit systeem is het tool Geo-Worlds, dat geïnspireerd is op het tool Tarski-Worlds dat hoort bij het boek van Barwise en Etchemendy: Language, Proof and Logic, CSLI publications (2002). LogicPalet biedt wel veel meer mogelijkheden dan Tarksi-Worlds. In LogicPalet wordt ook gebruik gemaakt van twee logische redeneersystemen: de theoremprover Spass van het Max Planck Institute for Computer Science, en de modelgenerator IDP die ontwikkeld wordt in de KRR-groep van Marc Denecker.

Aanbevolen literatuur

- J. Barwise, J. Etchemendy, *Language, Proof and Logic,* CSLI publications (2002)
- J. van Benthem, H. van Ditmarsch, J. Ketting, W. Meyer-Viol, *Logica voor informatici,* Addison Wesley Nederland (1994).

Inhoudsopgave

Hoofdstuk 1

Inleiding: van Nederlands naar logica

1.1 Logica voor informatica

Deze cursus gaat over logica en het gebruik ervan in de informatica.

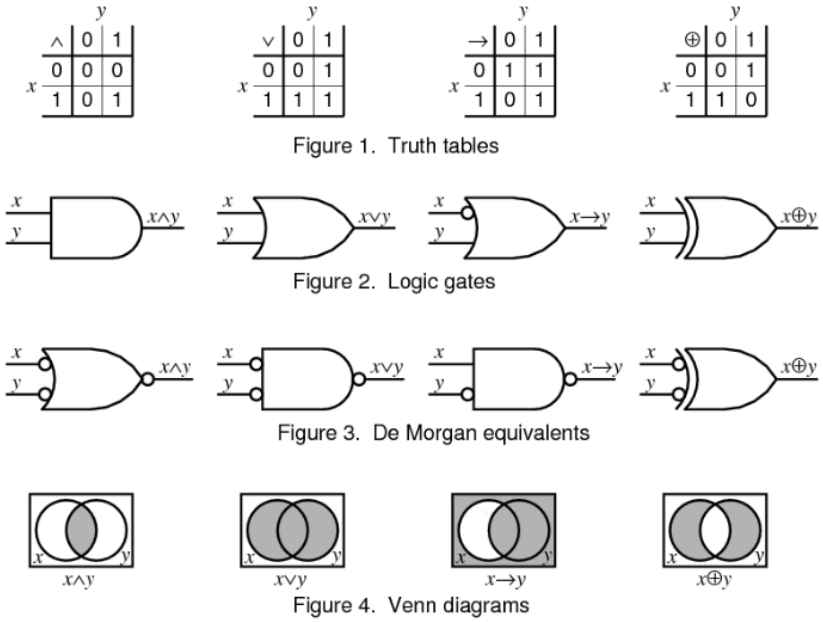
Het woord *logica* heeft verschillende betekenissen afhankelijk van de context waarin het gebruikt wordt. "Een logica" is een kunstmatige *formele taal*, t.t.z.[1] een taal waarvan de syntactische vorm en de betekenis van de uitdrukkingen op wiskundige manier vastgelegd wordt. In een andere context betekent logica ook *de kunst van het redeneren*.

Een wiskundige taal versus redeneerkunst: dat is wel erg verschillend! Het verband is echter: om te redeneren heb je *informatie* nodig, en om de wetten van correct redeneren te bestuderen heb je dus een wiskundige taal nodig om die informatie voor te stellen. Op die manier ging de studie van de redeneerkunst dus hand in hand met de ontwikkeling van formele talen (zie sectie 1.2 voor een kort overzicht van de geschiedenis). In deze cursus zullen we gebruik maken van twee van de belangrijkste en de oudste formele talen, namelijk propositielogica en predicatenlogica.

Logica omvat dus niet alleen de wetenschappelijke studie van het redeneren maar ook van "informatie". En informatie is waar alles om draait in de *Informatica*. Een centraal thema van de Informatica is: informatie, en hoe je met informatie in computersystemen omgaat. Informatie is een soort *grondstof* die mensen gebruiken om problemen op te lossen en taken uit te voeren. Informatici schrijven programma's om zo'n taken te laten uitvoeren door een computer. Om die programma's te kunnen schrijven moeten zij beschikken over voldoende basisinformatie. Bv. wie een programma schrijft om een uurrooster te berekenen voor de 1ste bachelor, moet het volgende weten: *(a) noch studenten, noch docenten kunnen twee hoorcolleges tergelijkertijd bijwonen*, *(b) in 1 auditorium kunnen geen twee hoorcolleges tergelijkertijd plaats vinden* , *(c) de capaciteit van het auditorium moet groter zijn dan het aantal studenten die het hoorcollege volgen*, enz. (a), (b) en (c) zijn drie stukjes informatie die in zo'n uurroosterprogramma verwerkt moeten worden. Hoe zo'n programmeur dat precies doet wordt overigens nog altijd niet goed begrepen.

Logica bestudeert dus informatie, een centraal onderwerp van de Informatica. Dat maakt logica van theoretisch belang voor Informatica. Maar logica en logische redeneersystemen worden ook steeds belangrijker in de praktijk via softwaretoepassingen. Hieronder volgt een (erg onvolledig)

[1] t.t.z. : afkorting van "t is te zeggen, dat wil zeggen".



Figuur 1.1: Booleaanse Algebra en circuits

overzicht:

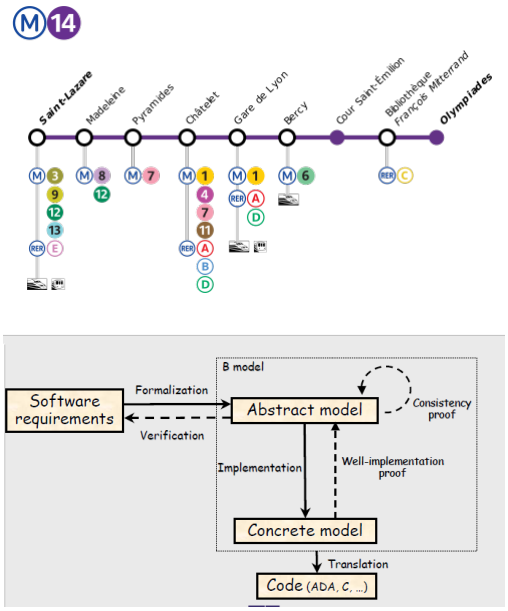
Booleaanse algebra is een andere benaming voor propositielogica en wordt intensief gebruikt in de electronica, om circuits en processoren te ontwerpen.

Relationele databanken zijn van de meest gebruikte informaticasystemen en kunnen beschouwd worden als logische redeneersystemen. Queries aan databanken zijn logische uitdrukkingen. Databanken implementeren een eenvoudige maar belangrijke vorm van redeneren. Het zijn tegenwoordig zonder meer de meest courante toepassingen van logica.

Formele modellering en specificatie. Een eerste stap in het ontwikkelen van software is het opstellen van *specificaties*. Verschillende aspecten kunnen gemodelleerd worden: belangrijke achtergrondkennis, concepten en informatie uit het probleemdomein, of gedetailleerde beschrijvingen van *wat* de software moet doen, en alle informatie die daarvoor van belang is. Meestal in natuurlijke taal, maar in toenemende mate worden deze specificaties in een logica geformuleerd waardoor redeneersystemen gebruikt kunnen worden om de eigenschappen van het systeem na te gaan, of om het te simuleren.

Automatische programmaverificatie. In dit domein worden de eigenschappen van de input en output van een programma beschreven door middel van logische formules: de eigenschappen van de input noemt men precondities, die van de output postcondities. Men ontwikkeld automatische bewijssystemen om de correctheid van programma's bewijzen: als de preconditie voldaan is voor oproep, dan zal de postconditie voldaan zijn na de oproep. Een voorbeeld hiervan is de taal spec# van Microsoft of het Verifast tool dat hier aan het departement Computerwetenschappen van de KUL ontwikkeld wordt.

(Semi-)automatische programmageneratie genereert software uit formele specificaties. Een voorbeeld is de software van de automatische metro-lijn 14 in Parijs die op deze manier door Siemens werd ontwikkeld (zie Figuur 1.2). Sinds zijn indienststelling in 1998 is nog geen enkele softwarefout vastgesteld!



- Parijse Metro 14: volautomatisch, in dienst sinds 1998
- Ontwikkeld door Siemens met formele methode: B-method
- Interactief proces: een serie van steeds concreter formele specificaties
- De correctheid van elke volgende specificatie werd automatisch bewezen ten opzichte van de vorige.
- Het (meest concrete) resultaat wordt automatisch omgezet naar een programma (Ada/C/Java).
- Arbeidsintensief: 110.000 lijnen in de B taal → 86.000 lijnen Ada.
- Er werd tot nog toe geen enkele softwarefout vastgesteld. De software Metro14.1.0 was nog steeds operationeel bij het verschijnen van deze paper: `http://deploy-eprints.ecs.soton.ac.uk/8/1/fm_sc_rs_v2.pdf`

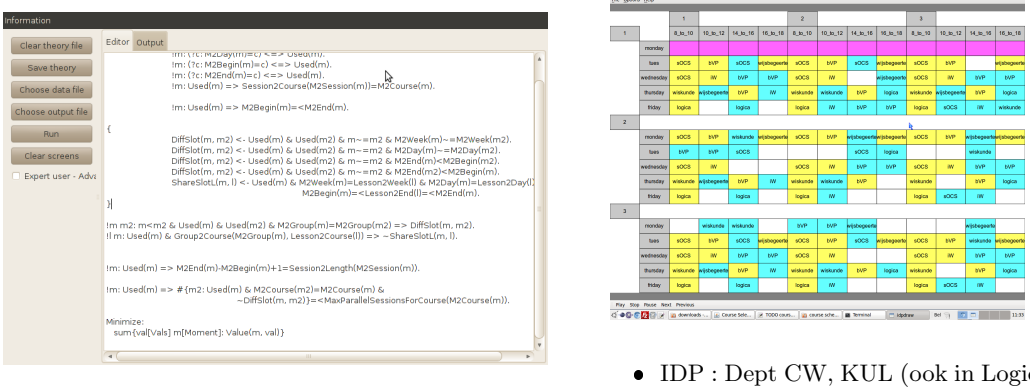
Figuur 1.2: Semi-automatische code-generatie

Het is vooralsnog een grote investering om deze logische methodes toe te passen. In de huidige stand van zaken is de toepassing ervan voorlopig beperkt tot *kritische* software-applicaties waar bugs een dramatische impact zouden kunnen hebben op mensen of bedrijven en dus ten alle prijze vermeden moeten worden (bv. het metro-systeem, software voor medische instrumenten, besturingssoftware voor raketten, besturingssystemen, enz). Maar de redeneersystemen worden krachtiger en het gebruik ervan eenvoudiger, en het gebruik neemt toe.

Declaratief programmeren (of problem solving). Een logisch redeneersysteem lost "het probleem" op enkel gebruik makend van de logische specificatie. Programma's schrijven hoeft niet meer. Dergelijke systemen worden ontwikkeld in het gebied van de kunstmatige intelligentie en de computationele logica. Er gebeurt veel onderzoek naar de ontwikkeling van formele talen en solvers ervoor, voor een snel toenemende scala van toepassingen: scheduling, planning, natuurlijke taal verwerking, diagnose, modellering van architectuur, bestuderen van privacy, toegangsrechten, enz. Vele toepassingen zitten nog in de onderzoeksfase. Een voorbeeld van logisch kennisbanksysteem dat diverse types van inferentie ondersteunt is het aan het departement Computerwetenschappen ontwikkelde systeem IDP dat wordt geïllustreerd in Figuur 1.3.

Microsoft is een voorbeeld van een bedrijf dat veel onderzoek doet in dit domein. Over declaratief programmeren gaat het in een recent interview van Bill Gates
`http://www.infoworld.com/d/developer-world/gates-talks-declarative-modeling-language-effort-386` (2011)

> *With the [Microsoft] declarative language project, the goal is to make programming declarative rather than procedural. "Most code that's written today is procedural code. And there's been this holy grail of development forever, which is that you shouldn't have to write so much [procedural] code," Gates said. "We're investing very heavily to say that customization of applications, the dream, the quest, we call it, should take a tenth as much code as it takes today."*



- Theorie Uurrooster.idp
- IDP berekent (optimale) structuur S die voldoet aan theorie Uurrooster.idp

- IDP : Dept CW, KUL (ook in Logic-Palet)
- Een generator van eindige modellen.
- Taal van IDP: een uitbreiding van predikatenlogica

Figuur 1.3: IDP voor (1) berekenen en (2) visualiseren van uurroosters

Conclusie Logica is van fundamenteel belang voor de Informatica, omdat het de wetenschap is van informatie, de grondstof van informatici, en van het gebruik van informatie om taken op te lossen.

De toepassing van logica en logische systemen is sterk in opmars. In databanksystemen worden logica en bepaalde vormen van redeneren dagdagelijks toegepast in de vorm van querytalen zoals SQL. Toch kun je de vraag stellen waarom logica en logische systemen niet meer aandacht krijgen in de informatica en waarom er niet méér toepassingen van logische systemen bestaan. Er zijn daar verschillende redenen voor:

- Ten eerste is het ontwikkelen van domeinonafhankelijke solvers die problemen kunnen oplossen door te redeneren over logische theorieën zeer complex, en de vooruitgang is traag. Voor veel toepassingen kunnen logische systemen nog niet toegepast worden, eenvoudigweg omdat er niet voldoende efficiënte tools zijn.

- Ten tweede heeft de wetenschappelijk wereld nog geen goed idee van welke soorten informatie relevant zijn in informatica-toepassingen, in welke formele taal deze informatie kan voorgesteld worden, en welke vormen van redeneren nuttig zouden zijn om het probleem op te lossen. Er is veel dat we nog niet begrijpen.

De belangrijkste doelstellingen van deze cursus zijn enerzijds, om te leren hoe gedachten/informatie op een klare en eenduidige manier uit te drukken, door gebruik te maken van logica, en anderzijds om een idee te krijgen van welke vormen van redeneren nuttig zijn in de context van informaticatoepassingen.

1.2 Korte geschiedenis van de logica

Er gebeurt zeer veel onderzoek naar logica, niet alleen in de Informatica maar ook in filosofie en in wiskunde. Dat is waar de oorsprong van logica ligt.

De Griekse wijsgeer *Aristoteles* (384-322 v.C.) was de eerste die de regels van het correct redeneren ontleedde. Hij heeft geen formele taal ontwikkeld maar is bekend omwille van zijn redeneerregels, *syllogismen* genaamd, zoals bv.

$$\frac{\begin{array}{c}\text{Alle grieken zijn sterfelijk}\\ \text{Socrates is een griek}\end{array}}{\text{Socrates is sterfelijk}}$$

Leibniz (Duits wiskundige en wijsgeer (1646-1716)) stelde voor de regels van het redeneren met wiskundige middelen te bestuderen. Hij was van oordeel dat complexe "gedachten" (waarmee ongeveer hetzelfde bedoeld wordt als "informatie" in de vorige sectie) konden samengesteld worden uit eenvoudiger gedachten met behulp een soort wiskundige operatoren gelijkaardig aan rekenkundige operatoren $+$, $\times$. Zelf heeft hij dat idee nooit volledig uitgewerkt.

Op het einde van de 19^{de} eeuw begonnen *Boole, Peano, Frege, Russell*, en anderen met de *wiskundige* studie van de redeneerregels (*symbolische* of *mathematische logica*) (voordien was de logica het domein van wijsgeren). Hun doel was de wiskunde op meer exacte wijze te funderen, om zo een aantal *paradoxen* weg te werken die te wijten waren aan een te slordige fundering (*Grondslagen* onderzoek in de wiskunde).

In 1879 publiceerde Frege het *"Begriffschrift"*, wat nu gekend is als de eerste versie van *predicatenlogica*. In predicatenlogica beschikt men over een zeer beperkte set van operatoren: $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, \forall, \exists$. Ze laten toe om complexe "gedachten" te construeren uit eenvoudiger "gedachten", zoals Leibniz in gedachten had. Met deze kleine verzameling van operatoren kunnen verbazingwekkend veel "gedachten" uitgedrukt worden, alhoewel dit niet altijd zo eenvoudig is, en hoewel er belangrijke uitzonderingen zijn.

Sinds *Gödel* (1930) is de "wiskundige logica" *meer* dan de studie der redeneerregels en grondslagenonderzoek (zie Hoofdstuk 5). De "wiskundige logica" heeft implicaties voor de filosofie, zuivere wiskunde, computerwetenschappen, kunstmatige intelligentie en de taalkunde.

Van meet af aan stond logica aan de wieg van de informatica. *Alan Turing*, één van de uitvinders van de computer, was een logicus. *John McCarthy*, vader van de artificiële intelligentie, stelde voor logica als basistaal van intelligente machines te gebruiken. *Edgar Codd* ontwikkelde de relationele databanken op basis van logica. *Hoare* gebruikte logica om over de correctheid van programma's te redeneren, enz. In vrijwel alle domeinen van de informatica speelt logica een basisrol, op zijn minst bij de ontwikkeling van de theoretische grondslagen ervan maar ook steeds meer bij concrete toepassingen, zoals we zonet gezien hebben.

Belangrijke toepassingen in de informatica zijn zoals gezegd computers zelf (gebaseerd op boolse logica), databanken om data op te slaan en op te vragen, en in het algemeen vele softwaretools die het mogelijk maken complexe informatie op te slaan en computers ermee te laten redeneren, problemen op te lossen, of de correctheid van programma's te bewijzen.

1.3 Notaties en basisbegrippen

In deze tekst maken we gebruik van de volgende standaard notaties :

- $\mathbb{N} = \{0, 1, 2, 3, \cdots\}$ (de natuurlijke getallen),
- $\mathbb{N}_0 = \mathbb{N} \backslash \{0\}$,

- $\mathbb{Z} = \{0, 1, -1, 2, -2, 3, -3, \cdots\}$ (de gehele getallen),
- $\mathbb{Q} = \{a/b \mid a, b \in \mathbb{Z}\,, b \neq 0\}$ (de rationale getallen),
- $\mathbb{R} =$ de verzameling der reële getallen.

Bovendien gebruiken we de volgende afkortingen :

- Geg. Gegeven
- T.B. Te Bewijzen
- Bew. Bewijs
- Q.E.D. Quod Erat Demonstrandum : einde bewijs
- V.T.B. Voldoende Te Bewijzen.
- asa als en slechts als
- t.t.z. 't is te zeggen
- m.a.w. met andere woorden
- i.p.v. in plaats van

We gebruiken ook een aantal griekse letters:

- θ theta
- Σ Sigma
- α alpha

Verzamelingen. We gebruiken een aantal basisconcepten uit verzamelingentheorie.

- We schrijven $A \subset B$ om aan te duiden dat A een deelverzameling is van B; dit sluit niet uit dat $A = B$.
- Een *n-tal* van een verzameling D is een rij $(d_1, \ldots, d_n)$ van lengte n, bestaande uit elementen van D. Een koppel is een 2-tal.
- Een n-voudige *relatie* R over een verzameling D is een verzameling van n-tallen van D. We noemen D ook wel het *domein* van R.
- Het n-voudig *cartesisch product* van D is de verzameling van alle n-tallen van D. Deze verzameling wordt genoteerd als D^n.
- Een n-voudige relatie met domein D is dus een deelverzameling van D^n.

- Een *afbeelding* of *functie* f van verzameling D_1 naar verzameling D_2 is een verzameling van 2-tallen zodat voor elke $a \in D_1$ er precies één $b \in D_2$ bestaat zodat $(a, b) \in f$. We noemen D_1 het domein en D_2 het bereik van f.

  We gebruiken de volgende notatie om een functie te noteren. Bv.

  $$F : \mathbb{N} \to \mathbb{N} : n \to n^2$$

  is de kwadraat functie in de natuurlijke getallen, en bestaat dus uit alle koppels van de vorm (n, n^2). Of

  $$G : \mathbb{N}^2 \to \mathbb{N} : (n, m) \to n^2 + m^2$$

  is de functie die koppels (n, m) afbeeldt op $n^2 + m^2$.

- Een *bijectie* f van verzameling D_1 naar verzameling D_2 is een afbeelding zodat voor elk object $b \in D_2$ een uniek object $d \in D_1$ bestaat zodat $f(d) = b$.

Inductieve of recursieve definities. In logica worden veel concepten op een recursieve manier gedefinieerd. Een bekend voorbeeld van een recursief gedefinieerd concept is dat van de fibonacci getallen. Voor elke $n \in \mathbb{N}$ definiëren we $fib(n)$ recursief (of inductief):

- $fib(0) = 0$
- $fib(1) = 1$
- $fib(n + 2) = fib(n) + fib(n + 1)$

Dus bestaat die rij uit de getallen $0, 1, 1, 2, 3, 5, 8, \ldots$. Ze wordt bekomen door eerst de basisgevallen uit te rekenen (voor 0 en 1) en vervolgens iteratief de recursieve regel toe te passen om $fib(n)$ te berekenen voor steeds grotere n.

Een dergelijk type van definitie wordt een *inductieve* or *recursieve* definitie genoemd. Deze hier bestaat uit twee basisregels en één inductieve of recursieve regel.

1.4 Waarom predicatenlogica?

Er worden veel logica's en formele talen gebruikt in de Informatica, en er worden er steeds nieuwe geïntroduceerd. Waarom kiezen we dan voor zo'n 130 jaar oude taal als predikatenlogica? De reden is enerzijds omdat het een eenvoudige en toch zeer expressieve, domeinonafhankelijke taal is. Anderzijds is predikatenlogica een soort basistaal in de zin dat de concepten ervan op één of andere manier terugkeren in vrijwel alle andere logica's of formele talen.

Predikatenlogica, in de vorm zoals hier voorgesteld wordt, wordt niet vaak gebruikt in softwaresystemen. Deze logica is immers te weinig expressief om bepaalde types van informatie op een natuurlijke manier in uit te drukken. In sommige systemen worden daarom meer expressieve talen gebruikt. Bv. predikatenlogica wordt niet letterlijk gebruikt in databanksystemen. Daar gebruikt men SQL. Maar toch is het zo dat de principes van SQL gebaseerd zijn op predikatenlogica, en dus, als je predikatenlogica beheerst dan begrijp je al heel veel van SQL.

In deze cursus zullen we verwijzen naar veel verschillende op logica gebaseerde softwaresystemen: databanksystemen, verificatiesystemen, constraintsystemen, modelgeneratoren, enz. Vrijwel alle

systemen gebruiken hun eigen logica. Het is ondoenbaar om al die talen te introduceren in deze cursus, of zelfs in de volledige opleiding Informatica. Maar die logica's zijn allemaal gebaseerd op, of overlappen met predicatenlogica, en daarom kunnen we in deze cursus de werking van die systemen toch uitleggen, door gebruik te maken van predikatenlogica, en door te verwijzen naar vormen van redeneren in de context van predikatenlogica.

1.5 Introductie tot predicatenlogica

Normaal gezien drukken we informatie uit in natuurlijke taal (het Nederlands bv.). Maar natuurlijke taal is vaak vaag, onvolledig of zelfs dubbelzinnig. Eén van de belangrijkste doelstellingen van deze cursus is om gedachten/informatie op een klare en eenduidige manier te leren uitdrukken, door gebruik te maken van logica.

Om logica te introduceren maken we gebruik van een *relationele databank*. Dat is een systeem om informatie in op te slaan en op te vragen. Databanksystemen zoals MS Access of Oracle laten je toe om een relationele databank te bouwen.

In een databank wordt de informatie opgeslagen in tabellen. Een tabel met n kolommen is eigenlijk een n-voudige relatie, t.t.z. een verzameling van n-tallen. Elk n-tal komt overeen met een rij in de tabel. Bv. de eerste rij van de tabel *Instructor* in Figuur 1.4 is het koppel $(Ray, CS230)$.

Met een databank kunnen we ook een *domein* D associeren, dit is de verzameling van al de items die voorkomen in de tabellen. Een tabel komt dus overeen met een relatie over het domein D.

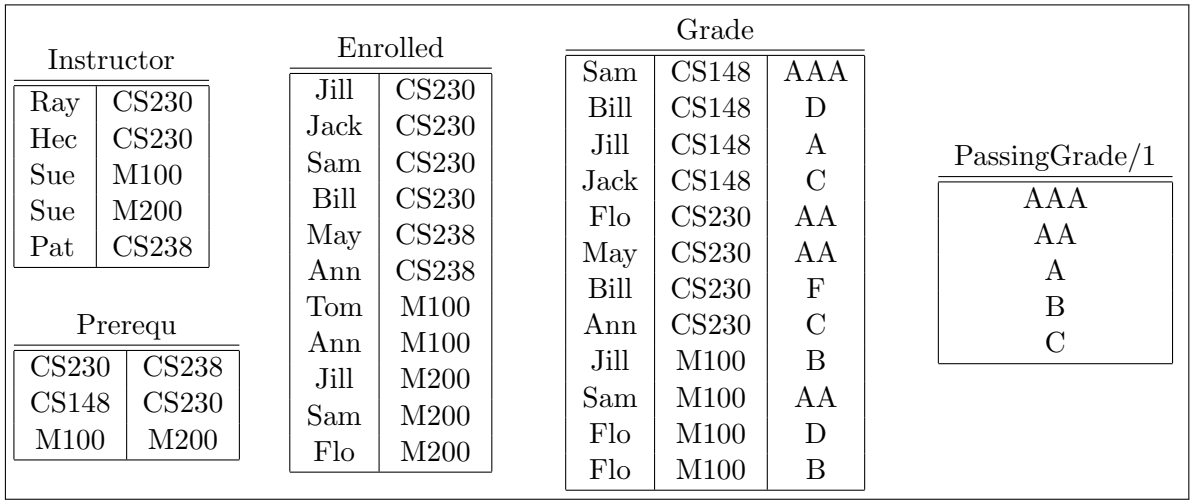
De combinatie van een domein en een aantal relaties tussen elementen uit dat domein vormen een *structuur*. We zullen zien dat dit concept een sleutelconcept is in de logica. Merk op dat normaal gezien de gegevens in een databank evolueren in de loop van de tijd, bv. doordat nieuwe rijen toegevoegd of verwijderd worden. Maar een databank op één specifiek tijdstip komt dus overeen met een *structuur*.

Tenslotte, de informatie die in een databank zit is van een eenvoudige soort, met name *data*. Data zijn een zeer concrete en logisch eenvoudige vorm van informatie. Bv. dat Jack geregistreerd is voor de cursus CS230, of dat Ray docent is van CS230 zijn data. Informatie van complexere logische aard, zoals *"geen enkele cursus is een prerequisite voor zichzelf"* of *"elke student heeft hoogstens 1 score voor een vak"*, kan niet voorgesteld worden in tabelvorm in een databank. Het zijn samengestelde uitspraken, opgebouwd met de logische operatoren zoals $\wedge, \Rightarrow, \forall$, etc..

We nemen als voorbeeld een studentendatabank met de volgende relaties (of tabellen - vanaf nu gebruiken we deze woorden als synoniemen):

- Instructor: lesgever ... is docent voor vak ...
- Enrolled: student ... is ingeschreven voor vak ...
- Prerequ: vak ... is (directe) nodige voorkennis voor vak ...
- Grade: student ... behaalde voor vak ... de score ...
- PassingGrade: score ... is een slaag-score ($\geq C$)

We veronderstellen dat de studentendatabank de betreffende relaties beschrijft op een volledige en correcte manier. Dan kunnen we de databank gebruiken voor het oplossen van een aantal vragen. Hieronder tonen we een lijst van vragen. Elke vraag komt overeen met een uitspraak waarvan we de waarheid in de databank na moeten gaan. We geven telkens aan hoe deze uitspraak in predicatenlogica voor te stellen.



Instructor

Ray	CS230
Hec	CS230
Sue	M100
Sue	M200
Pat	CS238

Prerequ

CS230	CS238
CS148	CS230
M100	M200

Enrolled

Jill	CS230
Jack	CS230
Sam	CS230
Bill	CS230
May	CS238
Ann	CS238
Tom	M100
Ann	M100
Jill	M200
Sam	M200
Flo	M200

Grade

Sam	CS148	AAA
Bill	CS148	D
Jill	CS148	A
Jack	CS148	C
Flo	CS230	AA
May	CS230	AA
Bill	CS230	F
Ann	CS230	C
Jill	M100	B
Sam	M100	AA
Flo	M100	D
Flo	M100	B

PassingGrade/1

AAA
AA
A
B
C

Figuur 1.4: De studentendatabank

- *Behaalde Sam ooit een AAA in CS148?.*
  Vertaling:
  $$Grade(Sam, CS148, AAA)$$
  Antwoord: waar.

- *Is Sue docent van zowel M100 als M200 ?.*
  $$Instructor(Sue, M100) \wedge Instructor(Sue, M200)$$
  Antwoord: waar.

  $\wedge$ betekent "en"

- *Is CS148 of M100 (directe) nodige voorkennis voor CS230 ?'*
  $$Prerequ(CS148, CS230) \vee Prerequ(M100, CS230)$$
  Antwoord: waar.

  $\vee$ betekent "of"

- *Is precies één van CS148 of M100 (directe) nodige voorkennis voor CS230 ?'*
  $$(Prerequ(CS148, CS230) \vee Prerequ(M100, CS230)) \wedge \neg(Prerequ(CS148, CS230) \wedge Prerequ(M100, CS230))$$
  Antwoord: waar.

  $\neg$ betekent "het is niet het geval dat"

Wat denk je hiervan?

$$(Prerequ(CS148, CS230) \wedge \neg Prerequ(M100, CS230)) \vee (\neg Prerequ(CS148, CS230) \wedge Prerequ(M100, CS230))$$

Of hiervan:

$$(Prerequ(CS148, CS230) \Leftrightarrow \neg Prerequ(M100, CS230))$$

$\boxed{\Leftrightarrow \text{ betekent “als en slechts als”}}$

Het lijkt alsof deze drie formules dezelfde eigenschap uitdrukken en *equivalent* zijn!

- *Vereist CS238 nodige voorkennis?*

  $$(\exists x) Prerequ(x, CS238)$$

  Antwoord: waar.

  $\boxed{(\exists x)\ldots \text{ betekent “er bestaat een object “x” zodat ...”}}$

  Hier is x een nieuw type symbool, namelijk een *logische variabele*. Die dient als een soort plaatshouder om een object van onbekende identiteit aan te duiden.

- *Is May geslaagd voor CS230 ?*

  $$(\exists x)(Grade(May, CS230, x) \wedge PassingGrade(x))$$

  Antwoord: waar.

- *Heeft Ray minstens één student?*

  $$(\exists x)(Instructor(Ray, x) \wedge (\exists y) Enrolled(y, x))$$

  Antwoord: waar.

- *Heeft Flo al een examen van een vak afgelegd?*

  $$(\exists x)(\exists y) Grade(Flo, x, y)$$

  Antwoord: waar.

- *Doceert Ray al de vakken die (directe) nodige voorkennis zijn voor CS238 ?*

  $$(\forall x)(Prerequ(x, CS238) \Rightarrow Instructor(Ray, x))$$

  Antwoord: waar.

  $\boxed{\ldots \Rightarrow \ldots \text{ betekent “Indien } \ldots \text{ dan } \ldots \text{”} \text{ of ook “}\ldots \text{ impliceert } \ldots \text{”}}$

  $\boxed{(\forall x)\ldots \text{ betekent “Voor elk object “x” geldt ...”}}$

  Hierbij is x ook weer een logische variabele.

- *Doceert Ray al de vakken die (directe) nodige voorkennis zijn voor M100 (in plaats van CS238)?*

$$(\forall x)(Prerequ(x, M100) \Rightarrow Instructor(Ray, x))$$

Vreemde vraag! $M100$ heeft geen prerequisites!
Antwoord: triviaal waar. Hij geeft elk van die 0 cursussen.

- *Is er iemand die ingeschreven is voor CS230 en die steeds geslaagd is voor alle vakken die hij/zij tot hiertoe aflegde?*

$$(\exists x)(Enrolled(x, CS230) \wedge (\forall y)(\forall z)[Grade(x, y, z) \Rightarrow PassingGrade(z)])$$

Antwoord: waar.

- *Is iedereen die ingeschreven is voor CS230, geslaagd voor minstens één vak gedoceerd door Sue?*

$$(\forall x)(Enrolled(x, CS230) \Rightarrow (\exists w)(\quad Instructor(Sue, w) \wedge \\ (\exists s)(Grade(x, w, s) \wedge PassingGrade(s))))$$

Antwoord: onwaar!

- *Heeft John examen voor CS230 afgelegd?*

$$(\exists s)Grade(John, CS230, s)$$

Merk op: John behoort niet tot domein van de databank. Een Databanksysteem zal antwoorden: onwaar. Maar in logica zal de waarheid van deze zin onbepaald zijn.

- *Zijn er studenten die bij elke docent les volgen?*

$$(\exists x)(\forall y)((\exists w)Instructor(y, w) \Rightarrow (\exists w)(Instructor(y, w) \wedge Enrolled(x, w)))$$

Antwoord: onwaar!

Merk op dat deze formule twee deelformules bevat van de vorm $(\exists w)\ldots$. De w's in verschillende deelformules zijn volledig onafhankelijk. We kunnen dus evengoed één van de w's vervangen door een andere variabele, zoals in:

$$(\exists x)(\forall y)((\exists w)Instructor(y, w) \Rightarrow (\exists z)(Instructor(y, z) \wedge Enrolled(x, z))$$

Wat is er fout met

$$(\exists x)(\forall y)(\exists w)(Instructor(y, w) \wedge Enrolled(x, w)))?$$

Deze zin zegt dat er een x bestaat zodat voor elk object y uit de universum - studenten, scores, vakken, docenten – een object w bestaat zodat y docent is van w en x ingeschreven is voor w. Dat is evident verkeerd; het zal NOOIT voldaan zijn omdat een score of een vak geen instructor zijn.

De rol van $(\exists w)Instructor(y, w)$ in de implicatie is de universele kwantor te beperken tot docenten. De oorspronkelijke zin is equivalent aan de volgende:

$$(\exists x)(\forall y)(\quad \underbrace{\neg(\exists w)Instructor(y, w)} \vee \\ \underbrace{(\exists w)(Instructor(y, w) \wedge Enrolled(x, w)))})$$

De eerste subformule is voldaan voor alle y's die geen docent zijn. De tweede is voldaan voor docenten van x.

- *Is er een student die precies één vak volgt?*

$$(\exists x)(\exists w)(Enrolled(x,w) \wedge (\forall u)(Enrolled(x,u) \Rightarrow u = w))$$

  Antwoord: waar.

$\ldots = \ldots$ betekent "$\ldots$ is gelijk aan $\ldots$"

We zien dat elke vraag overeenkomt met een logische uitdrukking, een logische *zin*. De vraag die de databank moet oplossen is:

nagaan of een logische zin waar is in een structuur.

Dit is een eenvoudige maar belangrijke vorm van redeneren over logische zinnen.

n-voudige queries. Een meer algemene vraag die we aan een databank willen stellen is: geef alle objecten of n-tallen die aan een voorwaarde voldoen. Bv. geef alle studenten die minstens voor 1 vak een AAA behaalden. Of geef alle koppels van studenten en vakken zodat de student gebuisd is voor het vak. Het antwoord op zo'n vraag is niet *waar* of *onwaar*, maar is een tabel, t.t.z. een relatie, dus een verzameling van n-tallen van objecten uit het domein van de databank.

In de terminologie van databanken noemt men dit een *query* (meervoud *queries*). Een gebruiker die een query wil stellen zal een *query-voorschrift* aan de databank opgeven. Dit query-voorschrift is een symbolische logische beschrijving van de verzameling n-tallen die hij of zij zoekt. Daarom zullen we een query hier voorstellen als een expressie van de vorm:

$$\{(x_1, \ldots, x_n) : A\}$$

Hierbij dienen variabelen $x_1, \ldots, x_n$ als plaatshouders voor verschillende objecten in een gezocht n-tal (of in een rij van de gezochte tabel) en A is een logische uitdrukking die de voorwaarde aangeeft op deze objecten. Dit soort expressie lijkt sterk op de manier waarop in wiskunde verzamelingen omschreven worden, zoals bv.

$$\{n \in \mathbb{N} \mid n \text{ is een priemgetal}\}$$

en dat is geen toeval want een query-voorschrift is inderdaad een symbolische omschrijving van een verzameling die vervolgens berekend moet worden door het databanksysteem.

Een vraag van het eenvoudige soort met een logische zin kan beschouwd worden als een 0-voudige query.

Een databanksysteem zal een query oplossen door de corresponderende relatie in de databank te berekenen, t.t.z. de verzameling van n-tallen $(a_1, \ldots, a_n)$ uit het domein van de databank die voldoen aan de voorwaarde A. Dit is een tweede vorm van logisch redeneren.

Opmerking 1.5.1. Zo'n query-voorschrift bevat een formule van predicatenlogica (namelijk A) maar is zelf geen uitdrukking van predicatenlogica.

Hieronder volgen een aantal voorbeelden.

- *Gevraagd: alle koppels (x,y) zodat student x geslaagd is voor vak y.*

$$\{(x,y) : (\exists z)(Grade(x,y,z) \wedge PassingGrade(z))\}$$

  Een databanksysteem zal deze vraag beantwoorden met de volgende relatie:

Sam	CS148
Jill	CS148
Jack	CS148
Flo	CS230
May	CS230
Ann	CS230
Jill	M100
Sam	M100
Flo	M100

- *Gevraagd: alle studenten die precies één vak volgen.*

$$\{x : (\exists w)(Enrolled(x, w) \wedge (\forall u)(Enrolled(x, u) \Rightarrow u = w))\}$$

Opmerking 1.5.2. We schrijven de haakjes niet voor een 1-tal. M.a.w., we schrijven $\{x : \dots\}$ in plaats van $\{(x) : \dots\}$.

- *Gevraagd: alle excellente studenten. Daarmee bedoelen we studenten die minstens één examen afgelegd hebben en op alle afgelegde examens een AA of AAA behaalden.*

$$\{x : (\exists y)(\exists z)Grade(x, y, z) \wedge (\forall y)(\forall z)(Grade(x, y, z) \Rightarrow z = AA \vee z = AAA)\}$$

- *Gevraagd: alle slaag-scores die reeds door minstens één student behaald zijn.*

$$\{x : PassingGrade(x) \wedge (\exists y)(\exists z)Grade(y, z, x)\}$$

- *Gevraagd: alle studenten die ingeschreven zijn voor CS230 en die steeds geslaagd zijn voor alle vakken die ze tot hiertoe aflegden.*

$$\{x : Enrolled(x, CS230) \wedge (\forall y)(\forall z)(Grade(x, y, z) \Rightarrow PassingGrade(z))\}$$

We stellen hier een tweede vorm van redeneren vast:

bereken de relatie voorgesteld door een query-voorschrift in een structuur.

Databanksystemen kunnen zoiets efficiënt, zelfs voor enorme databanken.

Logica in Nederlands vertalen. Alle symbolen van predicatenlogica zijn nu geïntroduceerd en hun betekenis uitgelegd. In principe moet je nu in staat zijn om een logische uitdrukking te vertalen naar het Nederlands. Neem als voorbeeld:

$$(\exists x)(\exists y)Enrolled(x, CS230) \wedge Enrolled(y, CS230) \wedge \neg x = y \wedge \\ (\forall z)(Enrolled(z, CS230) \Rightarrow z = x \vee z = y)$$

Letterlijk vertaald geeft dit:

er bestaat een x en y zodat x en y zijn ingeschreven in de cursus CS230 en x is verschillend van y en voor elke z geldt dat als z ingeschreven voor CS230, dan is z gelijk aan x of y.

Vertaald in meer courant Nederlands betekent dit dat er exact twee studenten CS230 volgen (*onwaar* in de gegeven studentendatabank). De grootste moeilijkheid van logica is dat je in staat moet zijn dit soort vertalingen heen en weer te doen. Sommigen vinden dit gemakkelijk, anderen niet, maar je kan het wel leren door te oefenen.

Integriteitsbeperkingen. Logische formules duiken op in databanken als de taal om queries te stellen. Ze zijn ook op een andere manier van belang.

Een databank evolueert in de tijd maar kan meestal niet zomaar elke willekeurige vorm aannemen. In een zinvolle databank zullen bepaalde eigenschappen steeds voldaan blijven. Zulke eigenschappen noemt men *integriteitsbeperkingen*. Wanneer dit niet het geval is kan men bizarre antwoorden krijgen op queries. Of computerprogramma's die de databank gebruiken zullen dan vaak in de fout gaan.

Om een databanksysteem toe te laten te controleren of de integriteitsbeperkingen voldaan zijn en blijven bij elke aanpassing van de databank, moeten ze geformuleerd worden in een formele taal, bijvoorbeeld predicatenlogica.

In het geval van de studentengegevensbank:

- *Geen enkel vak is (directe) nodige voorkennis voor zichzelf.*

  $$\neg(\exists w)Prerequ(w, w)$$

  Voldaan.

- *Niemand kan meer dan één score hebben voor hetzelfde vak.*

  $$\neg(\exists x)(\exists w)(\exists s)(\exists t)(Grade(x, w, s) \wedge Grade(x, w, t) \wedge s \neq t)$$

  Opgepast: deze beperking is niet voldaan voor onze databank. Dat betekent dat de databankmanager de integriteit van de databank zal moeten herstellen, bv. door 1 van de twee scores van Flo voor M100 te vernietigen. Hij past daarbij een nieuwe vorm van redeneren toe:

pas een structuur aan zodat ze aan een logische zin A voldoet.

  Het spreekt vanzelf dat men hier kleine aanpassingen wil! Er zijn ook meerdere oplossingen, wat menselijke tussenkomst nodig maakt.

- Een aantal integriteitsbeperkingen hebben te maken met de *types* van de relaties. Bv. *Geen enkele student kan een docent zijn.*

  $$\neg(\exists x)((\exists w)Instructor(x, w) \wedge (\exists w)Enrolled(x, w))$$

  Voldaan?

  In deze zin drukt $(\exists w)Instructor(x, w)$ uit dat x een docent is en $(\exists w)Enrolled(x, w))$ dat x een student is. Merk op dat de w in beide ongerelateerd is.

  We hebben soortgelijke integriteitsbeperkingen om aan te geven dat vakken, studenten, docenten en scores paarsgewijs disjunct zijn, dus geen gemeenschappelijk elementen bevatten.

- *Iedereen die ingeschreven is voor een vak moet geslaagd zijn voor alle vakken die (directe) nodige voorkennis zijn voor dat vak.*

  $$(\forall x)(\forall w)(Enrolled(x, w) \Rightarrow (\forall t)(Prerequ(t, w) \Rightarrow (\exists s)(Grade(x, t, s) \wedge PassingGrade(s))))$$

  Voldaan?

Merk op dat het nagaan van een integriteitsbeperking ook weer neerkomt op het nagaan of deze *waar* is in de databank. Er zit echter een addertje onder het gras. Bij elke aanpassing van de databank moet efficiënt nagegaan moeten worden of de integriteitsbeperking waar blijft. Mogelijks is dit zeer kostelijk. Slimme metodes zullen dit probleem *incrementeel* oplossen door gebruik van het feit dat de integriteitsbeperkingen voordien voldaan waren en dat er telkens niet veel aan de databank verandert.

Een goedkopere methode is enkel transacties toe te laten waarvan we kunnen garanderen dat ze de integriteitsbeperkingen zullen bewaren. Maar dat vereist verificatie van de transacties. We zullen dit in Hoofdstuk 4 bespreken.

Tenslotte komen we terug op de types. In sommige queries introduceerden we typeringen voor variabelen (bv. $(\exists y)Instructor(x, y)$ om uit te drukken dat x een docent is), en in andere niet. Wanneer moet het en wanneer moet het niet? Men kan zelfs de vraag stellen of het wel een goede manier is om docenten te karakteriseren als diegene die een cursus geven. Immers, het is mogelijk dat een docent tijdelijk geen les geeft. Klassieke logica is niet getypeerd en dat heeft zijn nadelen. We komen hierop terug in Hoofdstuk 4.

Correcte vertalingen. Een andere vraag is wanneer een logische query (een zin of een verzamelingenuitdrukking) een *correcte vertaling* is van een Nederlandse query. Het volstaat natuurlijk niet dat het antwoord juist is in de huidige databank. Immers, dan zou $Enrolled(Flo, M200)$ een correcte vertaling zijn van de query of Ray instructor is van CS230 (want beide zijn *waar*).

Het antwoord is: *een logische query is een correcte vertaling wanneer het antwoord op de logische query in* **elke zinvolle databank** *correct is.* De zinvolle databanken zijn diegene die voldoen aan de integriteitsbeperkingen.

Neem nu de volgende query: *geef alle studenten die geslaagd zijn voor alle vakken waaraan ze deelgenomen hebben aan de examens.* We zien nu al dat er een probleem is met deze query, namelijk voor Flo die zowel gebuisd als geslaagd lijkt voor M100. Vergelijk de volgende twee logische queries.

$$\{x : (\forall c)(\forall g)(Grade(x, c, g) \Rightarrow PassingGrade(g))\}$$

Deze query geeft alle studenten met enkel geslaagde scores. Daar behoort Flo niet toe wegens haar D score.

$$\{x : (\forall c)((\exists g)Grade(x, c, g) \Rightarrow (\exists g)(Grade(x, c, g) \wedge PassingGrade(g)))\}$$

Deze query geeft de studenten x die voor elk vak c dat ze afgelegd hebben (t.t.z. $(\exists g)Grade(x, c, g)$) een slaag-score hebben behaald (namelijk $(\exists g)(Grade(x, c, g) \wedge PassingGrade(g))$). Flo voldoet aan deze query want ze heeft slaagscores AA en B voor de vakken die ze heeft afgelegd.

Beide logische queries kunnen als *correct* beschouwd worden. Ze hebben hetzelfde antwoord in elke *zinvolle* databank die voldoet aan de integriteitsbeperking dat elke student hoogstens 1 score voor een vak heeft.

We zullen twee queries of twee logische formules *equivalent* noemen als ze dezelfde antwoorden hebben in elke structuur. Bovenstaande queries zijn dus beide correct maar niet *equivalent*, want er bestaat dus een databank (zij het een *zinloze*) waarin de antwoorden verschillend zijn. De huidige studentengegevensbank is een voorbeeld.

Oefening 1.5.1. *Test jezelf!!! Los al de vorige voorbeelden zelf op met behulp van het software packet LogicPalet. Voor informatie over dit tool verwijs ik naar Toledo.*

Met de formule-editor van LogicPalet kun je formules opstellen. Je kunt het antwoord berekenen op een query in een structuur en je kunt nakijken of je oplossing equivalent *is met de antwoorden in de cursus door gebruik te maken van AskSPass. Meer details over deze oefening vind je op de Toledo-webpagina van de cursus, onder Documenten/Huiswerken 1.5.1.*

Merk op: LogicPalet berekent of je oplossing equivalent is met die van de cursus, niet of ze correct *is. De opgaven zijn zodanig opgesteld dat een* correcte *logische formulering meestal wel logisch equivalent zal zijn, tenzij jouw formulering vergezocht is.*

1.6 Samenvatting

In dit hoofdstuk hebben de rol van logica in de informatica besproken, namelijk als de wetenschap van informatie en hoe met informatie te rederenen om problemen op te lossen.

Het belangrijkste doel van de cursus is om studenten aan te leren om "gedachten", "informatie", "eigenschappen", of hoe je het ook wilt noemen, uit te drukken in logica. Als aanzet hiertoe hebben we de betekenis van de logische operatoren van predikatenlogica geïntroduceerd.

Een ander doel van de cursus is om te laten zien welke vormen van redeneren er bestaan en hoe ze van pas komen in informaticatoepassingen. In de context van databanken hebben we

- nagaan of een logische zin *waar* is in een structuur
- de relatie berekenen van een relatie-voorschrift in een structuur (een veralgemening van het vorige)

Dit zijn de belangrijkste functionaliteiten van een databanksystemen. Verder zagen we ook:

- een structuur aanpassen zodat een formule A voldaan is.

We komen later nog terug op databanksystemen.

Hoofdstuk 2

De predikatenlogica

In het inleidende hoofdstuk hebben we kennis gemaakt met de vorm en betekenis van logische uitdrukkingen en met structuren (de databank). We hebben gemerkt dat logische uitdrukkingen waar of onwaar kunnen zijn in een structuur. In dit hoofdstuk veralgemenen we deze bevindingen.

2.1 Syntax van predicatenlogica

Een logische formule is een string, een reeks van tekens gevormd volgens strikte regels. Zo'n string is samengesteld uit *symbolen* die zelf vaak gevormd zijn uit een reeks van tekens. Er zijn de volgende types van symbolen:

- **Hulptekens**: haakjes: ( ) en komma: ,
- **Logische symbolen**: deze zijn universeel van aard.
  - Logische connectieven:
    - de conjunctie $\wedge$, betekent "en"
    - de disjunctie $\vee$, betekent "of"
    - de negatie $\neg$, betekent "niet"
    - de implicatie $\Rightarrow$, betekent "als ... dan ..."
    - de equivalentie $\Leftrightarrow$, betekent "als en slechts als", afgekort "asa".
  - De kwantoren
    - de existiële kwantor $\exists$, betekent "er bestaat"
    - de universele kwantor $\forall$, betekent "voor alle".
  - Het gelijkheidssymbool $=$
- **Niet-logische symbolen**: deze zijn domeinspecifiek. Er zijn drie soorten:
  - Constantensymbolen (ook kortweg constanten genoemd)
    - Voorbeelden uit de studentendatabank: $Ray, Hec, CS230, \ldots$.
  - Predikaatsymbolen, ook relatiesymbolen genoemd. Elk predikaatsymbool heeft een vast aantal argumenten. Dit aantal wordt de *ariteit* van het predikaatsymbool genoemd. Een n-voudig predikaatsymbool P wordt vaak P/n genoteerd.
    - Vaak gebruikte predikaatsymbolen : P, Q, R
    - Voorbeelden uit de studentendatabank: $Grade/3, Enrolled/2, PassingGrade/1$.

- Functiesymbolen, ook functoren genoemd, elk met een ariteit, net zoals voor predikaatsymbolen. Net als predikaten schrijven we F/n : om aan te geven dat F een n-voudig functiesymbool is.
  - Vaak gebruikte functiesymbolen : F, G
  - Om het onderscheid te maken tussen predikaatsymbolen en functiesymbolen zullen we schrijven bv. $Plus/2:, Times/2:$. De ":" na de ariteit geeft aan dat het om een functiesymbool gaat.
  - Functiesymbolen komen niet voor in de studentendatabank.

Predikaatsymbool versus functiesymbool. Een predikaatsymbool dient om *feiten* aan te duiden, zoals $Enrolled(Jack, CS230)$, terwijl een functiesymbool dient om *objecten* aan te duiden zoals $Plus(2, 3)$ dat het getal 5 zou kunnen aanduiden, of $Leeftijd(Ray)$, dat de leeftijd van Ray zou kunnen aangeven. Feiten kunnen waar of onwaar zijn, objecten niet.

Een **constante** wordt vaak gezien als een functiesymbool van ariteit 0, t.t.z. zonder argumenten. Wanneer we schrijven $c/0:$ dan bedoelen we dat c een constante is.

Notatie 2.1.1. Predikaat- en functiesymbolen beginnen met een hoofdletter en constanten kunnen met hoofd- of kleine letter beginnen.

Vocabularium. Wanneer we logica willen gebruiken in een toepassing (bv. de studentengegevensbank), is de eerste stap het kiezen van een aantal niet-logische symbolen om te kunnen spreken over bepaalde relaties, functies en objecten uit het toepassingsdomein. Deze symbolen vormen het vocabularium waarin we vervolgens eigenschappen kunnen uitdrukken. In de studentendatabank bestond het vocabularium van de queries gewoon uit de namen van tabellen en objecten uit de databank.

Een **vocabularium** is een verzameling van niet-logische symbolen. Een vocabularium wordt vaak aangeduid door het symbool Σ.

Bv. $\Sigma = \{P/1, Q/2, F/1:, G/2:, C/0:, C'/0:\}$ is het vocabularium dat bestaat uit predikaatsymbolen $P/1, Q/2$, functiesymbolen $F/1:, G/2:$ en constanten C, C'.

Definitie 2.1.1. Een **term** is een string die kan bekomen worden door herhaaldelijke toepassing van de volgende regels :

- Een constante-symbool c is een term.
- Als $t_1, \ldots, t_n$ n termen zijn, en $G/n:$ een functiesymbool, dan is $G(t_1, \ldots, t_n)$ een term.

Een term t is een term van vocabularium Σ indien alle niet-logische symbolen in t tot Σ behoren.

Een **formule** is een string die kan bekomen worden door herhaaldelijke toepassing van de volgende regels :

- Als $t_1, \ldots, t_n$ n termen zijn en P/n een relatiesymbool van ariteit n is, dan is $P(t_1, \ldots, t_n)$ een formule.
- Als t_1, t_2 termen zijn, dan is $t_1 = t_2$ een formule.

- Als A, B formules zijn die we al geconstrueerd hebben, dan zijn $(\neg A)$, $(A \wedge B)$, $(A \vee B)$, $(A \Rightarrow B)$, $(A \Leftrightarrow B)$ ook formules.
- Als x een constante-symbool is en A een formule, dan zijn $(\exists x)A$ en $(\forall x)A$ ook formules. In zo'n geval wordt x ook een *variabele* genoemd.

Merk op: de definities van term en formule zijn inductieve definities. De verzameling van termen of formules is alles wat geconstrueerd kan worden door iteratief toepassen van deze regels.

Notatie 2.1.2. Formules worden genoteerd als A, B, C, termen door $t, t_1, \ldots$.

Voorbeeld 2.1.1. Neem predicaatsymbolen $P/2, Q/1, R/3$, functiesymbolen $F/1 :, G/2 :$ en constantesymbolen c, x, y.

- $F(G(G(c, x), F(x)))$ is een term.
- $F(x, G(y))$ is geen term (de ariteit van F klopt niet met het aantal argumenten)
- $F(G(x, F(x)$ is geen term wegens ontbrekende haakjes.
- $(P(x, y) \wedge (\forall v)R(c, x, G(x, F(c))))$ is een formule.
- $P(x, y) \wedge F(G(y))$ is geen formule (een conjunctie met een term is niet toegelaten)
- $P(x, y) = Q(c)$ is geen formule (gelijkheid tussen formules is niet toegelaten)
- $P(c, c, c)$ is geen formule wegens een verkeerd aantal argumenten.

Deelformules. Een formule A is een rij van symbolen. Een *deelformule* van een formule A is een formule die voorkomt in A. Het is dus een deelrij van opeenvolgende symbolen van A die zelf een formule is.

Voorbeeld 2.1.2.

$$(\exists v_1)(P(v_0, v_1, v_5, v_2) \Rightarrow \underbrace{(\forall v_3)(Q(v_1, v_2) \wedge R(v_1, v_2, v_3))}_{\text{deelformule}}) \wedge Q(v_5, v_5)$$

Variabelen versus constanten Elk voorkomen van een constante-symbool x in een deelformule $(\exists x)B$ of $(\forall x)B$ van A noemen we een **variabel (of gebonden) voorkomen** van x, en we noemen x de **variabele** van deze deelformule. Variabelen zullen we meestal noteren als x, y, z, v, w.

Elk (niet-logisch) symbool kan meerdere keren voorkomen in een formule. Een voorkomen van een symbool noemen we een **vrij** voorkomen in A als het geen variabel voorkomen is. Bv. het constante-symbool x heeft vier voorkomens in $Q(x) \wedge (\exists x)P(x, x)$. Het eerste voorkomen is vrij, en de drie laatste voorkomens zijn variabel. Dezelfde formule bevat ook vrije voorkomens van predikaatsymbolen $Q/1$ en $P/2$.

Een *vrij symbool* van een formule A is elk symbool met een vrij voorkomen in A. Gegeven een vocabularium Σ, dan noemen we elk vrij symbool van een formule A dat niet voorkomt in Σ een vrije variabele van A (ten op zichte van Σ). Dus, A heeft zowel vrije als gebonden variabelen (t.o.v. Σ).

Definitie 2.1.2. Een **zin** van Σ is een formule A zodat elk vrij symbool van A behoort tot Σ. Een **theorie** van Σ is een verzameling van zinnen van Σ.

Voorbeeld 2.1.3. Neem de volgende zin met predikaatsymbolen $P/2, R/3$ en constante-symbolen v_0, v_1, v_3, c.

$$(\forall v_0)(\exists v_1)(\forall v_3)(P(v_0, v_1) \Rightarrow R(c, v_1, v_3))$$

Dit is geen zin van $\Sigma = \{P/2\}$, evenmin van $\Sigma = \{P/2, R/3\}$ maar wel een zin van $\Sigma = \{P/2, R/3, c/0 :\}$.

De verzameling van alle zinnen van Σ wordt de **predikatenlogicataal** van Σ genoemd. Elk vocabularium heeft dus zijn eigen predicatenlogicataal.

Opmerking 2.1.1. Of een formule A een zin is of niet hangt dus af van het bedoelde vocabularium.

Soorten formules. Een formule $A \Rightarrow B$ noemen we een *implicatie*, A de *premisse* of *conditie* en B de *conclusie*.

Een formule $A \wedge B$ noemen we een *conjunctie*, met A en B de *conjuncten*.

Een formule $A \vee B$ noemen we een *disjunctie*, met A en B de *disjuncten*.

Notatie 2.1.3. Om het geheel overzichtelijker te maken zullen we dikwijls $t_1 \neq t_2$ schrijven in plaats van $\neg\, t_1 = t_2$.

Over de haakjes. Omwille van de duidelijkheid zullen we soms afwijken van de strikte vormingsregels van de formules.

We zullen soms andere haakjes zoals [ ] en { } gebruiken om duidelijker subformules af te bakenen.

Let wel: als je haakjes op de verkeerde plaats toevoegt dan verandert de betekenis. Bv. $\neg Q(c) \wedge Q(x)$ betekent iets anders dan $\neg(Q(c) \wedge Q(x))$.

Net als voor rekenkundige uitdrukkingen zijn er een aantal conventies die toelaten om haakjes weg te laten, zonder onduidelijkheid te creëren over de betekenis van de formule — bv. $1 \times 3 + 4$ staat voor $(1 \times 3) + 4$.

Ten eerste mogen de buitenste haakjes altijd weggelaten worden. Bv. $A \vee B$ staat voor $(A \vee B)$.

Ten tweede mogen we ook haakjes weglaten binnenin de formule op voorwaarde dat het duidelijk is waar ze zouden moeten staan. Waar haakjes weggelaten mogen worden (en waar ze dus teruggezet zouden moeten worden) wordt bepaald door **voorrangsregels**:

1. De kwantoren binden sterker dan de connectieven.
   Bv. $(\exists x)P(x) \wedge Q(x)$ staat voor $((\exists x)P(x)) \wedge Q(x)$, niet voor $(\exists x)(P(x) \wedge Q(x))$.

2. $\neg$ bindt sterker dan alle andere connectieven.
   Bv. $\neg P(c) \wedge Q(c)$ staat voor $(\neg P(c)) \wedge Q(c)$.

3. $\wedge$ bindt sterker dan $\vee$, $\vee$ sterker dan $\Rightarrow$, $\Rightarrow$ sterker dan $\Leftrightarrow$.
   Bv. $Q(x) \vee P(c, c) \wedge Q(c)$ staat voor $(Q(x) \vee (P(c, c) \wedge Q(c)))$ en niet $((Q(x) \vee P(c, c)) \wedge Q(c))$ omdat $\wedge$ sterker bindt en voorrang krijgt op $\vee$.
   Bv. $Q(x) \vee P(c, c) \Rightarrow Q(c) \wedge R(c, x, y)$ staat voor $((Q(x) \vee P(c, c)) \Rightarrow (Q(c) \wedge R(c, x, y)))$

4. Voor identieke connectieven wordt de binding sterker van links naar rechts.
   Bv. $Q(x) \vee P(c,c) \vee Q(c)$ staat voor $((Q(x) \vee P(c,c)) \vee Q(c))$.

Opmerking 2.1.2. Zo'n verkorte notatie zoals $P(x,y) \vee Q(c) \wedge Q(y)$ mag je niet beschouwen als een formule. Het *is* geen formule, wel een *afgekorte schrijfwijze* van een formule. Zoals in het nederlands: "bv." is geen woord maar enkel een afgekorte schrijfwijze van het woord "bijvoorbeeld". Als we zo'n verkorte notatie gebruiken, dan bedoelen we altijd de formule waarvoor ze staat, met de haakjes op de juiste plaats.

Let op met deelformules als je de verkorte schrijfwijze gebruikt! $P(x,y) \vee Q(c)$ is geen deelformule van $P(x,y) \vee Q(c) \wedge Q(y)$! Dat wordt duidelijk als je de haakjes invult: $(P(x,y) \vee Q(c))$ komt niet voor in $(P(x,y) \vee (Q(c) \wedge Q(y)))$.

Teveel haakjes weglaten komt de duidelijkheid van een formule ook niet ten goede. In de cursustekst laten we zelden haakjes weg, maar in de oefeningen zullen we dit vaker doen wanneer we met ingewikkelde formules werken. De bedoeling is tenslotte om een formule op een duidelijke manier te noteren. Daarvoor is ook indentatie nuttig. Schrijf liever

$$\begin{array}{ll}(\forall x) & \{ \\ & \quad (\exists w) S(w,w,x) \wedge (\exists u)(\exists v)(\exists w)[L(u,v) \wedge L(v,w) \wedge L(w,x)] \\ & \quad \Rightarrow \\ & \quad (\exists y)(\exists z)[P(y) \wedge P(z) \wedge S(y,z,x)] \\ & \} \end{array}$$

dan een kopbreker als:

$$(\forall x)((\exists w)(S(w,w,x) \wedge (\exists u)(\exists v)(\exists w)(L(u,v) \wedge L(v,w) \\ \wedge L(w,x))) \Rightarrow (\exists y)(\exists z)(P(y) \wedge P(z) \wedge S(y,z,x)))$$

Opmerking 2.1.3. Er bestaat een grote variatie in de manier waarop men logische formules schrijft. Het lijkt alsof zowat elk boek en elk systeem zijn eigen notatie heeft! Bv. het implicatiesymbool $\Rightarrow$ wordt soms vervangen door $\rightarrow, \Rightarrow, \supset$. Of i.p.v. $(\forall x)A$ zet men de haakjes elders $(\forall x : A)$. Enz. Dit zijn natuurlijk geen belangrijke verschillen; het gaat enkel over *syntactische suiker*. (Maar als je formules in een systeem dient in te typen kan het wel vervelend zijn. Gelukkig wordt dit goed ondersteund in LogicPalet.)

De variatie wordt nog groter als we naar logische softwaretools kijken. Vele systemen beperken zich tot ASCII-karakters, een beperkte set van computersymbolen. Hierin komen symbolen zoals $\Rightarrow, \wedge, \forall, \ldots$ niet voor en moeten vervangen worden door een combinatie van ASCII-karakters. Bv. $\forall$ door `!` en $\exists$ door `?`, $\Rightarrow$ door `=>`, enz. De details hangen af van systeem tot systeem. In de theoremprover Spass bv. schrijft men $A \wedge B$ als `and(`A`,`B`)`, en $(\forall x)A(x)$ wordt `forall([x],`A`(x))`.

LogicPalet ondersteunt een grotere set van karakters namelijk Unicode-symbolen. De symbolen $\forall, \exists, \wedge, \vee, \neg, \Rightarrow, \Leftrightarrow, \neq$ zijn geen ASCII-karakters maar wel Unicode-karakters. (Het laat toe om formules uit electronische documenten met copy en paste over te brengen naar LogicPalet!)

LogicPalet ondersteunt zowel een Unicode-syntax als een ASCII-syntax waarin symbolen

$$\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow, \neq, \forall, \exists$$

vervangen zijn door de ASCII-karakters

$$\&\,, |\,, \sim\,, =>\,, <=>\,, \sim=\,, !\,, ?$$

LogicPalet laat je toe van Unicode-syntax te vertalen naar ASCII-syntax en omgekeerd, en ook naar de syntax van Spass en van het IDP systeem. Probeer enkele voorbeelden uit om te zien hoe de andere connectieven in Spass-syntax geschreven worden.

2.2 Formele semantiek van predicatenlogica

Het basisconcept van de logica is *waarheid*. Wat maakt een logische zin waar, zoals de volgende integriteitsbeperking van de studentendatabank?

$$\neg(\exists x)Prerequ(x, x)$$

Dit hangt natuurlijk af van de *waarde* van $Prerequ$. De zin is waar voor de tabel voor $Prerequ$ in de studentendatabank. Maar de zin kan onwaar worden voor andere waardes:

- waar voor de relatie $<$ in de natuurlijke getallen;
- onwaar voor de relatie $\leq$ in de natuurlijke getallen;
- onwaar voor de relatie "x is even oud als y" in het domein van KUL-studenten.
- ...

De waarheid van een logische zin hangt dus af van de gekozen waardes van de niet-logische symbolen.

Een *structuur*, ook wel *interpretatie* genoemd, is niets anders dan een toekenning van waardes aan niet-logische symbolen.

Definitie 2.2.1. Een **structuur** $\mathfrak{A}$ bestaat uit een niet-lege verzameling $D_{\mathfrak{A}}$, het **domein** of **universum** van $\mathfrak{A}$, en een toekenning van waarden $\tau^{\mathfrak{A}}$ aan niet-logische symbolen τ:

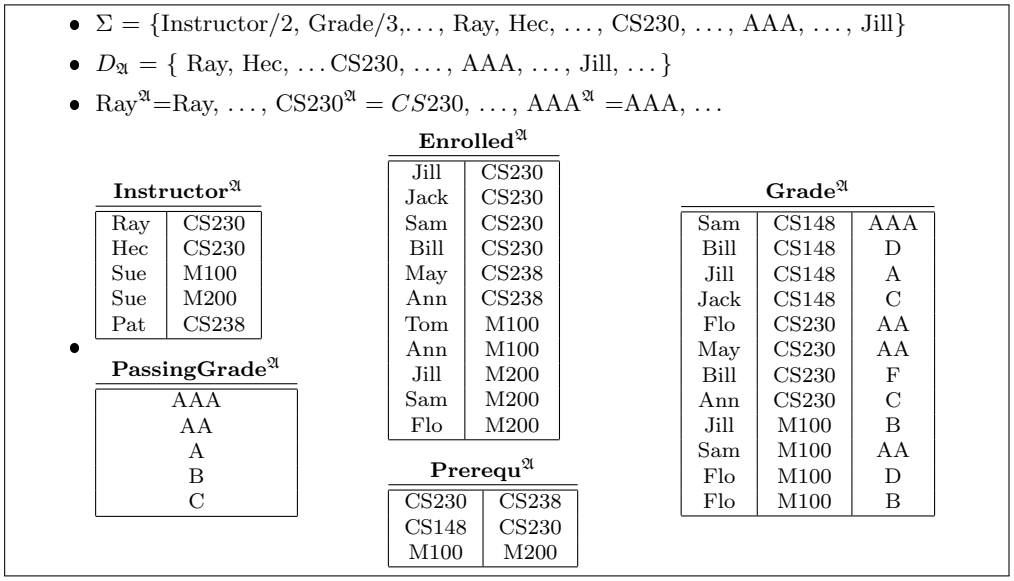
- De waarde $c^{\mathfrak{A}}$ voor een constante c is een element uit het domein $D_{\mathfrak{A}}$.
- De waarde $F^{\mathfrak{A}}$ voor een functie-symbool F/n : is een functie $F^{\mathfrak{A}} : D_{\mathfrak{A}}^n \to D_{\mathfrak{A}}$ die n-tallen $(a_1, \ldots a_n)$ op elementen van het universum afbeeldt.
- De waarde $P^{\mathfrak{A}}$ voor een predicaatsymbool P/n is een n-voudige relatie $P^{\mathfrak{A}}$ in $D_{\mathfrak{A}}$, dus $P^{\mathfrak{A}} \subset D_{\mathfrak{A}}^n$.

We noemen $\tau^{\mathfrak{A}}$ ook de **interpretatie** van τ in $\mathfrak{A}$.

Definitie 2.2.2. Het vocabularium $\Sigma_{\mathfrak{A}}$ van een structuur $\mathfrak{A}$ bestaat uit de verzameling van niet-logische symbolen die een interpretatie hebben in $\mathfrak{A}$.

Een structuur $\mathfrak{A}$ **interpreteert** een term t of logische formule A als elk niet-logisch vrij symbool van t of A een interpretatie heeft in $\mathfrak{A}$. Dus als A een zin is van $\Sigma_{\mathfrak{A}}$.

Opmerking 2.2.1. Gegeven een structuur $\mathfrak{A}$, dan noemen een formule A een zin als A een zin is van $\Sigma_{\mathfrak{A}}$, ma.w., als $\mathfrak{A}$ elk vrij symbool van A interpreteert.



Figuur 2.1: Een database als structuur

Een structuur kent een *waarde* toe aan elk symbool van Σ. Maar een structuur bepaalt ook het antwoord op de volgende vraag: als we zeggen dat "voor alle x geldt dat ...", over welke objecten gaat het dan? In de context van een gegeven structuur $\mathfrak{A}$, gaat het namelijk over al de elementen van het domein, het universum van $\mathfrak{A}$.

Voorbeeld 2.2.1. Twee structuren kunnen dezelfde interpretatie geven aan alle symbolen maar verschillen in hun domein. Bv. beschouw de volgende twee structuren:

- $\mathfrak{A}_1$ heeft domein $\{a, b\}$ en $P^{\mathfrak{A}_1} = \{a\}, Q^{\mathfrak{A}_1} = \{b\}$.
- $\mathfrak{A}_2$ heeft domein $\{a, b, c\}$ en $P^{\mathfrak{A}_2} = \{a\}, Q^{\mathfrak{A}_2} = \{b\}$.

Hoewel $P/1$ne $Q/1$ dezelfde waarde hebben in beide, is de zin $(\forall x)(P(x) \vee Q(x))$ waar in $\mathfrak{A}_1$ en onwaar $\mathfrak{A}_2$. Dit komt omdat de universele kwantor in beide structuren een verschillende betekenis heeft.

Het domein van een structuur bestaat niet noodzakelijk uit strings of getallen of andere wiskundige objecten. Bv. het domein van een structuur kan bestaan uit alle 1ste bachelors van de KUL. Dan zijn c en $c^{\mathfrak{A}}$, P en $P^{\mathfrak{A}}$, en F en $F^{\mathfrak{A}}$ telkens volkomen verschillende objecten: het eerste is telkens een symbool, t.t.z. een rij van tekens, terwijl het tweede een waarde is (domeinelement, relatie of functie). Verwar deze niet!

Voorbeeld 2.2.2. De studentendatabank in Figuur 1.4 kan beschouwd worden als een specifiek soort structuur van een specifiek vocabularium.

Dit vocabularium Σ bestaat uit predikaatsymbolen $Instructor/2, Enrolled/2, \ldots$ en constantenymbolen voor elk van de domeinelementen van de databank.

De studentendatabank is dan de structuur $\mathfrak{A}$ met als domein het databankdomein, voor elk predikaatsymbool $P \in \Sigma$ is $P^{\mathfrak{A}}$ de corresponderende relatie/tabel in Figuur 1.4, en voor elke constante $c \in \Sigma$ is $c^{\mathfrak{A}} = c$. M.a.w., de interpretatie van een constante is zichzelf.

Een databankstructuur heeft een aantal speciale eigenschappen die niet gelden voor structuren in het algemeen.

- Het vocabularium van een databank bevat geen functiesymbolen, enkel constanten en predikaatsymbolen.
- Het domein is eindig, toch voor de eenvoudige databanken die we hier in deze cursus zullen zien.
- Alle constanten hebben zichzelf als interpretatie.
- De interpretatie van twee constanten verschilt dus altijd.
- Elk element x van het domein van een databank is de interpretatie van een constante, namelijk van x zelf.

Voorbeeld 2.2.3. Beschouw de alternatieve structuur $\mathfrak{B}$ van hetzelfde vocabularium als de studentendatabank.

- $D_{\mathfrak{B}} = \{$ Ray, Hec, ... CS230, ..., AAA, ..., Jill, ... $\}$
- Ray$^{\mathfrak{A}}$=Ray, ..., CS230$^{\mathfrak{A}}$=Ray, ..., AAA$^{\mathfrak{A}}$ =Ray, ...
- De interpretaties van de predicaatsymbolen in $\mathfrak{B}$ zijn dezelfde als in Figuur 2.1.

Kortom, we bekomen $\mathfrak{B}$ door elk constantesymbool de waarde *Ray* te geven.

Dit is een legale structuur! Dezelfde zinnen hebben een heel andere betekenis als we ze interpreteren in de structuur $\mathfrak{A}$ van Figuur 2.1 dan in $\mathfrak{B}$. Bv. de zin $Prerequ(CS230, CS238)$ is waar in $\mathfrak{A}$ maar niet in $\mathfrak{B}$ want in $\mathfrak{B}$ betekent deze zin dat Ray voorkennis is van zichzelf, wat niet waar is in $\mathfrak{B}$.

Voorbeeld 2.2.4. Neem Σ bestaande uit predicaatsymbolen $P/1$, $L/2$ en functiesymbool $S/2$: en constanten $zero, one$. Zij $\mathfrak{A}$ de volgende structuur voor Σ:

- Domein $D = \mathbb{N}$, de verzameling van natuurlijke getallen
- $zero^{\mathfrak{A}} = 0$, $one^{\mathfrak{A}} = 1$
- $P^{\mathfrak{A}} = \{n \in \mathbb{N} | n \text{ is een priemgetal}\} = \{2, 3, 5, 7, 11, \dots\}$
- $L^{\mathfrak{A}} = \{(n, m) \in \mathbb{N}^2 | n < m\} = \{(0, 1), (0, 2), \dots, (1, 2), \dots\}$
- $S^{\mathfrak{A}} : \mathbb{N}^2 \to \mathbb{N} : (n, m) \to n + m$; dit is de som-functie.

Merk op dat deze structuur een oneindig domein heeft, dat er slechts twee elementen gedenoteerd worden door constanten, namelijk 0 en 1, en ook dat veel termen dezelfde interpretatie hebben. Bv. $one, S(zero, one), S(one, zero)$ hebben dezelfde interpretatie, namelijk 1. Wat de interpretatie is van een term wordt straks uitgelegd.

Voorbeeld 2.2.5. Een volkomen verschillende maar perfect legale structuur $\mathfrak{A}_1$ van het vocabularium Σ uit het vorige voorbeeld is het volgende:

- Domein $D = \mathbb{N}$, de verzameling van natuurlijke getallen

- $zero^{\mathfrak{A}_1} = one^{\mathfrak{A}_1} = 20$.
- $P^{\mathfrak{A}_1} = \{n \in \mathbb{N} | n \text{ is een even getal}\} = \{2, 4, 6, \dots\}$
- $L^{\mathfrak{A}_1} = \{(n, m) \in \mathbb{N}^2 | n = m^2\} = \{(0, 0), (1, 1), \dots, (4, 2), (9, 3), \dots\}$
- $S^{\mathfrak{A}_1} : \mathbb{N}^2 \to \mathbb{N} : (n, m) \to max(n, m)$; dit is de maximum-functie.

Een logische zin van Σ heeft natuurlijk een totaal andere betekenis als we hem interpreteren in $\mathfrak{A}_1$ dan als we hem interpreteren in de structuur $\mathfrak{A}$ van het vorige voorbeeld. Neem als voorbeeld $(\exists x)L(x, zero)$. Deze betekent in $\mathfrak{A}$ dat er een getal in het universum bestaat dat strikt kleiner dan 0 is, wat onwaar is. In $\mathfrak{A}_1$ betekent de zin dat er een getal bestaat dat het kwadraat is van 20, wat wel waar is. Dat de waarheid van een zin afhangt van de structuur is eigenlijk evident. Zo hangt het antwoord op een query in een databank vanzelfsprekend af van de inhoud van de databank.

Oefening 2.2.1. *Vergelijk de betekenis van* $(\forall x)(\exists y)L(x, y)$ *in beide structuren.*

We zullen zien dat het soms nodig is om structuren uit te breiden voor een symbool of de waarde te wijzigen van een geïnterpreteerd symbool. Zij $\mathfrak{A}$ een structuur, τ een symbool en v een geldige waarde voor het symbool τ in het domein van $\mathfrak{A}$. Dus: v is een domeinelement indien τ een constante is, een n-voudige relatie indien τ een n-voudige predikaatsymbool, en een n-voudige functie indien τ een n-voudig functiesymbool.

Dan is $\mathfrak{A}[\tau : v]$ de structuur die identiek is aan $\mathfrak{A}$ behalve dat $\tau^{\mathfrak{A}[\tau:v]} = v$.

We kunnen een structuur ook voor meerdere symbolen tergelijkertijd uitbreiden of wijzigen en gebruiken daarvoor de notatie $\mathfrak{A}[\tau_1 : v_1, \dots, \tau_n : v_n]$.

Als t een term is geïnterpreteerd door een structuur $\mathfrak{A}$ dan bepaalt t een uniek domeinelement van $\mathfrak{A}$. Dit noemen we de interpretatie van t in $\mathfrak{A}$:

Definitie 2.2.3. Zij gegeven een structuur $\mathfrak{A}$ die een term t interpreteert. We definiëren de interpretatie $t^{\mathfrak{A}}$ van t in $\mathfrak{A}$ door inductie op het aantal symbolen van t:

- Als t een constante is dan is $t^{\mathfrak{A}}$ gekend: het is de interpretatie van t in $\mathfrak{A}$.
- Als t een term $F(t_1, \dots, t_n)$ is, dan is $t^{\mathfrak{A}} = F^{\mathfrak{A}}(t_1^{\mathfrak{A}}, \dots, t_n^{\mathfrak{A}})$, de functie $F^{\mathfrak{A}}$ toegepast op het koppel bestaande uit de interpretaties van $t_1, \dots, t_n$.

Op dezelfde manier kunnen we ook bepalen of een zin A van structuur $\mathfrak{A}$ waar is in $\mathfrak{A}$.

Definitie 2.2.4. Zij A een formule, $\mathfrak{A}$ een structuur met domein D die A interpreteert (t.t.z., A is een zin van $\Sigma_{\mathfrak{A}}$). We definiëren dat A *waar* is in $\mathfrak{A}$ (notatie $\mathfrak{A} \models A$) door middel van inductie op het aantal symbolen in A:

- $\mathfrak{A} \models P(t_1, \ldots, t_n)$ asa $(t_1^{\mathfrak{A}}, \ldots, t_n^{\mathfrak{A}}) \in P^{\mathfrak{A}}$. (Atoom-regel)
- $\mathfrak{A} \models t_1 = t_2$ asa $t_1^{\mathfrak{A}} = t_2^{\mathfrak{A}}$. (=-regel)
- $\mathfrak{A} \models \neg A$ asa $\mathfrak{A} \not\models A$, of in woorden, als het niet het geval is dat $\mathfrak{A} \models A$. ($\neg$-regel)
- $\mathfrak{A} \models A \wedge B$ asa $\mathfrak{A} \models A$ en $\mathfrak{A} \models B$. ($\wedge$-regel)
- $\mathfrak{A} \models A \vee B$ asa $\mathfrak{A} \models A$ of $\mathfrak{A} \models B$ (of allebei). ($\vee$-regel)
- $\mathfrak{A} \models A \Rightarrow B$ asa $\mathfrak{A} \not\models A$ of $\mathfrak{A} \models B$ (of allebei). ($\Rightarrow$-regel)
- $\mathfrak{A} \models A \Leftrightarrow B$ asa $\mathfrak{A} \models A$ en $\mathfrak{A} \models B$ of als $\mathfrak{A} \not\models A$ en $\mathfrak{A} \not\models B$. M.a.w. als A en B dezelfde waarheidswaarde hebben. ($\Leftrightarrow$-regel)
- $\mathfrak{A} \models (\exists x)A$ asa er een domeinelement $a \in D$ bestaat zodat $\mathfrak{A}[x : a] \models A$. ($\exists$-regel)
- $\mathfrak{A} \models (\forall x)A$ asa er voor elk domeinelement $a \in D$ geldt dat $\mathfrak{A}[x : a] \models A$. ($\forall$-regel)

We zeggen dat A *onwaar* is in $\mathfrak{A}$ (notatie $\mathfrak{A} \not\models A$) als A een zin is van $\Sigma_{\mathfrak{A}}$ die niet waar is in $\mathfrak{A}$.

Deze definitie van waarheid van een zin is de belangrijkste van deze cursus. Bijna alle verdere concepten in de cursus zijn erop gebaseerd. Het concept van waarheid is ook van centraal belang in databanken en andere softwaretools voor formele talen.

Opmerking 2.2.2. Waarheid of onwaarheid van een formule A die niet geïntrpreteerd is door $\mathfrak{A}$ is niet gedefinieerd. Je kunt dus niet zeggen dat $\mathfrak{A} \not\models A$ indien $\mathfrak{A}$ de formule A niet interpreteert! Waarheid of onwaarheid is enkel gedefinieerd voor zinnen (van $\Sigma_{\mathfrak{A}}$).

Definitie 2.2.5. Zij $\mathfrak{A}$ een structuur, en A een zin (van $\Sigma_{\mathfrak{A}}$). De waarheidswaarde van A in $\mathfrak{A}$, notatie $A^{\mathfrak{A}}$, is **t** indien $\mathfrak{A} \models A$ en **f** in het andere geval, dus indien $\mathfrak{A} \not\models A$.

Opmerking 2.2.3. De regels in deze definitie vertalen elk logisch connectief in zijn nederlandse versie: $\wedge$ in "en", $\vee$ in "of", $\forall$ in "voor elk", enz. Zoals in:

$$\text{“}\mathfrak{A} \models A \underline{\wedge} B \text{ als } \mathfrak{A} \models A \text{ } \underline{\text{en}} \text{ } \mathfrak{A} \models B\text{”}.$$

Door alle logische connectieven op die manier te vertalen vinden we bv. dat:

$$\begin{gathered}
\mathfrak{A} \models (\forall x)(Q(x) \Rightarrow (\exists y)P(x, z, y)) \\
\text{asa} \\
\text{voor elke } a \in D\text{: } \mathfrak{A}[x : a] \models (Q(x) \Rightarrow (\exists y)P(x, z, y)) \\
\text{asa} \\
\text{voor elke } a \in D\text{: er geldt } \mathfrak{A}[x : a] \not\models Q(x) \text{ (inclusieve) of } \mathfrak{A}[x : a] \models (\exists y)P(x, z, y)) \\
\text{asa} \\
\text{voor elke } a \in D\text{: ofwel behoort } a \text{ niet tot } Q^{\mathfrak{A}}\text{, ofwel bestaat een } b \in D \text{ zodat het koppel} \\
(a, z^{\mathfrak{A}}, b) \text{ tot de relatie } P^{\mathfrak{A}} \text{ behoort} \\
\text{asa} \\
\text{voor elke } a \in Q^{\mathfrak{A}} \text{ bestaat een } b \in D \text{ zodat het koppel } (a, z^{\mathfrak{A}}, b) \text{ tot de relatie } P^{\mathfrak{A}} \text{ behoort.}
\end{gathered}$$

We bekomen dus een vertaling naar een nederlandse zin die spreekt over de structuur. Deze vertaling is erg nuttig als je zelf een logische zin moet evalueren in een structuur.

Definitie 2.2.6. Deze nederlandse zin (of liever, zijn betekenis) wordt ook wel de **informele betekenis** van de zin in $\mathfrak{A}$ genoemd.

Waarheidstabellen. De manier waarop in $\mathfrak{A}$ de waarheid van een formule berekend wordt uit zijn deelformules wordt weergegeven in de volgende waarheidstabellen:

A	$\neg A$
t	**f**
f	**t**

A	B	$A \wedge B$	$A \vee B$	$A \Rightarrow B$	$A \Leftrightarrow B$
t	**t**	**t**	**t**	**t**	**t**
t	**f**	**f**	**t**	**f**	**f**
f	**t**	**f**	**t**	**t**	**f**
f	**f**	**f**	**f**	**t**	**t**

Deze tabellen zijn een gevolg van Definitie 2.2.4.

We zeggen dat A *onwaar* is in $\mathfrak{A}$ (notatie $\mathfrak{A} \not\models A$) als A niet waar is in $\mathfrak{A}$. Maar wanneer is dat precies? De volgende eigenschap maakt dit expliciet. De eigenschap is een gevolg van Definitie 2.2.4.

Eigenschap 2.2.1. Zij A een formule, $\mathfrak{A}$ een structuur met domein D die A interpreteert (t.t.z., A is een zin van $\Sigma_{\mathfrak{A}}$). A is *onwaar* in $\mathfrak{A}$ onder de volgende omstandigheden.

- $\mathfrak{A} \not\models P(t_1, \ldots, t_n)$ asa $(t_1^{\mathfrak{A}}, \ldots, t_n^{\mathfrak{A}}) \notin P^{\mathfrak{A}}$. (Atoom-regel)
- $\mathfrak{A} \not\models t_1 = t_2$ asa $t_1^{\mathfrak{A}} \neq t_2^{\mathfrak{A}}$. (=-regel)
- $\mathfrak{A} \not\models \neg A$ asa $\mathfrak{A} \models A$. (¬-regel)
- $\mathfrak{A} \not\models A \wedge B$ asa $\mathfrak{A} \not\models A$ of $\mathfrak{A} \not\models B$ (of allebei). (∧-regel)
- $\mathfrak{A} \not\models A \vee B$ asa $\mathfrak{A} \not\models A$ en $\mathfrak{A} \not\models B$. (∨-regel)
- $\mathfrak{A} \not\models A \Rightarrow B$ asa $\mathfrak{A} \models A$ and $\mathfrak{A} \not\models B$. (⇒-regel)
- $\mathfrak{A} \not\models A \Leftrightarrow B$ asa $\mathfrak{A} \models A$ en $\mathfrak{A} \not\models B$ of als $\mathfrak{A} \not\models A$ en $\mathfrak{A} \models B$. M.a.w. als A en B niet dezelfde waarheidswaarde hebben. (⇔-regel)
- $\mathfrak{A} \not\models (\exists x)A$ asa voor elk domeinelement $a \in D$ geldt dat $\mathfrak{A}[x : a] \not\models A$. (∃-regel)
- $\mathfrak{A} \not\models (\forall x)A$ asa er een domeinelement $a \in D$ bestaat zodat $\mathfrak{A}[x : a] \not\models A$. (∀-regel)

Bew. Al deze eigenschappen volgen direct uit Definitie 2.2.4. Voor de connectoren volgen ze ook uit de waarheidstabellen.

Het bewijs voor universeel gekwantificeerde formules gaat als volgt. Er geldt $\mathfrak{A} \not\models (\forall x)A$ asa niet voor alle $a \in D$ geldt $\mathfrak{A}[x:a] \models A$. Dat betekent dat voor minstens één $a \in D$ geldt dat A niet waar is in $\mathfrak{A}[x:a]$, t.t.z. $\mathfrak{A}[x:a] \not\models A$.

Het bewijs voor existentieel gekwantificeerde formules is analoog. Q.E.D.

Een nuttige eigenschap.

Eigenschap 2.2.2. Indien $\mathfrak{A}, \mathfrak{A}'$ hetzelfde domein hebben en dezelfde interpretatie geven aan de niet-logische symbolen van A, dan geldt:

$$\mathfrak{A} \models A \quad \text{asa} \quad \mathfrak{A}' \models A.$$

Dus, de waarheidswaarde van A is onafhankelijk van de waarde van niet-logische symbolen die niet voorkomen in A en evenmin van de waarde $v^{\mathfrak{A}}$ van variabelen van A. Dat komt omdat telkens een gekwantificeerde deelformule $(\exists v)B$ of $(\forall v)B$ geëvalueerd wordt, de waarde van v toch wordt vervangen (zie $\exists$- en $\forall$-regel).

Terminologie 2.2.1. Als $\mathfrak{A} \models A$, zeggen we ook dat

- A is waar in $\mathfrak{A}$
- A is voldaan in $\mathfrak{A}$
- $\mathfrak{A}$ voldoet aan A
- $\mathfrak{A}$ is een **model** van A

Als $\mathfrak{A} \not\models A$, dan zeggen we dat A onwaar is in $\mathfrak{A}$.

$\mathfrak{A}$ is een model van een theorie T (een verzameling zinnen) als elke uitspraak uit T waar is in $\mathfrak{A}$.

n-voudige queries. Het relatievoorschrift die we in Hoofdstuk 1 introduceerden om een n-voudige query uit te drukken is een string van de vorm $\{(x_1, \ldots, x_n) : A\}$.

Let wel, zo'n verzamelingenuitdrukking is geen formule of term van predicatenlogica!

Definitie 2.2.7. Zij A een formule, $\mathfrak{A}$ een structuur die alle vrije symbolen van A interpreteert behalve $x_1, \ldots, x_n$. De interpretatie van $\{(x_1, \ldots, x_n) : A\}$ in $\mathfrak{A}$ (notatie $\{(x_1, \ldots, x_n) : A\}^{\mathfrak{A}}$) is de volgende relatie:

$$\{(a_1, ..., a_n) \in D_{\mathfrak{A}}^n | \mathfrak{A}[x_1 : a_1, \ldots, x_n : a_n] \models A\}$$

Daarmee ligt het wiskundig vast wat het antwoord is op een n-voudige query. Een databanksysteem dient dus onderandere om de interpretatie van dergelijke queries te berekenen. De

algoritmes om dit efficiënt te doen voor de enorme databanken van tegenwoordig zijn natuurlijk zeer gesofisticeerd, maar ze zijn in de grond gebaseerd op Definitie 2.2.4. We zullen in een volgend hoofdstuk een eenvoudig algoritme zien om queries op te lossen, dat direct is afgeleid uit Definitie 2.2.4.

Logisch equivalente formules. In de discussie over correcte vertalingen in Hoofdstuk 1, werden twee logische vertalingen van queries *equivalent* genoemd als ze hetzelfde antwoord hebben in alle databanken (ook in de "zinloze" databanken die niet voldoen aan de integriteitsbeperkingen). De volgende definitie veralgemeent dit.

Definitie 2.2.8. Twee formules A en B zijn *logisch equivalent* als in alle structuren die beide interpreteren ze dezelfde waarheidswaarde hebben; m.a.w. als voor elke structuur $\mathfrak{A}$ die A en B interpreteert, geldt:

$$\mathfrak{A} \models A \quad \text{asa} \quad \mathfrak{A} \models B$$

Opmerking 2.2.4. In LogicPalet is het tool ASKSpass beschikbaar om de logische equivalentie van twee formules na te gaan. Zoals de naam suggereert wordt hiervoor de theoremprover Spass aangeroepen.

2.3 Berekenen van de waarheid van zinnen

In deze sectie gebruiken we Definitie 2.2.4 om een gedetailleerd bewijs te geven van de waarheid of onwaarheid van zinnen over de studentengegevensbank uit Hoofdstuk 1.

Dat doen we in de eerste plaats om te laten zien dat de wiskundige definitie correct is, t.t.z. de nederlandse vertaling van een zin is waar in een structuur asa we kunnen bewijzen dat de zin waar is op basis van de regels van Definitie 2.2.4.

Voorbeeld 2.3.1. Geg. de studentendatabankstructuur $\mathfrak{A}$ uit Figuur 2.1.

T.B. $\mathfrak{A} \models (\exists x) Instuctor(Ray, x)$.

Of in woorden, bewijs dat $(\exists x) Instuctor(Ray, x)$ waar is in $\mathfrak{A}$.

De zin betekent dat Ray een vak doceert. In de studentendatabank is deze zin waar. Immers, Ray doceert het vak $CS230$. Een gedetailleerd bewijs dat $\mathfrak{A} \models (\exists x) Instuctor(Ray, x)$ waar is in $\mathfrak{A}$ maakt dit expliciet.

Bew. Volgens de atoom-regel geldt: $Instructor(Ray, x)$ is waar in een structuur $\mathfrak{B}$ asa $(Ray^{\mathfrak{B}}, x^{\mathfrak{B}}) \in Instructor^{\mathfrak{B}}$.

Aangezien $Ray^{\mathfrak{A}} = Ray$ en $(Ray, CS230) \in Instructor^{\mathfrak{A}}$ geldt dat $Instructor(Ray, x)$ waar is in $\mathfrak{A}[x : CS230]$, t.t.z. $\mathfrak{A}[x : CS230] \models Instructor(Ray, x)$.

Uit de $\exists$-regel volgt dan dat $(\exists x) Instructor(Ray, x)$ waar is in $\mathfrak{A}$.

We herformuleren dit bewijs met een compacte notatie die ook in de oefeningen gebruikt zal worden. Het bewijs bestaat uit een boom van te bewijzen uitspraken en redeneerstappen bestaande uit afleidingen van deze uitspraken door middel van regels van Definitie 2.2.4. Ook wordt gebruik gemaakt van gevallenonderscheid voor universele kwantor.

TB (1) $\mathfrak{A} \models (\exists x) Instuctor(Ray, x)$
TB (2) $\mathfrak{A}[x : CS230] \models Instructor(Ray, x)$
$(Ray, CS230) \in Instructor^{\mathfrak{A}}$ en $Ray^{\mathfrak{A}} = Ray$
(2) volgt nu uit Atoom-regel
(1) volgt uit $\exists$-regel toegepast op (2)

Q.E.D.

Voorbeeld 2.3.2.

T.B. $\mathfrak{A} \models (\exists x)(\exists w)(Enrolled(x, w) \wedge (\forall u)(Enrolled(x, u) \Rightarrow u = w))$

Bew. Vertaald in het nederlands betekent deze zin dat er een student is die voor juist 1 vak is ingeschreven. Dat is het geval voor student $Jack$ en cursus $CS230$. Het zijn deze waarden die we kiezen als toekenning voor x en w. Hieronder volgt het bewijs in de compacte notatie.

TB (1)$\mathfrak{A} \models (\exists x)(\exists w)(Enrolled(x, w) \wedge (\forall u)(Enrolled(x, u) \Rightarrow u = w))$
TB (2) $\mathfrak{A}[x : Jack] \models (\exists w)(Enrolled(x, w) \wedge (\forall u)(Enrolled(x, u) \Rightarrow u = w))$
TB (3) $\mathfrak{A}[x : Jack, w : CS230] \models Enrolled(x, w) \wedge (\forall u)(Enrolled(x, u) \Rightarrow u = w)$
TB (4) $\mathfrak{A}[x : Jack, w : CS230] \models Enrolled(x, w)$
$(Jack, CS230) \in Enrolled^{\mathfrak{A}}$
(4) volgt uit Atoom-regel
TB (5) $\mathfrak{A}[x : Jack, w : CS230] \models (\forall u)(Enrolled(x, u) \Rightarrow u = w)$
TB (6) voor elke $a \in D_{\mathfrak{A}}$: $\mathfrak{A}[x : Jack, w : CS230, u : a] \models Enrolled(x, u) \Rightarrow u = w$
A) Voor $a = CS230$:
$\mathfrak{A}[x : Jack, w : CS230, u : a] \models u = w$ (=-regel)
Voor $a = CS230$ volgt (6) nu uit $\Rightarrow$-regel
B) Voor de andere $a \in D_{\mathfrak{A}}$ $(a \neq CS230)$
$(Jack, a) \notin Enrolled^{\mathfrak{A}}$ (Jack enkel in CS230)
$\mathfrak{A}[x : Jack, w : CS230, u : a] \not\models Enrolled(x, u)$ (Atoom-regel)
Voor andere a volgt (6) uit $\Rightarrow$-regel
Uit A+B volgt (6) voor alle $a \in D_{\mathfrak{A}}$
(5) volgt $\forall$-regel op (6)
(3) volgt uit $\wedge$-regel op (4) en (5)
(2) volgt uit $\exists$-regel op (3)
(1) volgt uit $\exists$-regel op (2).

Merk op dat de laatste stap van een te bewijzen meestal een triviale toepassing is van de gepaste regel van Definitie 2.2.4. In de oefenzittingen hoef je deze niet te schrijven.

Q.E.D.

Voorbeeld 2.3.3.

T.B. $\mathfrak{A} \not\models (\forall x)\{Enrolled(x, CS230) \Rightarrow (\exists w)[\ Instructor(Sue, w) \wedge (\exists s)(Grade(x, w, s) \wedge PassingGrade(s))]\}$

Bew. De zin drukt uit dat elke student ingeschreven voor CS230, geslaagd is voor een vak van Sue. Dit is niet het geval. Een tegenvoorbeeld is $Jack$. Immers, $Jack$ is ingeschreven voor $CS230$ maar hij heeft geen score voor de vakken van Sue, dus zeker geen slaag-score. Deze keer moeten we dus aantonen dat een zin onwaar is. We gebruiken dezelfde notatie.

TB (1) $\mathfrak{A} \not\models (\forall x)\{Enrolled(x, CS230) \Rightarrow (\exists w)Instructor(Sue, w) \wedge (\exists s)(Grade \ldots$
TB (2) $\mathfrak{A}[x : Jack] \not\models Enrolled(x, CS230) \Rightarrow (\exists w)Instructor(Sue, w) \wedge (\exists x)Grade \ldots$
TB (3) $\mathfrak{A}[x : Jack] \models Enrolled(x, CS230)$ (de premisse geldt)
$(Jack, CS230) \in Enrolled^{\mathfrak{A}}$ en $CS230^{\mathfrak{A}} = CS230$
(3) volgt uit Atoom-regel
TB (4) $\mathfrak{A}[x : Jack] \not\models (\exists w)Instructor(Sue, w) \wedge (\exists x)Grade \ldots$ (de conclusie geldt niet)
TB Voor elke $a \in D_{\mathfrak{A}}$: (5) $\mathfrak{A}[x : Jack, w : a] \not\models Instructor(Sue, w) \wedge (\exists s)Grade \ldots$
A) Veronderstel $a = M100$ of $a = M200$
TB (6) $\mathfrak{A}[x : Jack, w : a] \not\models (\exists s)Grade(x, w, s) \wedge PassingGrade(s)$
TB Voor elke $d \in D_{\mathfrak{A}}$: (7) $\mathfrak{A}[x : Jack, w : a, s : d] \not\models Grade(x, w, s) \wedge PassingGrade(s)$
Veronderstel $d \in D_{\mathfrak{A}}$:
$(Jack, a, d) \notin Grade^{\mathfrak{A}}$
$\mathfrak{A}[x : Jack, w : a, s : d] \not\models Grade(x, w, s)$ (Atoom-regel)
(7) volgt uit $\wedge$-regel
(6) volgt uit $\exists$-regel op (7)
(5) volgt uit $\wedge$-regel op (6)
B) Veronderstel $a \in D_{\mathfrak{A}}$ zodat $a \neq M100$ en $a \neq M200$
TB (8) $\mathfrak{A}[x : Jack, w : a] \not\models Instructor(Sue, w)$
$(Sue, a) \notin Instructor^{\mathfrak{A}}$ en $Sue^{\mathfrak{A}} = Sue$
(8) volgt uit Atoom-regel
(5) volgt uit $\wedge$-regel op (8)
Uit A+B volgt (5) voor elke $a \in D_{\mathfrak{A}}$
(4) volgt uit $\exists$-regel op (5)
(2) volgt uit $\Rightarrow$-regel op (3) en (4)
(1) volgt uit $\forall$-regel op (2)

Q.E.D.

Oefening 2.3.1. *Geef een gedetailleerd bewijs van de waarheid of onwaarheid van andere formules uit Hoofdstuk 1.*

Oefening 2.3.2. *Bewijs dat de interpretatie van*

$$\{x : (\exists y)Prerequ(x, y) \wedge \neg(\exists y)Prerequ(y, x)\}$$

de volgende verzameling is : $\{CS149, M100\}$.

Opmerking 2.3.1. Het is belangrijk om in staat te zijn om een gedetailleerd bewijs te geven waarom een zin waar of onwaar is in een structuur. Het laat zien dat je de definitie ook begrijpt.

De techniek die we hierboven hebben toegepast bestaat uit twee stappen:

- eerst de zin vertalen in een nederlandse zin over die structuur en te bepalen of deze waar is of niet en argumenteren waarom niet
- vervolgens dit argument vertalen in een reeks toepassingen van de regels van Definitie 2.2.4.

Opmerking 2.3.2. LogicPalet bevat een uitlegtool om uit te leggen waarom een zin waar of onwaar is in een gegeven structuur. Deze verklaringen zijn in feite niets anders dan bewijzen van de waarheid of onwaarheid van een zin in een structuur. Probeer deze functionaliteit uit om te zien welke soort verklaringen je krijgt.

2.4 Waarheid in oneindige structuren

In de wiskundige logica zijn vrijwel alle "interessante" structuren oneindig. Bv. de verschillende soorten getallen $\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}, \ldots$. Natuurlijk komen ook vaak getallen of numerieke uitdrukkingen voor in toepassingen van logica in de informatica, zodat de structuren in zo'n toepassing ook oneindig zijn.

Eenvoudige algoritmes om de waarheid van een zin in een oneindige structuur na te gaan bestaan er niet. Toch kunnen we soms de waarheid van een zin achterhalen.

Neem het vocabularium en structuur $\mathfrak{A}$ met domein $\mathbb{N}$ uit Voorbeeld 2.2.4:
- $P^{\mathfrak{A}}$ is de verzameling van priemgetallen.
- $L^{\mathfrak{A}} = \{(n, m) | n < m\}$, de "strikt kleiner dan"-relatie
- $S^{\mathfrak{A}} = \mathbb{N}^2 \to \mathbb{N} : (n, m) \to n + m$, de som-functie.

Voorbeeld 2.4.1.

T.B. $\mathfrak{A} \models (\forall x)\neg L(x, x)$

Bew. Voor een willekeurig getal $a \in \mathbb{N}$ is $\neg L(x, x)$ waar in $\mathfrak{A}[x : a]$ asa $(a, a) \notin L^{\mathfrak{A}}$ asa $a \not< a$. Dit is het geval voor elk natuurlijk getal.

Uit de $(\forall)$-regel volgt dan het te bewijzen.

In $\mathfrak{A}$ betekent deze zin dat geen enkel getal strikt kleiner is dan zichzelf.

Voorbeeld 2.4.2.

T.B. $\mathfrak{A} \models (\forall x)(\exists y)L(x, y)$.

Bew. We zullen deze keer gebruik maken van de nederlandse vertaling van deze zin. De vertaling is: voor elk getal n bestaat er een getal m zodat $n < m$. Dat is zo, bv. $m = n + 1$. Q.E.D.

Voorbeeld 2.4.3.

T.B. $\mathfrak{A} \not\models (\exists y)(\forall x)L(x, y)$.

Bew. Deze zin betekent dat er een getal n bestaat zodat elk getal m strikt kleiner is dan n. Dat is niet het geval aangezien bv. $n \not< n$. Q.E.D.

Opmerking 2.4.1. De zinnen in Voorbeelden 2.4.2 en 2.4.3 zijn identiek behalve dat de kwantoren verwisseld werden: $(\forall x)(\forall y)$ versus $(\exists y)(\forall x)$. Dit laat zien dat kwantoren verwisselen de betekenis van een zin verandert.

Voorbeeld 2.4.4. De volgende zin is ook waar, maar niet zo makkelijk te bewijzen.

$$(\forall x)(P(x) \Rightarrow (\exists y)(P(y) \wedge L(x, y)))$$

In $\mathfrak{A}$ betekent deze zin: voor elk priemgetal bestaat een strikt groter priemgetal. Dat is het geval want er bestaan oneindig veel priemgetallen.

Bew. (niet kennen voor het examen)
We bewijzen deze eigenschap uit het ongerijmde[1]. Veronderstel dat de zin onwaar is in $\mathfrak{A}$. Dus bestaat een priemgetal n zodat er geen groter priemgetal bestaat dan n. Dan is n het grootste priemgetal. Nu is het getal $m = (2 \times 3 \times \cdots \times n - 1 \times n) + 1$ zeker groter dan n en dus geen priemgetal, en aangezien $m - 1$ deelbaar door elk getal in $[2, n]$ is m zelf ondeelbaar voor elk van die getallen. Maar elk getal dat geen priemgetal is, is te schrijven als een produkt van een aantal priemgetallen. Deze priemgetallen zijn delers van m en dus strikt groter dan n. Dat is een contradictie. Q.E.D.

Voorbeeld 2.4.5. De laatste zin die we bekijken is:

$$\begin{array}{ll} (\forall x) & \{ \\ & \quad (\exists w) S(w,w) = x \wedge (\exists u)(\exists v)(\exists w)[L(u,v) \wedge L(v,w) \wedge L(w,x)] \\ & \quad \Rightarrow \\ & \quad (\exists y)(\exists z)[P(y) \wedge P(z) \wedge S(y,z) = x)] \\ & \} \end{array}$$

Wat betekent deze zin? Daarvoor moeten we hem eerst ontleden.
- $(\exists w) S(w,w) = x$ betekent dat x een even getal is.
- $(\exists u)(\exists v)(\exists w)[L(u,v) \wedge L(v,w) \wedge L(w,x)]$ betekent dat x strikt groter is dan 2.
- de conclusie zegt dat x de som is van twee priemgetallen.

Deze zin betekent dus dat elk even getal $n > 2$ de som is van twee priemgetallen.

Is deze zin waar in $\mathfrak{A}$? Niemand weet het! Dit is *Goldbach's conjectuur*. Er is een beloning van 1 miljoen dollar voor wie het antwoord vindt.

Oefening 2.4.1. *Ga na wat de betekenis is van de zinnen van deze voorbeelden in de structuur $\mathfrak{A}_1$ waarin L geïnterpreteerd wordt als "... is strikt groter dan ...".*

Oefening 2.4.2. *Neem Σ bestaande uit 1 relatiesymbool, namelijk $L/2$. Beschouw twee structuren:*

- *$\mathfrak{A}_1$ heeft als domein $\mathbb{Z}$, en $L^{\mathfrak{A}_1} = \{(n,m) \in \mathbb{Z}^2 | n < m\}$.*
- *$\mathfrak{A}_2$ heeft als domein $\mathbb{Q}$, en $L^{\mathfrak{A}_2} = \{(n,m) \in \mathbb{Q}^2 | n < m\}$.*

Toon aan dat de volgende zin waar is in één en onwaar in de andere structuur.

$$(\forall x)(\forall y)(L(x,y) \Rightarrow (\exists z)(L(x,z) \wedge L(z,y)))$$

Ter informatie 1. Een belangrijke vraag is of er algoritmes bestaan om de waarheid van zinnen in zo'n oneindige structuren te bepalen.

Voor sommige structuren is dat het geval, voor andere niet. Bv. voor de oneindige structuur $\mathfrak{A}_1$ die je bekomt door $\mathfrak{A}$ uit Voorbeeld 2.2.4 te beperken tot het vocabularium $\Sigma' = \{zero, one, S/2 :$

[1] Een bewijs uit het ongerijmde vertrekt vanuit de hypothese dat het gegeven waar en het te bewijzen onwaar is. Vervolgens wordt een contradictie afgeleid.

} bestaat een algoritme om de waarheid van elke zin van Σ' in $\mathfrak{A}_1$ te bepalen. Deze eigenschap wordt de *beslisbaarheid van Presburger-rekenkunde* genoemd.

Maar als we $\mathfrak{A}_1$ een beetje uitbreiden tot $\mathfrak{A}_2$ door een nieuw functiesymbool $Times/2$: te introduceren waarvoor $Times^{\mathfrak{A}_2} : \mathbb{N}^2 \to \mathbb{N} : (n, m) \to n \times m$ (de produktfunctie $\times$), dan bekomen we een structuur waarvoor geen algoritme bestaat dat de waarheid van elke zin kan bepalen. Deze eigenschap wordt de *onbeslisbaarheid van de natuurlijke getallen* of ook de *Stelling van Church-Turing* genoemd naar de wiskundige logici die dit oorspronkelijk bewezen hebben.

Beide eigenschappen zijn beroemde resultaten uit de logica. Het algoritme voor Presburger rekenkunde heeft veel toepassingen in de informatica.

2.5 Structuren construeren waarin een zin waar is

Nu gaan we op zoek naar structuren waarin een gegeven logische zin waar of juist onwaar is.

Opmerking 2.5.1. Volgens de $\neg$-regel is een zin A onwaar in een structuur asa zijn negatie $\neg A$ waar is in die structuur. Het nuttige gevolg is dat een methode om een structuur te construeren waarin een zin waar is, ook bruikbaar is om een structuur te construeren waarin een zin A onwaar is. Het volstaat de methode toe te passen op $\neg A$.

Voorbeeld 2.5.1. Neem Σ bestaande uit 1 predikaatsymbool $L/2$. We zoeken een structuur van Σ die voldoet aan volgende zin:

$$(\forall x)(\exists y)L(x, y) \wedge \neg(\exists y)(\forall x)L(x, y)$$

Vaak bestaan er heel kleine structuren waarin een zin waar is. Dit is zo'n geval. Neem de structuur $\mathfrak{A}$ met 2 domeinelementen:

- $D = \{a, b\}$
- $L^{\mathfrak{A}} = \{(a, a), (b, b)\}$

Het is makkelijk te verifiëren dat de zin voldaan is in deze structuur.

Er geldt dat $\mathfrak{A} \models (\forall x)(\exists y)L(x, y)$ en dat $\mathfrak{A} \not\models (\exists y)(\forall x)L(x, y)$. Wat meteen bewijst dat deze twee zinnen niet logisch equivalent zijn. Het vinden van een structuur waarin één zin waar is en de andere onwaar toont aan dat de zinnen niet equivalent zijn.

Het is interessant om zien dat de zinnen hierboven enkel verschillen door een omwisseling van de kwantoren. Omwisselen van kwantoren bewaart dus de equivalentie niet. We komen hierop terug in Sectie **??**.

Oefening 2.5.1. *Zoek een structuur die de gelijkaardige zin waar maakt:*

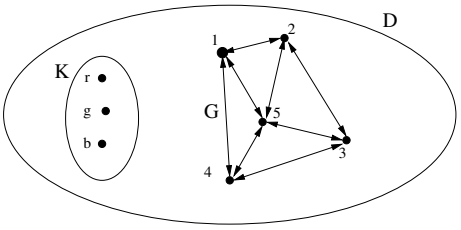
$$\neg(\forall x)(\exists y)L(x, y) \wedge (\exists y)(\forall x)L(x, y)$$

Vindt je er zo één? Zo ja dan verwijzen we naar eigenschap 3.3.9(3) in de discussie over het verwisselen van kwantoren in Hoofdstuk 3.

Voorbeeld 2.5.2. Zij Σ bestaande uit predikaatsymbolen $G/2, K/1$ en functiesymbool $C/1$:. We zoeken een structuur die aan deze zin voldoet:

$$(\forall x)K(C(x)) \wedge (\forall x)(\forall y)(G(x, y) \Rightarrow \neg C(x) = C(y))$$

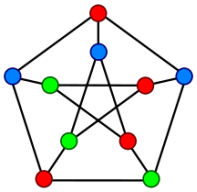
Bovendien moeten de interpretaties $K^{\mathfrak{A}}$ en $G^{\mathfrak{A}}$ zijn zoals de volgende figuur:



Dus $K^{\mathfrak{A}}$ bestaat uit 3 objecten, terwijl $G^{\mathfrak{A}}$ een symmetrische grafe is bestaande uit 16 koppels. Tussen haakjes, een grafe is een 2-voudige relatie. Een symmetrische grafe bevat voor elke boog (a, b) ook zijn omgekeerde boog (b, a). Het probleem hier is dus om een functie $C^{\mathfrak{A}}$ te vinden zodat de volledige structuur voldoet aan de bovenstaande zin. Een oplossing is:

x	1	2	3	4	5	r	g	b
$C^{\mathfrak{A}}(x)$	b	g	b	g	r	r	g	b

Dit probleem is een *grafenkleuringsprobleem*, een veralgemening van het *kaartkleuringsprobleem* (map colouring), waarin het de bedoeling is om elk land zo te kleuren dat aanpalende landen verschillende kleuren hebben. $K^{\mathfrak{A}}$ kan beschouwd worden als een verzameling kleuren, $G^{\mathfrak{A}}$ als een grafe, en $C^{\mathfrak{A}}$ als een kleuring van elk node van de grafe (en ook van de kleuren, maar deze zijn irrelevant). Een grafekleuring is een toekenning van een kleur aan elke node zodat twee verbonden nodes verschillende kleuren hebben.



Oefening 2.5.2. *Zoek een structuur voor Σ bestaande uit predikaatsymbool $L/2$ die aan volgende zin voldoet.*

$$(\forall x)(\forall y)(\forall z)(L(x,y) \wedge L(y,z) \Rightarrow L(x,z)) \wedge$$
$$(\neg(\exists x)L(x,x)) \wedge$$
$$(\forall x)(\exists y)L(x,y)$$

Pas op: uw structuur mag (of liever moet) oneindig zijn. Het is niet moeilijk om aan te tonen dat deze zin onwaar is in elke eindige structuur.

Oefening 2.5.3. *Beschouw de volgende uitspraak over de studentendatabank afkomstig uit Hoofdstuk 1:*

$$(\exists x)(\forall y)((\exists w)Instructor(y,w) \Rightarrow (\exists u)(Instructor(y,u) \wedge Enrolled(x,u)))$$

Pas de studentendatabank aan zodat deze zin waar wordt.

Dit is een variatie van het probleem van het construeren van een structuur omdat er al een structuur gegeven is.

Merk op dat een radikale oplossing is om de tabel van Instructor leeg te maken. Immers dan is de premisse van de implicatie onwaar voor elke y, en dus is de implicatie voldaan.

Zoek liever naar een kleine *aanpassing van de databank zodat de zin voldaan is. Dan doe je eigenlijk hetzelfde als wat een databankmanager doet als hij uitvist dat een integriteitsbeperking niet voldaan is en vervolgens de databank aanpast om de fout te herstellen. Let wel, misschien zijn er meerdere mogelijkheden.*

Je zult merken dat het opstellen van structuren die aan een zin voldoen veel inzicht geeft in deze zin. Maar het kan lijken alsof zoiets weinig practisch nut heeft. Niets is minder waar. Een brede klasse van informaticaproblemen komt neer op het opstellen van structuren die aan bepaalde zinnen voldoen. Alleen al het grafekleuringsprobleem heeft veel toepassingen in de informatica. We zullen daarop terugkomen in Hoofdstuk 4.

2.6 Geo-werelden en Decawerelden

Een zeer belangrijk en nuttig onderdeel van deze cursus zijn de oefeningen via het software-tool LogicPalet. Die leren je (onder andere):

- hoe eigenschappen te vertalen in logica,
- hoe de waarheid van logische zinnen in een structuur te bepalen,
- hoe structuren te construeren waarin een logische zin waar of onwaar is.

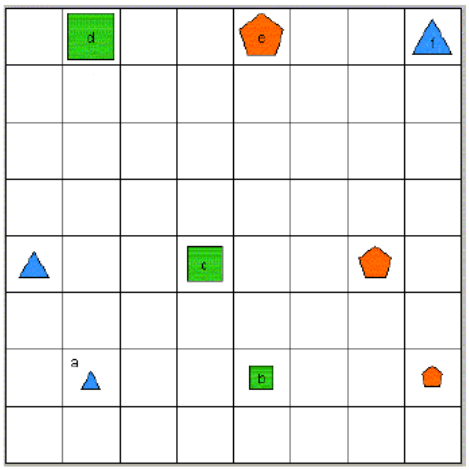
Een eerste toepassing is Geo-werelden. Een Geo-wereld is een structuur van een vocabularium Σ_{Geo} dat een aantal constanten $a, b, c, \ldots$ bevat en de volgende relaties.

$Triangle/1$	... is een driehoek
$Square/1$	... is een vierkant
$Pentagon/1$	... is een vijfhoek
$Small/1$	... is klein
$Medium/1$	... is middelgroot
$Large/1$	... is groot
$Smaller/2$	... is strikt kleiner dan ...
$Larger/2$	... is strikt groter dan ...
$LeftOf/2$	... is strikt meer naar links gelegen dan ...
$RightOf/2$	... is strikt meer naar rechts gelegen dan ...
$FrontOf/2$	... is strikt meer naar voren gelegen dan ...
$BackOf/2$	... is strikt meer naar achteren gelegen dan ...
$Between/3$	... bevindt zich tussen ... en ...

Deze structuren worden door LogicPalet grafisch voorgesteld zoals in Figuur 2.1. In deze wereld geldt bv. $LeftOf(d, e)$ en $LeftOf(d, c)$ (merk op dat d en c niet op dezelfde hoogte hoeven te liggen).

Met LogicPalet kun je vervolgens:

- een nederlandse zin vertalen in een logische zin van Σ_{Geo}
- LogicPalet de waarheid van deze zin in een Geo-wereld laten berekenen



Figuur 2.2: Een Geo-wereld

- met een grafische editor een nieuwe Geo-wereld samenstellen, bv. waarin een gegeven logische zin waar is.
- LogicPalet Geo-werelden laten genereren die aan een bepaalde logische zin voldoen
- De equivalentie van twee zinnen over Geo-werelden laten controleren door LogicPalet, of verifiëren of één zin de andere impliceert.

In deze toepassing ligt het vocabularium Σ_{Geo} vast (op de constanten na). In een andere deel van LogicPalet, DecaWorld, kun je zelf het vocabularium bepalen, maar het domein is beperkt tot hoogstens 10 elementen. Op het grafische na is de functionaliteit van het systeem dezelfde als voor Geo-werelden.

Er zijn een aantal huiswerken en tussentijdse toetsen die gemaakt moeten worden via LogicPalet. Informatie hierover wordt beschikbaar gemaakt via Toledo. Voor meer uitleg verwijzen we naar Toledo, de oefenzittingen en naar de documentatie bij LogicPalet.

2.7 Materiële implicatie

De materiële implicatie. Zo wordt het implicatiesymbool $\Rightarrow$ in predikatenlogica genoemd.

Over het algemeen is er een goede match tussen de logische connectieven en hun natuurlijke taal versie ($\wedge$ versus "en", $\vee$ versus "of", enz.). Dat is niet verbazend aangezien we zagen dat regels van Definitie 2.2.4 het logisch connectief gewoon vertalen in het nederlandse voegwoord.

De uitzondering is $\Rightarrow$, en zijn definitie is inderdaad soms wat problematisch. We vertalen $A \Rightarrow B$ in het nederlands als "als A dan B". Maar uit de $\Rightarrow$-regel volgt echter dat $A \Rightarrow B$ en $\neg A \vee B$ logisch equivalent zijn. De vraag is dus: hebben "als A dan B" en "A is onwaar of B is waar" wel dezelfde betekenis?

Bekijk bijvoorbeeld de uitspraak "Als vandaag de zon schijnt dan is logica een belangrijk vak". Aannemende dat logica inderdaad een belangrijk vak is, dan is deze uitspraak waar, althans volgens predikatenlogica. Dat strookt misschien niet volledig met ons taalgevoel. Het weer van de dag heeft niets te maken met het belang van het vak logica. Op die grond zouden sommigen

misschien verwachten dat de uitspraak onwaar is. Alleszins is de uitspraak wel vreemd; het is niet iets dat mensen zouden zeggen.

Bekijk vervolgens de uitspraak "Als de maan een dampkring heeft dan is het vak logica onbelangrijk". Daar de maan geen dampkring heeft is de uitspraak waar, volgens predikatenlogica. Weer zou men dezelfde kritiek kunnen uiten als hierboven.

Een meer frappant voorbeeld is de volgende voorwaardelijke uitspraak: "*als het gisteren had geregend, dan zou de barbecue niet doorgegaan zijn.*". Deze zin lijkt van de vorm $P \Rightarrow Q$, met P "het regende gisteren" en Q "de barbecue gaat niet door". Veronderstel dat er een grote tent was voorzien voor de barbecue voor het geval het zou regenen, en dat het gisteren niet regende. Dan is P onwaar in de "structuur" die overeenkomt met de werkelijke wereld. Bijgevolg is $P \Rightarrow Q$ waar in deze structuur wegens een onware premisse. Maar anderzijds zou de barbecue sowieso doorgegaan zijn, ook als het gisteren wel geregend had. Dus is de voorwaardelijke zin onwaar. Daarom is $P \Rightarrow Q$ geen correcte vertaling ervan! Wat betekent $P \Rightarrow Q$ dan wel? Het betekent "*het regende gisteren niet of de barebecue ging niet door*". Deze zin is inderdaad waar in de werkelijke wereld (waar het niet regende) en betekent duidelijk iets anders dan de voorwaardelijke zin.

Het probleem zit hem in de voorwaardelijk wijs. In die wijs spreken we immers niet over de structuur die de werkelijke wereld is, maar over structuren/werelden die afwijken van de werkelijke wereld. In het voorbeeld spreken we over werelden die verschillen van de echte wereld doordat het er gisteren wel regende. De zin zegt dat in die werelden de barbecue niet plaats vond. En dat is niet het geval, want in zo'n werelden ging de barbecue wel door, namelijk in de tent. Zo'n verwijzingen naar verschillende universums kunnen niet uitgedrukt worden in predikatenlogica.

Het werkelijke probleem is dat "als ... dan ..." in het nederlands (of zijn vertaling in andere natuurlijke talen) geen unieke betekenis heeft. Net zoals zoveel nederlandse woorden heeft "als ... dan ..." verschillende betekenissen, en hoe we een "als... dan..." interpreteren hangt af van de context waarin deze woorden gebruikt worden. Dus, hoe we $\Rightarrow$ ook definiëren in logica, er zullen altijd uitspraken zijn waarvoor de formele betekenis niet overeen zal komen met ons taalgevoel, eenvoudig weg omdat voor die nederlandse uitspraak ons taalgevoel er onbewust voor zorgt dat we een andere interpretatie voor "als ... dan ..." kiezen dan degene die we net gedefinieerd hebben in de logica.

Het is niet mogelijk om in logica aan een logisch symbool verschillende betekenissen te geven. Het zou ook niet gewenst zijn. Immers, ambiguïteiten willen we ten alle prijze vermijden in logica. Het enige dat we kunnen doen is: voor een frequent voorkomende betekenis van "als ... dan ... " een symbool kiezen. En dat is het geval met $\Rightarrow$. In de meeste gevallen is de materiële implicatie wel degelijk de juiste interpretatie van "als ... dan ...". De kans is groot dat bij het oefenen van deze cursus, je niet één keer een probleem ontdekt met een materiële implicatie, hoewel je er tientallen zult neerschrijven.

Dat neemt niet weg dat er zeer frappante voorbeelden van "als ... dan ... " zinnen bestaan waar materiële implicatie niet de juiste interpretatie is. Voorwaardelijke zinnen zijn zo'n voorbeelden maar er zijn er ook nog andere. Bekijk de volgende twee zinnen en hun voorgestelde formalisatie:

- Als je alle vakken succesvol aflegt, ben je geslaagd voor de bachelor Informatica:

$$((\forall c) Succ(c)) \Rightarrow Pass.$$

- Er is een vak zodat, als je het succesvol aflegt, je geslaagd bent voor de bachelor Informa-

tica:

$$(\exists c)(Succ(c) \Rightarrow Pass).$$

Deze nederlandse zinnen zijn zeker niet equivalent. Immers, de eerste is waar voor de bachelor Informatica, en de tweede is onwaar. Vreemd genoeg zijn de formele beweringen wel logisch equivalent. Ze kunnen omgezet worden in mekaar door middel van de fundamentele equivalenties die we in het volgende hoofdstuk zullen zien:

$$\begin{array}{lll} (\exists c)(Succ(c) \Rightarrow Pass) & \text{asa} & (\exists c)(\neg Succ(c) \vee Pass) \\ & \text{asa} & ((\exists c)\neg Succ(c)) \vee Pass \\ & \text{asa} & (\neg(\forall c)Succ(c)) \vee Pass \\ & \text{asa} & ((\forall c)Succ(c)) \Rightarrow Pass. \end{array}$$

Dit laat zien dat minstens één van de twee nederlandse zinnen verkeerd vertaald werd. De fout zit in de tweede logische zin $(\exists c)\ldots$. Deze is *geen* correcte vertaling van de nederlandse zin "er bestaat een vak waarvoor geldt: als je voor dat vak geslaagd bent, dan slaag je voor de bachelor". Het heeft te maken met het feit dat ons taalgevoel deze uitspraak interpreteert als een uitspraak over het examenreglement. Het examenreglement bepaalt voor elke mogelijke uitkomst van de examens of je al dan niet geslaagd bent voor de bachelor. De zin betekent dan: er bestaat een vak zodat *in elke volgens het examenreglement toegelaten wereld geldt dat als je er geslaagd bent voor dat vak, je er ook geslaagd ben voor de bachelor.* Zo'n uitspraak spreekt opnieuw over verschillende werelden, niet alleen over de enige echte wereld die we beleven. Daarom kan deze zin niet uitgedrukt worden in predikatenlogica. Er zijn wel meer complexe logica's met een alternative vorm van "als ... dan..." waarin deze zin wel kan uitgedrukt worden.

Oefening 2.7.1. *Neem het vocabularium $suc/2, pass/1$ waarbij $suc(x, y)$ betekent dat een student x slaagt voor vak y, en $pass(x)$ dat student x slaagt voor de bachelor. In de tweede zin willen we nu zeggen dat er een vak bestaat zodat elke student die geslaagd is voor dit vak, slaagt voor de bachelor. Druk de twee eigenschappen opnieuw uit in dit vocabularium en vergelijk: je zult merken dat de logische equivalentie verbroken is.*

2.8 Samenvatting

Waarheid van een logische zin is relatief: het hangt af van de structuur waarin de logische zin geïnterpreteerd wordt.

Definitie 2.2.4 is zonder meer de belangrijkste definitie van deze cursus. Zorg ervoor de je ze goed begrijpt.

In de inleiding hadden we gezegd dat een logica een taal is waarvan de syntactische vorm, maar ook de betekenis door een wiskundige theorie werd vastgelegd. Deze definitie legt de betekenis van formules op een wiskundige manier vast. Immers ze bepaalt wanneer twee zinnen dezelfde betekenis hebben, namelijk wanneer ze logische equivalent zijn asa ze waar zijn in exact dezelfde structuren.

Als je deze definities goed beheerst, dan ben je tot op zekere hoogte in staat om:

- te bewijzen dat een zin waar is in een structuur, of onwaar.

- een antwoord te construeren voor een query in een structuur. T.t.z. je kunt berekenen welke relatie wordt voorgesteld door een verzamelinguitdrukking $\{(x_1, \ldots, x_n) : A\}$.
- een structuur te construeren waarin een gegeven zin *waar* is.

Elk van deze opdrachten komt overeen met een type van frequent voorkomende taken in informatica-toepassingen. We zullen later nog andere vormen van redeneren zien die ook veel toepassingen hebben.

Hoofdstuk 3

Logische gevolgen en deductief redeneren

3.1 Logisch waar, logisch gevolg

Concepten zoals een gevolg, een implicatie, twee zinnen die equivalent zijn, een inconsistentie of een contradictie: het zijn eigenschappen van of relaties tussen "gedachten", "informaties", "proposities". In logica waar logische zinnen zo'n "gedachten" formeel uitdrukken, kunnen we deze concepten formeel definiëren gebruikmakend van het basisconcept van logica: de definitie van waarheid van een zin in een structuur. Op basis van dit concept kunnen een aantal andere fundamenteel belangrijke concepten in verband met informatie *formeel* gedefinieerd worden:

- een informatie/propositie kan een *tautologie* zijn, namelijk altijd voldaan in elke toestand van de wereld. Bv. er bestaat een les van dit vak dat niet doorgaat in de K06, ofwel gaat de volgende les door in de K06.
- één propositie kan een andere propositie *impliceren* of als *gevolg* hebben. Bv. als alle lessen van dit vak doorgaan in de K06, dan impliceert dit dat de volgende les doorgaat in K06.
- Een propositie kan *inconsistent* zijn, ook *tegenstrijdig* of *contradictorisch* genoemd. Bv. de propositie bestaande uit de voorgaande en dat de volgende les doorgaat in de N04, is inconsistent.
- Twee proposities kunnen logisch equivalent zijn.

Deze vertrouwde maar informele eigenschappen of relaties tussen "informaties" kunnen nu wiskundig geformaliseerd worden op basis van de definitie van waarheid in een structuur.

Definitie 3.1.1.

- Een logische formule A is **logisch waar**, of is een **tautologie** als ze waar is in alle structuren $\mathfrak{A}$ die A interpreteren. We noteren dit in een wiskundige notatie als $\models A$.
- Een logische formule A is **logisch consistent** als ze waar is in minstens één structuur die A interpreteert.

- Een logische formule A is **logisch inconsistent** of **logisch tegenstrijdig** of **logisch contradictorisch** als ze onwaar is in elke structuur die A interpreteert.

In het vervolg zullen we vaak het woord "logisch" weglaten van voor deze termen en spreken van gevolg, consistent, inconsistent, etc.

Let op het verschil tussen waarheid en logische waarheid. We kunnen niet van een logische zin zeggen dat hij waar is, tenzij we er een structuur bij geven die deze zin interpreteert. Maar we kunnen wel van zin zeggen dat hij logisch waar is zonder vermelding van een structuur.

Voorbeeld 3.1.1. De zin $(\forall x)(LesCur(x, LvI) \Rightarrow InAud(x, K06))$ is waar in de structuur van onze wereld waarbij $LesCur(x, y)$ betekent dat x een les is van cursus y. Maar deze zin is niet logisch waar.

De zin $((\forall x)(LesCur(x, LvI) \Rightarrow InAud(x, K06)) \wedge LesCur(L4, LvI)) \Rightarrow InAud(L4, K06)]$ is logisch waar, want waar in elke wereld, zelfs in die waar alle lessen van LvI doorgaan in N04, of zelfs waarin $LesCur$ iets totaal anders betekent dan lesgeven e.d., (Bv. waar $LesCur(x, y)$ betekent dat x minder curry bevat dan y).

De drie eigenschappen in de vorige definitie zijn natuurlijk zeer sterk gecorreleerd.

Propositie 3.1.1. *Een formule A is logisch waar asa $\neg A$ logisch inconsistent is asa $\neg A$ niet logisch consistent is.*

Bew. Een zin A is logisch waar asa A is waar in elke structuur asa $\neg A$ is onwaar in elke structuur (wat volgt uit de $\neg$-regel van Definitie 2.2.4) asa $\neg A$ is inconsistent asa $\neg A$ is niet consistent. Q.E.D.

Op het eerste zicht zou men misschien denken dat het onbegonnen werk is om van een formule aan te tonen dat deze waar is in *alle* structuren, want er zijn er zo oneindig veel, en bovendien zijn veel structuren zelf oneindig groot! Maar toch is dat vaak heel eenvoudig.

Voorbeeld 3.1.2. Nemen we een Nederlandse zin die evident waar is: "*Als alle lessen van dit vak plaats vinden in lokaal R1 of R2, en de volgende les van dit vak gaat niet door in lokaal R1, dan gaat ze door in lokaal 2* ". Deze zin is een instantiatie van een logische zin die logisch waar is:

T.B. $[(\forall x)(P(x, R1) \vee P(x, R2)) \wedge \neg P(C, R1)] \Rightarrow P(C, R2)$ is logisch waar.

We geven twee bewijzen, om twee vaak voorkomende bewijstechnieken te illustreren: bewijs uit het ongerijmde en bewijs door gevallenstudie. We noteren bovenstaande zin als A.

Bew. (door gevallenstudie) We bewijzen dat de uitspraak waar is in elke structuur van het vocabularium. Voor elke structuur $\mathfrak{A}$ geldt één van de volgende gevallen:

- Ofwel geldt $\mathfrak{A} \models P(C, R2)$ (lees: $P(C, R2)$ is waar in $\mathfrak{A}$). Dus voldoet $\mathfrak{A}$ aan de conclusie van A, en uit de $\Rightarrow$-regel volgt dat A waar is in $\mathfrak{A}$.
- Ofwel geldt $\mathfrak{A} \not\models P(C, R2)$. Nu kunnen we opnieuw een gevallenonderscheid maken:

- Ofwel geldt $\mathfrak{A} \models P(C, R1)$. In dit geval is de tweede conjunct van de premisse onwaar, en dus de premisse zelf ook niet. Uit de $\Rightarrow$-regel volgt dan dat de hele zin waar is in $\mathfrak{A}$.
- Ofwel geldt $\mathfrak{A} \not\models P(C, R1)$. In dit geval geldt voor $d = C^{\mathfrak{A}}$ dat $\mathfrak{A}[s : d] \not\models P(x, R1) \vee P(x, R2)$, en dus $\mathfrak{A} \not\models (\forall x)(P(x, R1) \vee P(x, R2))$ ($\forall$-regel). Dus is de premisse zelf ook niet voldaan ($\wedge$-regel). Ook nu volgt $\mathfrak{A} \models A$.

Er zijn geen andere gevallen meer, dus is het bewijs geleverd. Q.E.D.

Merk op dat we hierboven de oneindige klasse van alle structuren van het vocabularium van A hebben onderverdeeld in drie oneindige deelklassen: degene waarin $P(C, R2)$ waar is, degene waarin $P(C, R2)$ onwaar en $P(C, R1)$ waar is, en tenslotte, degene waarin $P(C, R2)$ en $P(C, R1)$ beide onwaar zijn. Elke structuur behoort tot exact één van deze subklassen. Hoewel alle structuren in een dergelijke deelklasse maar heel weinig met elkaar gemeen hebben, weten we er toch genoeg erover om te kunnen beslissen dat in elk de formule voldaan is. Zo kunnen we dus afleiden dat de formule voldaan is in elk van die oneindige klassen van structuren.

Bew. (uit het ongerijmde) Veronderstel dat deze zin A niet logisch waar is. Dan bestaat er een structuur $\mathfrak{A}$ waarin de zin onwaar is. Het volgt uit de $\Rightarrow$-regel dat daarin de premisse waar en de conclusie onwaar is. Als we de premisse met de $\wedge$-regel en de $\neg$-regel vereenvoudigen, krijgen we:

$$(1)\ \mathfrak{A} \models (\forall x)(P(x, R1) \vee P(x, R2)) \quad \text{en} \quad (2)\ \mathfrak{A} \not\models P(C, R1) \quad \text{en} \quad (3)\ \mathfrak{A} \not\models P(C, R2)$$

Uit de $\forall$-regel volgt dat voor elk element d uit het domein van $\mathfrak{A}$ het volgende geldt: $\mathfrak{A}[x : d] \models P(x, R1) \vee P(x, R2)$. Bijgevolg geldt ook $\mathfrak{A}[x : C^{\mathfrak{A}}] \models P(x, R1) \vee P(x, R2)$. Volgens de $\vee$-regel volgt dan dat er twee mogelijke gevallen zijn:

- Ofwel geldt $\mathfrak{A}[x : C^{\mathfrak{A}}] \models P(x, R1)$, wat in tegenspraak is met (2).
- Ofwel geldt $\mathfrak{A}[x : C^{\mathfrak{A}}] \models P(x, R2)$, wat in tegenspraak is met (3).

Er zijn geen andere mogelijkheden, dus bekomen we een contradictie. Q.E.D.

Opmerking 3.1.1. Het bewijs door gevallenonderscheid is een rechtstreeks bewijs: het werkt door de oneindige klasse van structuren op te splitsen in een beperkt aantal nog steeds oneindige deelklassen, zodat je voor elke deelklasse toch voldoende informatie hebt om te kunnen aantonen dat A waar is in elk element ervan. Cruciaal hier is hoe die subklassen te bepalen.

Het bewijs uit het ongerijmde lijkt anderzijds wel eleganter en doelgerichter dan een bewijs door gevallenonderscheid, ook al omdat je mag redeneren over één structuur. In het vervolg kiezen we dus deze manier van bewijzen. M.a.w., om te bewijzen dat een zin A logisch waar is, veronderstel dat er een structuur $\mathfrak{A}$ bestaat zodat $\mathfrak{A} \not\models A$ en leidt daaruit een contradictie af.

Overigens, elk bewijs uit het ongerijmde kan geherformuleerd worden als een bewijs door gevallenonderscheid en omgekeerd. Er is dus geen verschil in *bewijskracht* tussen deze methodes.

Analoog kun je bewijzen dat een formule A contradictorisch is door een gevallenonderscheid te maken en zo aan te tonen dat A onwaar is in elke structuur, maar een bewijs uit het ongerijmde is meestal gemakkelijker.

Tenslotte, de enige manier om te bewijzen dat een formule consistent is, is een model te construeren, t.t.z. een structuur waarin de formule waar is. Hoe dit te doen bespraken we al in het vorig hoofdstuk.

Oefening 3.1.1. *De formule* $(\forall x)P(x,y) \Rightarrow (\exists x)P(x,y)$ *is logisch waar. Om dit in te zien moet je gebruik maken van het feit dat het universum van een structuur minstens één element bevat (per definitie* $D_{\mathfrak{A}} \neq \emptyset$*).*

Oefening 3.1.2. *De formule* $(\forall x_1)\ldots(\forall x_n)(\exists y)y = F(x_1,\ldots,x_n)$ *is logisch waar. Om dit te bewijzen moet je gebruik maken van het feit in elke structuur* $F^{\mathfrak{A}}$ *een totale functie is en dus dat voor elk koppel* $(d_1,\ldots,d_n)$*,* $F^{\mathfrak{A}}(d_1,\ldots,d_n)$ *een bestaand element is van het domein.*

We kunnen in principe dus geen partiële functie modelleren met een functiesymbool.

Voorbeeld 3.1.3. De zin $(\exists x)(\forall y)y = x$ is niet logisch waar. Inderdaad, deze zin is waar in een structuur $\mathfrak{A}$ asa het universum van $\mathfrak{A}$ slechts één element bevat. Maar dus is deze zin wel consistent.

Dus ook de zin $\neg(\exists x)(\forall y)y = x$ is niet logisch waar maar wel consistent want het drukt uit dat het domein minstens 2 elementen bevat.

Dit laatste voorbeeld laat zien dat $\not\models A$ niet impliceert dat $\models \neg A$. Merk op dat daarentegen $\mathfrak{A} \not\models A$ wel impliceert dat $\mathfrak{A} \models \neg A$. Dit volgt namelijk uit de $\neg$-regel. Dus, logische waarheid en gewone waarheid hebben andere eigenschappen!

Definitie 3.1.2. Een logische formule A is een **logisch gevolg** van B als in elke structuur $\mathfrak{A}$ die beide formules interpreteert en waarin B waar is, ook A waar is. De wiskundige notatie om dit aan te geven is $B \models A$.

Of in wiskundige notatie, A is een logisch gevolg van B indien voor alle structuren $\mathfrak{A}$ die A en B te interpreteren en waarin $\mathfrak{A} \models B$ geldt dat $\mathfrak{A} \models A$. Wanneer $B \models A$ dan zeggen we ook wel: B **impliceert** A.

Opmerking 3.1.2. Het wiskundige teken $\models$ wordt ondertussen al in drie verschillende betekenissen gebruikt wordt:

- $\mathfrak{A} \models A$ betekent "formule A is waar in structuur $\mathfrak{A}$".
- $\models A$ betekent "formule A is logisch waar".
- $B \models A$ betekent "formule A is een logisch gevolg van formule B".

Logici zijn zuinig met hun tekens! De betekenis van het symbool moet je halen uit de context. Wees zorgvuldig.

Propositie 3.1.2.

(a) $B \models A$ *asa* $\models B \Rightarrow A$. *In woorden,* A *is een logisch gevolg van* B *asa de zin* $B \Rightarrow A$ *logisch waar is.*

(b) A *en* B *zijn logisch equivalent asa* $A \Leftrightarrow B$ *logisch waar is.*

Bew. We bewijzen enkel (a).

(⇒) Veronderstel dat $B \models A$. Neem een willekeurige structuur $\mathfrak{A}$ die B en A interpreteert. Ofwel geldt $\mathfrak{A} \not\models B$, en dan volgt uit de $\Rightarrow$-regel dat $\mathfrak{A} \models B \Rightarrow A$. Ofwel geldt $\mathfrak{A} \models B$. Omdat A een logisch gevolg is van B geldt ook $\mathfrak{A} \models A$, en toepassing van de $\Rightarrow$-regel geeft opnieuw $\mathfrak{A} \models B \Rightarrow A$. Dus geldt $\models B \Rightarrow A$.

(⇐) Veronderstel dat $\models B \Rightarrow A$, dus voor elke $\mathfrak{A}$ geldt $\mathfrak{A} \models B \Rightarrow A$. Uit de $\Rightarrow$-regel volgt dat als $\mathfrak{A} \models B$ dan $\mathfrak{A} \models A$. Bijgevolg geldt $B \models A$.

Q.E.D.

Definitie 3.1.3.

- Een zin A is **logisch inconsistent** of **logisch tegenstrijdig** of **logisch contradictorisch** met B indien A onwaar is in elk model van B dat ook A interpreteert.
- Een zin A is **logisch consistent** met zin B indien A voldaan is in minstens één model van B.
- Een zin A is **logisch equivalent** met zin B indien voor elke $\mathfrak{A}$, $A^{\mathfrak{A}} = B^{\mathfrak{A}}$ (ze hebben zelfde waarheidswaarde).

Dezelfde termen worden hier opnieuw gebruikt in een andere betekenis dan voorheen, namelijk om relaties tussen twee zinnen aan te duiden. Er is dan ook een sterk verband tussen bv. het logisch inconsistent zijn van een formule, en het logisch inconsistent zijn van een formule met een andere formule, zoals de volgende propositie aantoont.

Propositie 3.1.3. *A is logisch inconsistent met B asa $A \wedge B$ is logisch inconsistent. A is logisch consistent met B asa $A \wedge B$ is logisch consistent.*

Oefening 3.1.3. *Bewijs deze propositie. (Eenvoudig!)*

Opmerking 3.1.3. Een theorie $T = \{A_1, \dots A_n\}$ betekent hetzelfde als de fomule $A_1 \wedge \cdots \wedge A_n$. Bijgevolg kunnen we elk van de definities hierboven (gevolg, consistent, inconsistent) op evidente manier uitbreiden tot theorieën:

- een logische theorie T is logisch waar als elke zin in T logisch waar is.
- Een logische theorie T is inconsistent als er in elke structuur minstens één zin van T onwaar is.
- Twee logische theorie T, T' zijn consistent met elkaar als er een model bestaat van beide, en inconsistent met elkaar als er geen structuur bestaat die een model is van beide theorieën.

Opmerking 3.1.4. Een query is een symbolisch relatievoorschrift wat in deze cursus noteren als $\{(x_1, \dots, x_n) : A\}$. In Hoofdstuk 1 definieerden we dat twee queries $\{(x_1, \dots, x_n) : A\}$ en $\{(x_1, \dots, x_n) : B\}$ logisch equivalent zijn als ze dezelfde interpretatie hebben in elke structuur. Dat is het geval asa A en B logisch equivalent zijn.

Maar zoals we zagen, zijn niet alle *correcte* vertalingen van een informele query logisch equivalent. Bv. omdat geen enkel vak voorkennis van zichzelf kan zijn, zijn de queries $\{x : Prerequ(x, CS230)\}$ en $\{x : Prerequ(x, CS230) \wedge x \neq CS230\}$ beide correcte vertalingen van de query naar alle vakken die voorkennis zijn van CS230. Deze twee queries voldoen aan een zwakkere vorm van equivalentie die we als volgt kunnen definiëren: ze zijn equivalent modulo de theorie van alle integriteitsbeperkingen.

Definitie 3.1.4. Twee formules A, B zijn **logisch equivalent modulo** een theorie T indien geldt dat voor elk model $\mathfrak{A}$ van T dat A en B interpreteert, geldt dat: $\mathfrak{A} \models A$ asa $\mathfrak{A} \models B$.

Oefening 3.1.4. *Veronderstel dat T bestaat uit één zin C. Bewijs dat A, B equivalent zijn modulo C asa $C \wedge A, C \wedge B$ logisch equivalent zijn.*

3.2 Propositielogica

We zullen zien in Sectie 3.5 dat er in het algemeen geen eindigende automatische implementeerbare methode bestaat om te bewijzen dat een zin een logisch gevolg of equivalent of inconsistent is met een theorie. Maar er is een deellogica van predikatenlogica waarvoor dit wel mogelijk is, namelijk propositielogica.

Historisch gezien werd propositielogica ontwikkeld vóór de predikatenlogica, namelijk door Boole (1854). Propositielogica wordt daarom, in de electronica, ook vaak Boolse algebra genoemd. De titel van Boole's werk was *"An Investigation of the Laws of Thought"*, wat illustratief is voor de bedoeling van Boole en andere vroege logici om "gedachten" te bestuderen met wiskundige methodes. Dit is een bedoeling die wellicht weinig electronici nog zoeken in Boolse algebra, maar die wij in deze cursus zeker nastreven omdat we willen leren hoe informatie voor te stellen in logica.

Definitie 3.2.1. Een propositioneel symbool is een 0-voudig predikaatsymbool. Een propositioneel vocabularium Σ is een vocabularium dat enkel bestaat uit propositionele symbolen. Een propositionele formule is een formule opgebouwd uit propositionele symbolen en de logische connectieven $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$.

Een propositionele formule bevat dus geen kwantoren of variabelen of $=$ en is bijgevolg een logische zin. Een voorbeeld is $P \wedge (P \Rightarrow Q)$.

Een propositioneel symbool is een 0-voudig relatiesymbool. Maar wat is een 0-voudige relatie? Een verzameling van 0-voudige koppels? Er is maar één 0-voudig koppel, namelijk (), en dus zijn er maar twee 0-voudige relaties: de lege verzameling $\emptyset$ en $\{()\}$. De lege verzameling $\emptyset$ kunnen we identificeren met $\mathbf{f}$ en $\{()\}$ met $\mathbf{t}$. Bijgevolg drukt een propositioneel symbool een eigenschap uit. Het is allereenvoudigste soort logische zin. Bv. veronderstel dat de volgende betekenis toekennen aan P, Q, R: P wordt geïnterpreteerd als "De trein is laat", Q als "John mist zijn bus", dan zegt de zin $P \wedge (P \Rightarrow Q)$ dat de trein te laat is en dat als de trein te laat is dat John dan zijn bus mist.

Merk ook het verband op met relatievoorschriften. De interpretatie van een n-voudig relatievoorschrift $\{(x_1, \ldots, x_n) : A\}$ in een structuur is een n-voudige relatie. Wanneer $n = 0$, dan is de interpretatie van het relatievoorschrift $\{()\}$ wanneer A waar is in de structuur, of de lege verzameling $\emptyset$ wanneer A onwaar is. Nu we $\{()\}$ identificeren met $\mathbf{t}$ and $\emptyset$ met $\mathbf{f}$ betekent dit dat A en $\{() : A\}$ dezelfde interpretatie hebben! Dus elke zin A die we als query willen stellen aan een databank kan ook geformuleerd worden als de 0-voudige query $\{() : A\}$. Op die manier zien we dat het stellen van n-voudige queries een veralgemening is van het stellen van zinnen als 0-voudige queries.

In de context van een propositioneel vocabularium en theorie zijn gekwantificeerde zinnen zinloos. Ook is het domein van een structuur volkomen irrelevant. We kunnen bijgevolg de notie van structuur vereenvoudigen, zoals in de volgende definitie gebeurt.

Definitie 3.2.2. Een structuur van een propositioneel vocabularium Σ is een functie van Σ naar $\{\mathbf{t}, \mathbf{f}\}$. Voor $P \in \Sigma$ zullen we $\mathfrak{A}(P)$ blijven noteren als $P^{\mathfrak{A}}$.

Als het duidelijk is wat Σ is wordt een structuur van Σ ook wel genoteerd als de verzameling van ware symbolen. Dus voor $\Sigma = \{P, Q\}$ denoteert $\{P\}$ de structuur waarin P waar is en Q onwaar.

Opmerking 3.2.1. Het is echt van belang dat Σ vastligt als je die notatie gebruikt, anders is ze ambigu. Bv. $\{P\}$ noteert telkens een verschillende structuur indien $\Sigma = \{P\}$, $\Sigma = \{P, Q\}$, $\Sigma = \{P, Q, R\}$, etc.

Definitie 2.2.4 van waarheid blijft (bijna) onveranderd gelden, maar de regels voor $=$, $\exists$ en $\forall$ zijn overbodig. Van de resterende regels hoeft enkel de atoom-regel een minimale aanpassing. Deze wordt: $\mathfrak{A} \models P$ asa $P^{\mathfrak{A}} = \mathbf{t}$. De concepten van logische waarheid, logisch gevolg, logisch consistent, logisch inconsistent blijven onveranderd gelden.

Voorbeeld 3.2.1 (Een logische puzzel)**.** *Er zijn 3 dozen 1, 2 en 3. Eén ervan bevat goud, de anderen zijn leeg. Er zijn 3 boodschappen. Eén is waar, de anderen zijn niet waar.*

- *Boodschap 1: Het goud zit niet in doos 1*
- *Boodschap 2: Het goud zit niet in doos 2*
- *Boodschap 3: Het goud zit in doos 2*

Waar is het goud?

We maken een theorie om deze informatie uit te drukken.

- We maken een propositioneel vocabularium Σ:
  - H_i: "boodschap i is waar"
  - G_i: "het goud zit in doos i"
- Dat er maar één boodschap waar is, drukken we als volgt uit:

$$(H_1 \wedge \neg H_2 \wedge \neg H_3) \vee (\neg H_1 \wedge H_2 \wedge \neg H_3) \vee (\neg H_1 \wedge \neg H_2 \wedge H_3)$$

- Dat er in juist één doos goud zit, drukken we als volgt uit:

$$(G_1 \wedge \neg G_2 \wedge \neg G_3) \vee (\neg G_1 \wedge G_2 \wedge \neg G_3) \vee (\neg G_1 \wedge \neg G_2 \wedge G_3)$$

- De betekenis van de boodschappen: $H_1 \Leftrightarrow \neg G_1$, $H_2 \Leftrightarrow \neg G_2$, $H_3 \Leftrightarrow G_2$.

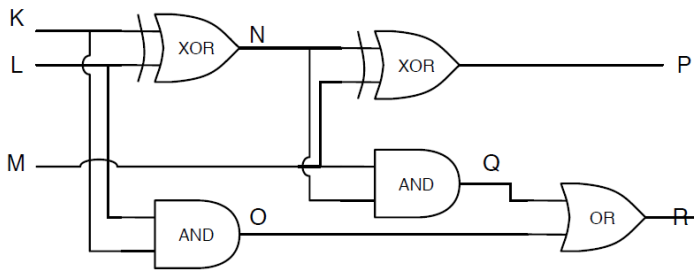
Nu we een theorie hebben kunnen we ons de vraag stellen hoe we deze kunnen gebruiken om de puzzel op te lossen. De oplossing (of eventueel meerdere oplossingen) zit verscholen in de modellen van deze theorie. Elk model $\mathfrak{A}$ is een situatie die consistent is met de puzzel, en zal één doos aanwijzen waar het goud in zit, namelijk als $G_i^{\mathfrak{A}}$ waar is. Er blijkt echter maar één model te zijn.

Bijgevolg, als we deze puzzel willen laten oplossen door een computer, dan hebben we een tool nodig dat het volgende berekent:

Bereken een model van een (propositionele) theorie.

Dit is een probleem met zeer veel toepassingen in de informatica (zie verderop).

Oefening 3.2.1. *Wat zijn de modellen en de corresponderende oplossingen? Je kunt de puzzel oplossen door een* waarheidstabel *op te stellen. (Zie verderop).*



Figuur 3.1: Een digitaal schakelnetwerk

Schakelnetwerken. Een electronisch schakelnetwerk bestaat uit poorten en schakelaars. Zo'n netwerk heeft een aantal invoerpoorten waar een spanning 0 of 1 (of **f** of **t**) kan aangelegd worden en een aantal uitvoerpoorten. Tussenin liggen de schakelaars die logische operaties toepassen op hun input.

Een voorbeeld van een booleaans circuit is te vinden in Figuur 3.1. In dit netwerk zijn K, L, M invoer- en P en R uitvoerpoorten. Drie soorten logische operaties worden gebruikt: $\wedge, \vee$ en `XOR` , een nieuwe logische operator met de volgende waarheidstabel:

A	B	A `XOR` B
t	**t**	**f**
t	**f**	**t**
f	**t**	**t**
f	**f**	**f**

Het is makkelijk in te zien dat A `XOR` B logisch equivalent is met $(A \vee B) \wedge \neg(A \wedge B)$. Dit wordt ook wel de exclusieve disjunctie genoemd, integenstelling tot $\vee$ die de inclusieve disjunctie genoemd wordt.

De uitvoerpoort P komt overeen met de formule die je bekomt door vanuit P alle paden te volgen tot aan de invoerpoorten: dit geeft (K `XOR` L) `XOR` M. Het signaal aan de uitvoerpoort P komt overeen met de waarheidswaarde van deze formule in de structuur $\mathfrak{A}$ van $\{K, L, M\}$ waarbij $K^{\mathfrak{A}}, L^{\mathfrak{A}}, M^{\mathfrak{A}}$ overeenkomt met de invoer die opgelegd wordt aan de invoerpoorten. Op die manier komt elke propositionele formule dus overeen met een schakelnetwerk en omgekeerd.

Bij het ontwerp van electronische schakelnetwerken zijn de volgende vormen van redeneren van cruciaal belang. Veronderstel dat we een bepaalde logische zin willen implementeren met een schakelnetwerk. Elke logisch equivalente zin realiseert dezelfde uitgangspoort, maar met een ander schakelnetwerk. Men heeft hier dus veel keuze. In elk geval kun je zien dat de volgende vorm van redeneren van belang is om na te gaan of een gekozen netwerk de gewenste logische zin realiseert:

bereken of propositionele formules A en B logisch equivalent zijn.

Dus, gegeven een gewenste logische formule A, en een voorstel voor een netwerk dat B implementeert, dan bestaat de *verificatie* van het netwerk erin om na te gaan of B logisch equivalent is met A.

Een veel ambitieuzer redeneertaak is de volgende:

Voor een gegeven propositionele formule A, bereken een propositionele formule B die logisch equivalent is aan A en zo weinig mogelijk logische poorten bevat.

Dat betekent dat de digitale netwerk die B realiseert kleiner is dan A maar toch dezelfde uitvoer genereert. Het netwerk van B is kleiner, goedkoper, minder energieverbruikend dan dat van A. Het berekenen van zo'n minimale formule is een zeer complex zoekprobleem - practisch onoplosbaar voor grote formules.

Logisch ware formules berekenen met waarheidstabellen. Elke propositionele zin wordt geïnterpreteerd door oneindig veel structuren. Maar elke zin A bevat slechts een eindig aantal niet-logische symbolen Σ_A. Er zijn slechts eindig veel structuren van Σ_A. Of A al dan niet waar is in een structuur $\mathfrak{A}$ hangt enkel af van de waardes van de symbolen van Σ_A. Dit volgt uit de volgende eigenschap.

Eigenschap 3.2.1. Als $\mathfrak{A}$ een zin A interpreteert, en $\mathfrak{A}'$ is de structuur bekomen door $\mathfrak{A}$ te beperken tot Σ_A, de niet-logische symbolen van A, dan geldt dat $\mathfrak{A} \models A$ asa $\mathfrak{A}' \models A$.

Dat betekent dat we kunnen berekenen of een propositionele zin A logisch waar of consistent is door gewoon alle structuren van Σ_A op te sommen en daarin de waarheidswaarde van A te berekenen. Een systematische manier om dat te doen is met behulp van een *waarheidstabel*. We illustreren dit met een voorbeeld.

Zij P, Q, R propositionele symbolen. Dan is

$$[(P \vee Q) \wedge R] \Leftrightarrow [(P \wedge R) \vee (Q \wedge R)]$$

een tautologie. We verifiëren dit door de waarheidstabel op te stellen. In deze tabel gebruiken we 0 voor **f** en 1 voor **t**.

P	Q	R	$(P \vee Q)$	$(P \vee Q) \wedge R$	$P \wedge R$	$Q \wedge R$	$(P \wedge R) \vee (Q \wedge R)$	$(P \vee Q) \wedge R \Leftrightarrow (P \wedge R) \vee (Q \wedge R)$
1	1	1	1	1	1	1	1	1
1	1	0	1	0	0	0	0	1
1	0	1	1	1	1	0	1	1
1	0	0	1	0	0	0	0	1
0	1	1	1	1	0	1	1	1
0	1	0	1	0	0	0	0	1
0	0	1	0	0	0	0	0	1
0	0	0	0	0	0	0	0	1

In deze tabel gaan we alle structuren af van de propositionele symbolen van de formule. Strikt genomen hebben we natuurlijk niet aangetoond dat alle structuren die deze zin interpreteren de zin ook waar maken. Immers, een structuur kan ook veel meer symbolen interpreteren dan alleen maar P, Q, R. Maar het volstaat om deze 8 structuren te controleren.

Er is ook een compactere manier om een waarheidstabel op te schrijven die we illustreren met een ander voorbeeld namelijk $(P \texttt{ XOR } Q) \Leftrightarrow ((P \vee Q) \wedge \neg(P \wedge Q))$:

P	Q	(	P	XOR	Q	)	$\Leftrightarrow$	((	P	$\vee$	Q	)	$\wedge$	$\neg$	(	P	$\wedge$	Q	))
1	1		1	0	1		1		1	1	1		0	0		1	1	1	
1	0		1	1	0		1		1	1	0		1	1		1	0	0	
0	1		0	1	1		1		0	1	1		1	1		0	0	1	
1	1		0	0	0		1		0	0	0		0	1		0	0	0	

Merk op dat onder $\Leftrightarrow$ alleen maar 1 voorkomt, dus is deze formule logisch waar.

We kunnen met waarheidstabellen nog andere essentiële vragen oplossen. We zagen al dat een formule A een logisch gevolg is van B asa $B \Rightarrow A$ logisch waar is; A, B zijn logisch equivalent asa $A \Leftrightarrow B$ logisch waar is; A is inconsistent met B asa $\neg(A \wedge B)$ logisch waar is. Dus het bepalen van logisch gevolg, equivalentie of inconsistentie kunnen we met waarheidstabellen oplossen.

Oefening 3.2.2. *Bewijs dat P `XOR` Q logisch equivalent is met $(P \Leftrightarrow \neg Q)$.*

Oefening 3.2.3. *In het vorig hoofdstuk werd beweerd dat de formule $P \wedge (\neg P \Rightarrow Q)$ equivalent is aan P. Bewijs dit met een waarheidstabel.*

Deze methode werkt goed, maar enkel voor kleine formules. Immers als A n propositionele symbolen bevat, dan is het aantal structuren die nagegaan moet worden 2^n, namelijk voor elke combinatie van 0 of 1 van elk symbool. Dit betekent dat de grootte van de waarheidstabel exponentieel is in het aantal symbolen. Voor een formule met 3 symbolen moet je 8 structuren testen, met 10 symbolen moet je al 1024 structuren testen; voor 100 symbolen is het aantal te testen structuren een getal met 30 cijfers.

Je kunt je indenken dat in de huidige processoren het aantal invoerpoorten veel groter is dan 100. Tegenwoordig voeren computers operaties uit op 64-bit getallen. Alleen al om de som van twee registers te berekenen zijn al 124 invoerbits nodig. Elke invoerbit komt overeen met een propositioneel symbool! Dat betekent dat het onbegonnen werk is om de correctheid van een digitale processor te berekenen met behulp van een waarheidstabel.

Gelukkig bestaan er veel efficiëntere methodes om de logische waarheid of de consistentie van een propositionele zin te berekenen. Dergelijke methodes worden ontwikkeld in het domein van *satisfiability checking*, kortweg SAT. SAT-systemen worden toegepast bij verificatie van digitale schakelnetwerken, maar ook steeds meer en steeds verder daarbuiten. Het blijkt immers dat zeer veel informaticaproblemen kunnen opgelost worden door ze te reduceren tot een SAT probleem. Bv. niet alleen verificatieproblemen bij verificatie van schakelnetwerken, maar ook puzzels, uurroosters, planning, diagnose, verificatie van software, simulatie, enz. SAT solvers worden ook gebruikt in LogicPalet. Wanneer je vraagt een GEO-world of een Deca-world te construeren waarin een logische zin voldaan is, roept LogicPalet het IDP-systeem op, dat op zijn beurt een SAT solver oproept.

Toch is het zo dat ook SAT solvers voor sommige problemen zeer inefficiënt zijn. Een openstaande wetenschappelijke vraag is of het wel mogelijk is om efficiënte algoritmes te ontwikkelen om de consistentie of logische waarheid van een booleaanse formule te bewijzen. Dit probleem wordt op dit ogenblik beschouwd als 1 van de 10 belangrijkste wiskundeproblemen! Het probleem heet het P=NP probleem.

Ter illustratie, een amerikaans instituut, het Clay mathematical institute, looft een prijs van **één miljoen dollar** uit voor wie dit probleem kan oplossen! Hetzelde geldt voor het vermoeden van Goldbach dat we in Hoofdstuk 2 tegen kwamen. Wie voelt zich geroepen?

Tenslotte zagen we al verschillende belangrijke redeneerproblemen voor propositionele logica:

logische waarheid, consistentie, berekenen van minimale equivalente formules. Er zijn nog heel wat andere vormen van redeneren die eveneens veel toepassingen hebben.

Bereken het aantal modellen van een propositionele zin.

Bereken een structuur die zoveel mogelijk zinnen voldoet in een verzameling van propositionele zinnen.

Er zijn er nog heel wat andere waar we hier niet op ingaan, maar die bv. optreden wanneer we willen bepalen welke fout er in een digitaal netwerk zit, of welke invoerpoorten geen effect kunnen hebben op de uitvoer, enz.

3.3 Laws of thought

Als we weten dat niet alle objecten eigenschap X hebben, betekent dat hetzelfde als dat er een object is dat de eigenschap X *niet* heeft? Als het niet waar is dat eigenschappen X en Y gelden, betekent dat hetzelfde als dat eigenschap X niet geldt of dat eigenschap Y niet geldt? Ons taalgevoel vertelt ons dat dit zo is. Dit waren het soort **wetten van het denken** dat mensen zoals Aristoteles, Leibniz, Boole, De Morgan, Frege wilden onderzoeken.

We zullen nu een aantal fundamentele equivalenties zien die zo'n "wetten van het denken" uitdrukken, en die eenvoudig volgen uit de definities van waarheid en logische waarheid. We kunnen deze intuïtieve wetten dus bewijzen!

Fundamentele equivalenties van de connectieven.

Eigenschap 3.3.1 (basiswetten van $\wedge$ and $\vee$)**.** Zij P, Q, R propositionele symbolen, dan zijn de volgende propositionele zinnen tautologieën.

1. $(P \vee Q) \Leftrightarrow (Q \vee P)$ (commutativiteit van $\wedge$ and $\vee$)
   $(P \wedge Q) \Leftrightarrow (Q \wedge P)$
2. $[(P \vee Q) \vee R] \Leftrightarrow [P \vee (Q \vee R)]$ (associativiteit van $\wedge$ and $\vee$)
   $[(P \wedge Q) \wedge R] \Leftrightarrow [P \wedge (Q \wedge R)]$
3. $[P \wedge (Q \vee R)] \Leftrightarrow [(P \wedge Q) \vee (P \wedge R)]$ (distributiviteit van $\wedge$ t.o.v. $\vee$)
   $[P \vee (Q \wedge R)] \Leftrightarrow [(P \vee Q) \wedge (P \vee R)]$ (distributiviteit van $\vee$ t.o.v. $\wedge$)
4. $(P \vee P) \Leftrightarrow P$ (idempotentie)
   $(P \wedge P) \Leftrightarrow P$

Opmerking 3.3.1. Vergelijk volgende Nederlandse zinnen:

"hij laadde zijn geweer en schoot" en "Hij schoot en laadde zijn geweer"

Ons taalgevoel zegt ons dat in beide de volgorde van de acties verschilt, dus is "en" hier niet commutatief. Dit laat zien dat wij mensen ook het woord "en" verschillend zullen interpreteren al naar gelang de context. Een "en" van gebeurtenissen interpreteren we meestal als dat de

eerst vermelde gebeurtenis ook het eerst gebeurt in de tijd, zeker als de beide gebeurtenissen niet gelijktijdig kunnen plaatsvinden. Daarentegen, de "en" in de zin "$\mathfrak{A} \models A \wedge B$ asa $\mathfrak{A} \models A$ en $\mathfrak{A} \models B$" in Definitie 2.2.4 waar we vastleggen wat $\wedge$ betekent heeft niet die temporele betekenis, en dat geldt voor de meest voorkomens van "en" in deze tekst.

Eigenschap 3.3.2 (basiswetten van $\neg$)**.** Zij P, Q propositionele symbolen, dan zijn de volgende propositionele zinnen tautologieën.

1. $\neg\neg P \Leftrightarrow P$ (dubbele ontkenning)
2. $P \vee \neg P$ (uitgesloten derde)
3. $\neg(P \vee Q) \Leftrightarrow (\neg P \wedge \neg Q)$ (wetten van De Morgan[1])
   $\neg(P \wedge Q) \Leftrightarrow (\neg P \vee \neg Q)$
4. $\neg(P \Rightarrow Q) \Leftrightarrow (P \wedge \neg Q)$ (negatie van de implicatie)

Eigenschap 3.3.3 (De betekenis van $\Rightarrow, \Leftrightarrow$)**.** Zij P, Q propositionele symbolen, dan zijn de volgende propositionele zinnen tautologieën.

1. $(P \Rightarrow Q) \Leftrightarrow (\neg P \vee Q)$ ($\Rightarrow$ in termen van $\neg, \vee$)
2. $(P \Leftrightarrow Q) \Leftrightarrow [(P \Rightarrow Q) \wedge (Q \Rightarrow P)]$ ($\Leftrightarrow$ in termen van implicaties)
3. $(P \Leftrightarrow Q) \Leftrightarrow [(P \wedge Q) \vee (\neg P \wedge \neg Q)]$ ($\Leftrightarrow$ in termen van $\wedge$ en $\vee$)

Eigenschap 3.3.4 (Eigenschappen van $\Rightarrow, \Leftrightarrow$)**.** Zij P, Q, R, S propositionele symbolen, dan zijn de volgende propositionele zinnen tautologieën.

1. $(P \Rightarrow Q) \Leftrightarrow (\neg Q \Rightarrow \neg P)$ (contrapositie bij $\Rightarrow$)
2. $[(P \Rightarrow Q) \wedge (Q \Rightarrow R)] \Rightarrow (P \Rightarrow R)$ (transitiviteit van $\Rightarrow$)
3. $(P \Leftrightarrow Q) \Leftrightarrow (Q \Leftrightarrow P)$ (commutativiteit $\Leftrightarrow$)
4. $(P \Leftrightarrow Q) \Leftrightarrow (\neg P \Leftrightarrow \neg Q)$ (contrapositie bij $\Leftrightarrow$)
5. $[(P \Leftrightarrow Q) \wedge (Q \Leftrightarrow R)] \Rightarrow (P \Leftrightarrow R)$ (transitiviteit van $\Leftrightarrow$)
6. $[P \Rightarrow (Q \Rightarrow R)] \Leftrightarrow [(P \wedge Q) \Rightarrow R]$
   $[P \Rightarrow (Q \Rightarrow (R \Rightarrow S))] \Leftrightarrow [(P \wedge Q \wedge R) \Rightarrow S]$
   enzovoort.

Opmerking 3.3.2. Zoals gezegd heeft "als ... dan..." veel betekenissen. In vele andere betekenissen dan de materiele implicatie, geldt contrapositie niet. Vergelijk volgende Nederlandse zinnen:

"Als het morgen mooi weer is dan ga ik naar zee" en "Als ik morgen niet naar zee ga, dan is het geen goed weer."

[1] De Morgan was een Engelse logicus die zijn resultaten ongeveer tergelijkertijd als Boole publiceerde (1847) maar niet in een wiskundige vorm zoals Boole.

Zijn ze equivalent? Volgens mijn taalgevoel niet. In de eerste zin gebruik ik een betekenis van "als ... dan ... " die we nog niet eerder gezien hebben: namelijk, het drukt uit dat ik de *intentie* heb om, als het morgen mooi weer is, naar zee te gaan. De tweede zin, zijn contrapositie, drukt geen intentie uit: ik heb niet de intentie om, als ik morgen niet aan zee zit, het geen goed weer te laten zijn.

Het is een kenmerk van "echte" materiele implicaties dat de contrapositie ervan wel geldig is. Vergelijk "als alle lessen door gaan in K06, dan gaat de volgende les door in K06" met "als de volgende les niet doorgaat in K06, dan gaan niet alle lessen door in K06". Deze zinnen zijn onmiskenbaar equivalent. Daar kan niemand onderuit.

Eigenschap 3.3.5 (Bewijsprincipes)**.** Zij P, Q propositionele symbolen, dan zijn de volgende propositionele zinnen tautologieën.

1. $(P \wedge \neg P) \Rightarrow Q$ (uit een contradictie volgt alles)
2. $[\neg P \Rightarrow (Q \wedge \neg Q)] \Rightarrow P$ (bewijs uit het ongerijmde)
3. $P \Leftrightarrow [(Q \Rightarrow P) \wedge (\neg Q \Rightarrow P)]$ (bewijs door gevallenonderscheid)

Waarom noemen we dit bewijsprincipes? Omdat ze een schema vormen voor hoe bepaalde eigenschappen te bewijzen. In voorbeeld 3.1.2 bewezen we voor de daar gespecifieerde formule A dat $\mathfrak{A} \models P(C, R2)$ impliceert dat $\mathfrak{A} \models A$ maar dat ook $\mathfrak{A} \not\models P(C, R2)$ impliceert dat $\mathfrak{A} \models A$. Dat bewijs kun je zien als een toepassing van de hierboven vermelde regel van bewijs door gevallenonderscheid. Ook het tweede gedeelte van het bewijs, namelijk dat $\mathfrak{A} \not\models P(C, R2)$ impliceert dat $\mathfrak{A} \models A$ werd bewezen door een tweede toepassing van hetzelfde principe.

Bew. van Eigenschappen 3.3.1-3.3.5. Elk van de equivalenties in deze eigenschappen kan gemakkelijk bewezen worden met waarheidstabellen.

Oefening 3.3.1. *Elk van deze equivalenties kan gemakkelijk bewezen worden met waarheidstabellen. Pik er een paar uit en verifieer.*

Oefening 3.3.2. *Sta bij elk van deze equivalenties even stil. Elk ervan drukt een eigenschap uit van Nederlandse voegwoorden "en", "of", en zinnen "als ... dan ... " en "... als en slechts als ... ". Vraag je telkens af of je het ermee eens bent.*

Generaliseren van logische waarheden. Strikt genomen hebben we tot nog toe ongeveer een 25-tal logische waarheden gezien. Op de oneindig veel formules die er bestaan is dat niet vet. Maar je voelt intuïtief aan dat elk van deze zinnen staat voor oneindig veel andere logische waarheden.

Neem de logisch ware zin $[(P \vee Q) \wedge \neg P] \Rightarrow Q$.

- Dit impliceert dat ook de zin $[(Q \vee (\exists x)R(x)) \wedge \neg Q] \Rightarrow (\exists x)R(x)$ logisch waar is. We bekomen deze namelijk door P te vervangen door Q en Q te vervangen door $(\exists x)R(x)$.

- Ook $(\forall x)(\forall y)\{[(R(x,y) \vee Q(y)) \wedge \neg R(x,y)] \Rightarrow Q(y)\}$ is logisch waar. We bekomen deze zin door P te vervangen door $R(x,y)$ en Q door $Q(y)$, en vervolgens de vrije symbolen x, y universeel te kwantificeren.

Deze voorbeelden laten het principe zien: je vervangt propositionele symbolen door formules waarna je universele kwantificatie mag toepassen op vrije symbolen.

Propositie 3.3.1 (Generalisatiepropositie)**.** *Zij A een logisch ware propositionele zin, en voor elk symbool P in A bestaat er een logische formule B_P . We construeren de formule A' door elk symbool P in A te vervangen door B_P. Dan is A' logisch waar.*

Bew. (uit het ongerijmde) Veronderstel dat de eigenschap niet geldig is. A is dus wel maar A' niet logisch waar. Dus bestaat een structuur $\mathfrak{A}$ waarin A' onwaar is. Beschouw nu de structuur $\mathfrak{A}'$ die aan elk symbool P van A de volgende waarde toekent: $P^{\mathfrak{A}'} = (B_P)^{\mathfrak{A}}$. Het is makkelijk in te zien, en in de volgende paragraaf zull we ook effectief bewijzen dat $A^{\mathfrak{A}'}$ en $A'^{\mathfrak{A}}$ identiek zijn. M.a.w. de waarheidswaarde van A in $\mathfrak{A}'$ en van A' in $\mathfrak{A}$ zijn gelijk. Hieruit volgt dat A onwaar is in $\mathfrak{A}'$ en dus niet logisch waar. Contradictie.

We bewijzen formeel dat $A^{\mathfrak{A}'} = A'^{\mathfrak{A}}$ door middel van het principe van *bewijs door inductie* op de grootte van A, t.t.z. op het aantal connectieven dat het bevat[2].

- De eigenschap geldt voor formules van grootte 0. Inderdaad, zo'n formule is van de vorm P, een atomische formule. Dan is A' niets anders dan B_P. Er geldt per definitie van $\mathfrak{A}'$ dat $P^{\mathfrak{A}'} = {B_P}^{\mathfrak{A}}$.

- Laat ons aannemen dat de eigenschap geldt voor alle formules van grootte $\leq n$. Dus, voor formules A van grootte n of kleiner geldt $A^{\mathfrak{A}'} = A'^{\mathfrak{A}}$. Deze veronderstelling noemt men de *inductiehypothese*.

  We hebben net bewezen dat deze hypothese geldig is voor $n = 0$. We zullen vervolgens bewijzen dat als de inductiehypothese geldig is voor n dan is ze ook geldig voor $n + 1$.

  Veronderstel dat A grootte $n+1$ heeft. A bevat dus minstens 1 connectief. We maken een gevallenonderscheid.

  - Veronderstel dat $A = B \wedge C$. Bijgevolg is A' van de vorm $B' \wedge C'$. De grootte van B en C is hoogstens n. Door de inductiehypothese volgt dat $B^{\mathfrak{A}'} = B'^{\mathfrak{A}}$ en $C^{\mathfrak{A}'} = C'^{\mathfrak{A}}$. Het spreekt dan vanzelf dat $(B \wedge C)^{\mathfrak{A}'} = (B' \wedge C')^{\mathfrak{A}}$.
  - De andere gevallen gaan analoog.

[2]Een bewijs door inductie dient om een eigenschap $\mathcal{H}(n)$ te bewijzen voor elk natuurlijk getal n. $\mathcal{H}(n)$ wordt de inductiehypothese genoemd.

Een bewijs per inductie verloopt volgens twee stappen:

- Bewijs $\mathcal{H}(0)$. Maw. de eigenschap geldt voor 0.
- Bewijs dat uit $\mathcal{H}(n)$ volgt dat $\mathcal{H}(n+1)$.

Dus: de eigenschap geldt voor 0, dus voor 1, dus voor 2, dus voor 3, etc., dus voor alle natuurlijke getallen. Gebruik dit principe bij wijze van oefening om de geldigheid van de volgende eigenschap $\mathcal{H}(n)$ te bewijzen: "n of $n+1$ is een even getal".

Aangezien de inductiehypothese geldt voor $n = 0$ en dat uit de inductiehypothese voor n die voor $n+1$ volgt, geldt ze voor alle natuurlijke getallen n, en dus voor formules van willekeurige grootte. Q.E.D.

Propositie 3.3.2. *Zij A een formule met vrije constante-symbolen $v_0, \ldots, v_n$. Dan is A logisch waar asa $(\forall v_0) \ldots (\forall v_n)A$ logisch waar is.*

Als een zin altijd waar is, onafhankelijk van de interpretatie van $v_0, \ldots, v_n$, dan is deze zin in elk domein waar voor alle waarden van v_i.

Bew. Uit het ongerijmde. Veronderstel dat $\models A$ en $\not\models (\forall v_0) \ldots (\forall v_n)A$. Dan bestaat een structuur $\mathfrak{A}$ zodat $\mathfrak{A} \not\models (\forall v_0) \ldots (\forall v_n)A$. Door n-toepassingen van de $\forall$-regel vinden we dat er domeinelementen $d_0, \ldots, d_n$ bestaan zodat $\mathfrak{A}[x_0 : v_0, \ldots, x_n : d_n] \not\models A$. Dus bestaat een structuur waarin A onwaar is. Contradictie. Q.E.D.

Hier is nog een zeer nuttige vervangingseigenschap.

Propositie 3.3.3 (Vervangpropositie)**.** *Zij A, B logisch equivalente formules. Zij C een formule waarin A één of meerdere keren als deelformule voorkomt. Veronderstel dat de formule C' kan bekomen worden door één of meerdere voorkomens van A te vervangen door B. Dan geldt dat C en C' logisch equivalent zijn.*

Voorbeeld 3.3.1. Als toepassing van deze eigenschappen geven we nu een alternatief bewijs van één van de fundamentele equivalenties die we zonet zagen: de contrapositie eigenschap 3.3.4(4): $(P \Leftrightarrow Q) \Leftrightarrow (\neg P \Leftrightarrow \neg Q)$.

$$\begin{array}{ll} \mathfrak{A} \models P \Leftrightarrow Q & \text{asa } \mathfrak{A} \models (P \wedge Q) \vee (\neg P \wedge \neg Q) \\ & \text{asa } \mathfrak{A} \models (\neg\neg P \wedge \neg\neg Q) \vee (\neg P \wedge \neg Q) \\ & \text{asa } \mathfrak{A} \models (\neg P \Leftrightarrow \neg Q) \end{array}$$

Bijgevolg zijn de eerste en de laatste formule waar in exact dezelfde structuren. Uit Propositie 3.1.2 volgt dat $(P \Leftrightarrow Q) \Leftrightarrow (\neg P \Leftrightarrow \neg Q)$ logisch waar is.

Laat ons deze stappen in detail bekijken. Voor de eerste stap vertelt Eig. 3.3.3(3) dat $(P \Leftrightarrow Q) \Leftrightarrow [(P \wedge Q) \vee (\neg P \wedge \neg Q)]$ logisch waar is. Prop. 3.1.2 geeft ons dat $P \Leftrightarrow Q$ logisch equivalent is met $(P \wedge Q) \vee (\neg P \wedge \neg Q)$ wat per definitie betekent dat voor elke structuur $\mathfrak{A}$, $\mathfrak{A} \models P \Leftrightarrow Q$ asa $\mathfrak{A} \models (P \wedge Q) \vee (\neg P \wedge \neg Q)$.

In de tweede stap gebruiken we de logisch ware formule $\neg\neg P \Leftrightarrow P$ (Eig. 3.3.2(1)). Uit de Generalisatiepropositie 3.3.1 volgt dat ook $\neg\neg Q \Leftrightarrow Q$ logisch waar is. Prop. 3.1.2 impliceert dat $\neg\neg P$ en P logisch equivalent zijn, en net zo $\neg\neg Q$ en Q. Uit de Vervangpropositie 3.3.3 volgt dat we het eerste voorkomen van P door $\neg\neg P$ en van Q door $\neg\neg Q$ mogen vervangen, met behoud van de waarheidswaarde.

Voor de derde stap gaat de precieze redenering als volgt: aangezien $(P \Leftrightarrow Q) \Leftrightarrow [(P \wedge Q) \vee (\neg P \wedge \neg Q)]$ logisch waar is (Eig. 3.3.3(3)), volgt uit de generalisatiepropositie (vervang P door $\neg P$ en Q door $\neg Q$) dat $(\neg P \Leftrightarrow \neg Q) \Leftrightarrow [(\neg P \wedge \neg Q) \vee (\neg\neg P \wedge \neg\neg Q)]$ logisch waar is.

Disjunctie is commutatief: $\models (P \vee Q) \Leftrightarrow (Q \vee P)$ 3.3.1(1). Door toepassing van de generalisatiepropositie bekomen we $\models [(\neg P \wedge \neg Q) \vee (\neg\neg P \wedge \neg\neg Q)] \Leftrightarrow [(\neg\neg P \wedge \neg\neg Q) \vee (\neg P \wedge \neg Q)]$.

Dus zijn $(\neg P \wedge \neg Q) \vee (\neg\neg P \wedge \neg\neg Q)$ en $(\neg\neg P \wedge \neg\neg Q) \vee (\neg P \wedge \neg Q)$ logisch equivalent. Door de vervangpropositie geldt dat $\models (\neg P \Leftrightarrow \neg Q) \Leftrightarrow [(\neg\neg P \wedge \neg\neg Q) \vee (\neg P \wedge \neg Q)]$.

Dus zijn $(\neg P \Leftrightarrow \neg Q)$ en $(\neg\neg P \wedge \neg\neg Q) \vee (\neg P \wedge \neg Q)$ logisch equivalent. Ze hebben dus dezelfde waarheidswaarde in $\mathfrak{A}$.

Q.E.D.

We zullen niet doorgaan met bewijzen te leveren op zo'n gedetailleerde manier, maar het is goed om zien dat we elke stap in ons bewijs kunnen argumenteren op basis van eerdere eigenschappen en proposities waarvan we de correctheid hebben bewezen. Het is hierop dat het bouwwerk van de wiskunde gebaseerd is. Dit is wat de Oude Grieken ontdekt hebben, en het is uit de studie van dergelijke gedetailleerde bewijzen dat logica gegroeid is!

Wetten van de kwantificatie.

Eigenschap 3.3.6. Zij A een formule waarin x niet vrij voorkomt.

1. $(\forall x)A \Leftrightarrow A$ is logisch waar.
2. $(\exists x)A \Leftrightarrow A$ is logisch waar.

Eigenschap 3.3.7 (negatie van een kwantor)**.** Zij A een formule en x een variabele. De volgende equivalenties zijn logisch waar.

1. $\neg(\exists x)A \Leftrightarrow (\forall x)\neg A$
2. $\neg(\forall x)A \Leftrightarrow (\exists x)\neg A$
3. $(\exists x)A \Leftrightarrow \neg(\forall x)\neg A$
4. $(\forall x)A \Leftrightarrow \neg(\exists x)\neg A$

Bew. Merk op dat (3) kan afgeleid worden uit (1) door contrapositie toe te passen $\neg\neg(\exists x)A \Leftrightarrow \neg(\forall x)\neg A)$ en vervolgens de dubbele negatie te schrappen. Op dezelfde manier kan (4) uit (2) afgeleid worden.

Wat (1) betreft, geldt:

$\mathfrak{A} \models \neg(\exists x)A$ asa $\mathfrak{A} \not\models (\exists x)A$
asa voor geen enkel domein element d van $\mathfrak{A}$ geldt $\mathfrak{A}[x:d] \models A$
asa voor elk domein element d van $\mathfrak{A}$ geldt $\mathfrak{A}[x:d] \not\models A$
asa voor elk domein element d van $\mathfrak{A}$ geldt $\mathfrak{A}[x:d] \models \neg A$
asa $\mathfrak{A} \models (\forall x)\neg A$.

Het bewijs van (2) is analoog. Q.E.D.

Eigenschap 3.3.8 (Doorschuiven van kwantoren)**.** Zij A en B formules, en x een variabele, dan geldt:

1. $\models (\exists x)(A \vee B) \Leftrightarrow (\exists x)A \vee (\exists x)B$ — $\exists$ schuift door $\vee$

2. $\models (\forall x)(A \wedge B) \Leftrightarrow (\forall x)A \wedge (\forall x)B$ — $\forall$ schuift door $\wedge$

3. $\models (\exists x)(A \wedge B) \Rightarrow (\exists x)A \wedge (\exists x)B$ — maar
   $\not\models (\exists x)(A \wedge B) \Leftarrow (\exists x)A \wedge (\exists x)B$ — tenzij
   $\models (\exists x)(A \wedge B) \Leftrightarrow (\exists x)A \wedge B$ — als x geen vrije constante is van B.

4. $\models (\forall x)(A \vee B) \Leftarrow (\forall x)A \vee (\forall x)B$ — maar
   $\not\models (\forall x)(A \vee B) \Rightarrow (\forall x)A \vee (\forall x)B$ — tenzij
   $\models (\forall x)(A \vee B) \Leftrightarrow (\forall x)A \vee B$ — als x geen vrije constante is van B

5. $\models (\forall x)(A \Rightarrow B) \Rightarrow ((\exists x)A \Rightarrow (\exists x)B)$ — maar
   $\not\models (\forall x)(A \Rightarrow B) \Leftarrow ((\exists x)A \Rightarrow (\exists x)B)$

6. $\models (\forall x)(A \Rightarrow B) \Rightarrow ((\forall x)A \Rightarrow (\forall x)B)$ — maar
   $\not\models (\forall x)(A \Rightarrow B) \Leftarrow ((\forall x)A \Rightarrow (\forall x)B)$

Dus, de formules volgend op $\models$ zijn tautologieën, en degene volgend op $\not\models$ zijn dat niet.

Slechts in een beperkt aantal gevallen kunnen we dus de kwantoren doorschuiven. We bewijzen 3 gevallen (1), (3) en (5). De andere zijn analoog of kunnen door contrapositie bewezen worden uit de vorige.

Bew. van (1). Zij $\mathfrak{A}$ een willekeurige structuur die A, B interpreteert.

($\Rightarrow$) Veronderstel dat $\mathfrak{A} \models (\exists x)(A \vee B)$, dan bestaat er een $d \in D_{\mathfrak{A}}$ zodat $\mathfrak{A}[x:d] \models A \vee B$. Dus er zijn twee gevallen: $\mathfrak{A}[x:d] \models A$ of $\mathfrak{A}[x:d] \models B$. In het eerste geval geldt $\mathfrak{A} \models (\exists x)A$, in het tweede geldt $\mathfrak{A} \models (\exists x)B$. Dus geldt $\mathfrak{A} \models (\exists x)A \vee (\exists x)B$.

($\Leftarrow$) Veronderstel dat $\mathfrak{A} \models (\exists x)A \vee (\exists x)B$. Er geldt dan dat (a) $\mathfrak{A} \models (\exists x)A$ of dat (b) $\mathfrak{A} \models (\exists x)B$.

In het geval (a) bestaat $d \in D_{\mathfrak{A}}$ zodat $\mathfrak{A}[x:d] \models A$. En dus ook $\mathfrak{A}[x:d] \models A \vee B$. Uit de $\exists$-regel volgt dan $\mathfrak{A} \models (\exists x)(A \vee B)$.

In geval (b) komen we tot precies dezelfde conclusie. Q.E.D.

Bew. van (3). Uit $\mathfrak{A} \models (\exists x)(A \wedge B)$ volgt dat een $d \in D_{\mathfrak{A}}$ bestaat zodat $\mathfrak{A}[x:d] \models A$ en $\mathfrak{A}[x:d] \models B$, en bijgevolg $\mathfrak{A} \models (\exists x)A \wedge (\exists x)B$. Dit geldt voor elke $\mathfrak{A}$ en dus is $(\exists x)A \wedge (\exists x)B$ een logische gevolg van $(\exists x)(A \wedge B)$. Bijgevolg is $(\exists x)(A \wedge B) \Rightarrow (\exists x)A \wedge (\exists x)B$ logisch waar.

Om aan te tonen dat $\not\models (\exists x)(A \wedge B) \Leftarrow (\exists x)A \wedge (\exists x)B$ moeten we A en B kiezen en een structuur construeren die deze implicatie onwaar maakt. Kies $A = P(x)$, $B = Q(x)$ en de structuur $\mathfrak{A}$ met domein $\{a, b\}$, $P^{\mathfrak{A}} = \{a\}$, $Q^{\mathfrak{A}} = \{b\}$. Verifieer dat in deze structuur de premisse waar is en de conclusie onwaar. Of ook, kies $A = (x = C), B = (\neg x = C)$.

Maar wat als x niet vrij is in B (en dus $(\exists x)B$ en B logisch equivalent zijn) dan geldt de omgekeerde richting wel! Als $\mathfrak{A} \models (\exists x)A \wedge B$, dan bestaat $d \in D_{\mathfrak{A}}$ zodat $\mathfrak{A}[x:d] \models A$ en $\mathfrak{A} \models B$. Aangezien x niet vrij is in B is de waarde van x in $\mathfrak{A}[x:d]$ van geen belang, zodat $\mathfrak{A}[x:d] \models B$. Bijgevolg geldt $\mathfrak{A} \models (\exists x)(A \wedge B)$. Q.E.D.

Bew. van (5). Veronderstel $\mathfrak{A} \models (\forall x)(A \Rightarrow B)$ and $\mathfrak{A} \models (\exists x)A$. Dan geldt voor een $d \in D_{\mathfrak{A}}$ dat $\mathfrak{A}[x:d] \models A$ (met de $\exists$-regel). Uit de $\forall$-regel volgt $\mathfrak{A}[x:d] \models A \Rightarrow B$. Bijgevolg $\mathfrak{A}[x:d] \models B$ ($\Rightarrow$-regel). En dus geldt $\mathfrak{A} \models (\exists x)B$ ($\exists$-regel).

Voor het niet gelden van de omgekeerde richting: kies een structuur waarn $P^{\mathfrak{A}}$ en $Q^{\mathfrak{A}}$ beide niet leeg zijn maar een lege doorsnede hebben. Dan geldt wel de rechtse deelformule, maar niet de linkse deelformule. Q.E.D.

Oefening 3.3.3. *Zie Oefening* **??**, *Oefening* **??**. *In het vorig hoofdstuk heb je de veralgemeende kwantoren vertaald naar logica, en heb je ook bepaald welke kwantoren elkaars negatie waren. Bewijs dit nu formeel met behulp van de wetten van negatie. Bijvoorbeeld:*

- *Alle P's zijn Q's:* $(\forall x)(P(x) \Rightarrow Q(x))$.
- *Minstens één P is geen Q:* $(\exists x)(P(x) \wedge Q(x))$.

De corresponderende zinnen zijn inderdaad elkaars negatie:

$$\begin{aligned} \neg(\forall x)(P(x) \Rightarrow Q(x)) &\equiv (\exists x)\neg(P(x) \Rightarrow Q(x)) \\ &\equiv (\exists x)\neg(\neg P(x) \vee Q(x)) \\ &\equiv (\exists x)(\neg\neg P(x) \wedge \neg Q(x)) \\ &\equiv (\exists x)(P(x) \wedge \neg Q(x)) \end{aligned}$$

Hoe zit het met het doorschuiven van kwantoren door $\Rightarrow$? De regels daarvan kunnen we vinden door $A \Rightarrow B$ te vertalen naar $\neg A \vee B$, en de bestaande toe te passen. Er zitten een paar merkwaardige gevallen bij. Bv.

$$\models (\exists x)(A \Rightarrow B) \Leftrightarrow (\forall x)A \Rightarrow (\exists x)B$$

Zeer vreemd! De meeste regels stroken met ons taalgevoel aan, maar deze? We botsen hier opnieuw tegen een materiële implicatie $(\exists x)(A \Rightarrow B)$ die wij mensen vaak anders zullen interpreteren dan logica doet. Wie zich daarvan wil vergewissen moet eens kijken naar de formule $(\exists x)(Succeeds(x) \Rightarrow Pass)$ in het laatste voorbeeld van Hoofdstuk 2. Doorschuiven van $\exists$ geeft $((\forall x)(Succeeds(x)) \Rightarrow (\exists x)Pass)$, wat vereenvoudigt tot $((\forall x)Succeeds(x)) \Rightarrow Pass)$. Beide zinnen zijn dus logisch equivalent, en dat is zeer onverwacht.

Let op als je logische zinnen neerschrijft van de vorm $(\exists x)(A \Rightarrow B)$.

Eigenschap 3.3.9 (verwisselen van kwantoren)**.** Zij A een formule en zij x, y variabelen, dan

1. $\models (\forall x)(\forall y)A \Leftrightarrow (\forall y)(\forall x)A$
2. $\models (\exists x)(\exists y)A \Leftrightarrow (\exists y)(\exists x)A$
3. $\models (\exists x)(\forall y)A \Rightarrow (\forall y)(\exists x)A$ maar
   $\not\models (\exists x)(\forall y)A \Leftarrow (\forall y)(\exists x)A$

Bew. van (1) en (2): evident.

Bew. van (3). De logische waarheid in de $\Rightarrow$ richting is evident: voor de tweede formule kies je voor elke y die ene x die volgens de eerste deelformule bestaat en A waar maakt.

Een tegenvoorbeeld voor de logische waarheid van de $\Leftarrow$ richting hebben we al eerder gegeven in Voorbeeld 2.5.1.

Nog een eigenschap van transformaties die equivalentie bewaren: we mogen gebonden variabelen in een formule $(\forall x)A[x]$ of $(\exists x)A[x]$ altijd vervangen door andere variabelen om bv. $(\forall y)A[y]$ of $(\exists y)A[y]$ te bekomen, op 1 voorwaarde: dat y niet vrij voorkomt in $A[x]$.

Propositie 3.3.4 (Een gebonden variabele van naam veranderen)**.** *Zij $A[x]$ een formule waarin o.a. x vrij mag voorkomen. Zij $A[y]$ de formule die men bekomt door in $A[x]$, op elke plaats waar x vrij voorkomt, x te vervangen door y.*

Als y niet voorkomt in de formule $A[x]$ dan hebben we:

$$\models (\exists x)A[x] \Leftrightarrow (\exists y)A[y]$$
$$\models (\forall x)A[x] \Leftrightarrow (\forall y)A[y]$$

Terminologie: $A[y]$ is bekomen uit $A[x]$ door **substitutie** van y voor x.

Dat de conditie dat y niet voorkomt in $A[x]$ nodig is, is ergens evident. Immers, het vooraan toevoegen van een kwantor $(\forall y)$ of $(\exists y)$ zou zo'n bestaand voorkomen van y binden en dus van betekenis doen veranderen. Als je voor $A[x]$ de formule $\neg x = y$ neemt, dan is $(\exists x)\neg x = y \Leftrightarrow (\exists y)\neg y = y$ evident niet logisch waar. Als je voor $A[x]$ de formule $(\forall y)x = y$ neemt, dan is $(\forall x)(\forall y)x = y \Leftrightarrow (\forall y)(\forall y)y = y$ evident niet logisch waar.

3.4 Normaalvormen

Vele logische tools vereisen dat hun invoerformules in een speciale vorm staan. Dit noemt men normaalvormen.

Definitie 3.4.1. Een formule A is in **prenex-normaalvorm** als geen enkele kwantor voorkomt in een conjunctie, disjunctie, implicatie of equivalentie.

Voorbeeld 3.4.1.

$(\exists z)(\forall x)(\exists v)\,((P(z) \Rightarrow Q(w,x)) \wedge \neg P(u))$	in prenex-normaalvorm
$(\exists z)\,((\exists x)P(x) \wedge (\exists u)Q(v,u))$	niet in prenex-normaalvorm
$(\exists z)P(z) \wedge Q(z)$	niet in prenex-normaalvorm

Merk op dat in deze laatste zin $\exists$ syntactisch gezien vooraan staat, maar toch komt deze $\exists$ voor binnen de conjunctie. M.a.w., de formule is $((\exists z)P(z)) \wedge Q(z)$ en is niet in prenex-normaalvorm.

Propositie 3.4.1 (Prenex-normaalvorm)**.** *Elke formule is logisch equivalent met een formule in prenex-normaalvorm.*

Bew. Elke formule kan in prenex-normaalvorm gebracht worden met equivalentie bewarende transformatiestappen. Eerst vertalen we $\Rightarrow$ en $\Leftrightarrow$ in $\wedge, \vee$ en $\neg$. Vervolgens worden de kwantoren naar buiten geschoven. Daarvoor gebruiken we de volgende transformaties :

- $\neg(\forall x)A$ wordt $(\exists x)\neg A$ (Eig. 3.3.7)
- $\neg(\exists x)A$ wordt $(\forall x)\neg A$ (Eig. 3.3.7).
- $A \wedge (\exists x)B[x]$ wordt $(\exists y)(A \wedge B[y])$
  Hierbij is y een nieuwe variabele die niet voorkomt in de formule. We hernoemen x door y (Prop. 3.3.4) en schuiven vervolgens $(\exists y)$ naar buiten (Eig. 3.3.8(3iii) is van toepassing aangezien y niet voorkomt in A). Deze methode is van toepassing voor alle formules hieronder.
  Merk op dat als x niet voorkomt in A, hernoeming overbodig is.
- $(\exists x)A[x] \wedge B$ wordt $(\exists y)(A[y] \wedge B)$
- $A \vee (\exists x)B[x]$ wordt $(\exists y)(A \vee B[y])$
- $(\exists x)A[x] \vee B$ wordt $(\exists y)(A[y] \vee B)$
- $A \wedge (\forall x)B[x]$ wordt $(\forall y)(A \wedge B[y])$
- $(\forall x)A[x] \wedge B$ wordt $(\forall y)(A[y] \wedge B)$
- $A \vee (\forall x)B[x]$ wordt $(\forall y)(A \vee B[y])$
- $(\forall x)A[x] \vee B$ wordt $(\forall y)(A[y] \vee B)$.

Door herhaalde toepassing van deze equivalentie bewarende omzettingen toe te passen, kunnen we al de kwantoren naar voren schuiven. Q.E.D.

Voorbeeld 3.4.2. Breng $(\exists x)P(x) \Rightarrow (\forall y)Q(x,y)$ in prenex-normaalvorm.

Oplossing :

$$
\begin{aligned}
&(\exists x)P(x) \Rightarrow (\forall y)Q(x,y) && \Leftrightarrow \\
&\neg(\exists x)P(x) \vee (\forall y)Q(x,y) && \Leftrightarrow \\
&(\forall x)\neg P(x) \vee (\forall y)Q(x,y) && \Leftrightarrow \\
&(\forall z)\,[\neg P(z) \vee (\forall y)Q(x,y)] && \Leftrightarrow \\
&(\forall z)(\forall y)\,[(\neg P(z)) \vee Q(x,y)]\ . &&
\end{aligned}
$$

Voorbeeld 3.4.3. Breng $(\exists x)P(x) \wedge (\exists x)R(x)$ in prenex-normaalvorm.

Oplossing :

$$
\begin{aligned}
&(\exists x)P(x) \wedge (\exists x)R(x) && \Leftrightarrow \\
&(\exists x)\,[P(x) \wedge (\exists x)R(x)] && \Leftrightarrow \\
&(\exists x)\,[P(x) \wedge (\exists y)R(y)] && \Leftrightarrow \\
&(\exists x)(\exists y)\,[P(x) \wedge R(y)]\ . &&
\end{aligned}
$$

Opmerking 3.4.1. In de procedure voor het bekomen van de prenex-normaalvorm mogen vrij symbolen nooit van naam veranderd worden.

Definitie 3.4.2.

Een **literal** is een atomaire formule $P(t_1, \ldots, t_n)$ of de negatie ervan $\neg P(t_1, \ldots, t_n)$.

Een formule is in **negatie-normaalvorm** als $\wedge, \vee$ en $\neg$ de enige connectieven zijn en als het negatiesymbool $\neg$ enkel in literals voorkomt.

Een **clause** is een disjunctie van literals.

Een formule is in **conjunctieve normaalvorm** (CNF) als het een conjunctie van clauses is. Zo een formule kan ook gezien worden als een theorie bestaande uit clauses.

Een formule is in **disjunctieve normaalvorm** (DNF) als het een disjunctie is van conjuncties van literals.

Een logische zin is in **prenex-conjunctieve normaalvorm** (pre-CNF) als het in prenex-normaalvorm is, en de kwantorvrije deelformule is in conjunctieve normaalvorm.

Een logische zin is in **prenex-disjunctieve normaalvorm** (pre-DNF) als het in prenex-normaalvorm is, en de kwantorvrije deelformule is in disjunctieve normaalvorm.

Propositie 3.4.2. *Een formule A is logisch equivalent met een formule B in prenex-disjunctieve normaalvorm en met een formule C in prenex-conjunctieve normaalvorm.*

Merk op dat de prenex-disjunctieve normaalvorm van een propositionele zin geen kwantoren bevat en dus in DNF is, en analoog voor CNF.

Bew. Eerst brengen we A in prenex-normaalvorm A'. Vervolgens brengen we A' in negatie-normaalvorm door $\neg$ door $\wedge$ and $\vee$ te schuiven gebruik makend van de wetten van de Morgan (Eig. 3.3.2). De laatste stap hangt af of we disjunctieve dan wel conjunctieve normaalvorm willen bereiken. Indien we disjunctieve normaalvorm willen bekomen, dan schuiven we $\wedge$ door $\vee$ door middel van de distributiviteit van $\wedge$ t.o.v. $\vee$. Willen we de conjunctieve normaalvorm, dan passen we herhaald de distributiviteit van $\vee$ t.o.v. $\wedge$ toe en schuiven zo $\vee$ door $\wedge$.

Ook van toepassing zijn idempotentie-regels, om herhalingen van dezelfde deelformule in een conjunctie of disjunctie te verwijderen, en het verwijderen van inconsistenties. Q.E.D.

Voorbeeld 3.4.4. $\neg(P \wedge Q) \wedge (R \vee \neg T)$ is niet in disjunctieve normaalvorm. We hebben echter de volgende logische equivalenties:

$$
\begin{aligned}
&\neg(P \wedge Q) \wedge (R \vee \neg T) \\
&\qquad \Leftrightarrow (\neg P \vee \neg Q) \wedge (R \vee \neg T) \quad \text{De Morgan} \\
&\qquad \Leftrightarrow [(\neg P \vee \neg Q) \wedge R] \vee [(\neg P \vee \neg Q) \wedge \neg T] \quad \text{distributiviteit} \\
&\qquad \Leftrightarrow [(\neg P \wedge R) \vee (\neg Q \wedge R)] \vee [(\neg P \wedge \neg T) \vee (\neg Q \wedge \neg T)] \\
&\qquad \Leftrightarrow (\neg P \wedge R) \vee (\neg Q \wedge R) \vee (\neg P \wedge \neg T) \vee (\neg Q \wedge \neg T)
\end{aligned}
$$

De laatste zin is in disjunctieve normaalvorm.

Voorbeeld 3.4.5. $\neg(P \Rightarrow Q) \Leftrightarrow (R \vee Q)$ is niet in disjunctieve normaalvorm. We hebben echter

de volgende equivalenties:

$$
\begin{aligned}
&[\neg(P \Rightarrow Q) \Leftrightarrow (R \vee Q)] \\
&\qquad \Leftrightarrow [\neg(P \Rightarrow Q) \Rightarrow (R \vee Q)] \wedge [(R \vee Q) \Rightarrow \neg(P \Rightarrow Q)] \\
&\qquad \Leftrightarrow [(P \Rightarrow Q) \vee (R \vee Q)] \wedge [\neg(R \vee Q) \vee \neg(P \Rightarrow Q)] \\
&\qquad \Leftrightarrow \big[(\neg P \vee \underline{Q}) \vee (R \vee \underline{Q})\big] \wedge [\neg(R \vee Q) \vee \neg(\neg P \vee Q)] \\
&\qquad \Leftrightarrow [\neg P \vee Q \vee R] \wedge [(\neg R \wedge \neg Q) \vee (P \wedge \neg Q)] \\
&\qquad \Leftrightarrow [(\neg P \vee Q \vee R) \wedge (\neg R \wedge \neg Q)] \vee [(\neg P \vee Q \vee R) \wedge (P \wedge \neg Q)] \\
&\qquad \Leftrightarrow (\neg P \wedge \neg R \wedge \neg Q) \vee (\underline{Q} \wedge \neg R \wedge \neg \underline{Q}) \vee (\underline{R} \wedge \neg \underline{R} \wedge \neg Q) \vee (\neg \underline{P} \wedge \underline{P} \wedge \neg Q) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee (\underline{Q} \wedge P \wedge \neg \underline{Q}) \vee (R \wedge P \wedge \neg Q) \\
&\qquad \Leftrightarrow (\neg P \wedge \neg Q \wedge \neg R) \vee (P \wedge \neg Q \wedge R)
\end{aligned}
$$

De laatste zin is in disjunctieve normaalvorm. Merk op dat we conjuncties met een symbool en zijn negatie verwijderd hebben.

Voorbeeld 3.4.6. Breng in disjunctieve normaalvorm

$$[(P \wedge R) \Rightarrow \neg Q] \wedge (Q \vee R).$$

$$
\begin{aligned}
&[(P \wedge R) \Rightarrow \neg Q] \wedge (Q \vee R) \\
&\qquad \Leftrightarrow [\neg(P \wedge R) \vee \neg Q] \wedge (Q \vee R)) \\
&\qquad \Leftrightarrow [\neg P \vee \neg R \vee \neg Q] \wedge (Q \vee R)) \\
&\qquad \Leftrightarrow [\neg P \wedge (Q \vee R)] \vee [\neg R \wedge (Q \vee R)] \vee [\neg Q \wedge (Q \vee R)]) \\
&\qquad \Leftrightarrow (\neg P \wedge Q) \vee (\neg P \wedge R) \vee (\neg R \wedge Q) \vee \underline{(\neg R \wedge R)} \vee \underline{(\neg Q \wedge Q)} \vee (\neg Q \wedge R)) \\
&\qquad \Leftrightarrow (\neg P \wedge Q) \vee (\neg P \wedge R) \vee (\neg R \wedge Q) \vee (\neg Q \wedge R)
\end{aligned}
$$

Propositie 3.4.3. *Een propositionele zin in disjunctieve normaalvorm is consistent asa het minstens één conjunctie bevat zonder tegengestelde literals P en $\neg P$.*

Een propositionele zin in conjunctieve normaalvorm is logisch waar asa elke clause een literal P en zijn negatie $\neg P$ bevat.

Bew. Als elke conjunctie B in de DNF formule A literals P en $\neg P$ bevat, dan zal elke conjunctie en dus A zelf onwaar zijn in elke structuur. Omgekeerd, als A een conjunctie $B = L_1 \wedge \cdots \wedge L_n$ bevat zonder tegengestelde literals, kies dan een willekeurige structuur $\mathfrak{A}$ die $L_1, \ldots, L_n$ waar maakt. $\mathfrak{A}$ voldoet aan A.

Het bewijs van het tweede deel is analoog. Q.E.D. .

We zien hier een potentieel belangrijk voordeel van formules in DNF en in CNF: we kunnen gemakkelijk herkennen of ze consistent dan wel logisch waar zijn. Je zou verwachten dat propositionele solvers die berekenen of een formule consistent of logisch waar is, deze omzetten naar DNF, respectievelijk CNF. Maar helaas werkt dat helemaal niet goed: immers vaak is de DNF en CNF vorm van een formule exponentieel veel groter dan de oorspronkelijke formule. Men heeft dus andere methodes ontwikkeld. Dit gebeurt in het domein van SAT solvers.

Oefening 3.4.1. *Breng de volgende formule in conjunctieve normaalvorm*

$$[(P \Rightarrow R) \Leftrightarrow \neg(R \wedge (Q \Rightarrow P))]$$

3.5 Formele bewijzen

In de wiskunde wordt een wiskundig bewijs, zoals bv. van de Generalisatiepropositie 3.3.1 en van alle andere proposities en stellingen tot nog toe, ook wel eens een *formeel bewijs* genoemd. Maar deze bewijzen zijn helemaal niet "formeel". Het zijn zorgvuldig geformuleerde, sluitende redeneringen over wiskundige objecten, maar ze zijn wel geformuleerd in het nederlands. Om ze te begrijpen moeten we beroep doen op ons intuïtief begrip van de nederlands. Er zijn geen vaste regels waarmee zulke bewijzen geconstrueerd mogen worden. Dus, wat men in de wiskunde vaak een formeel (of wiskundig) bewijs noemt, is eigenlijk helemaal geen formeel bewijs. De logica is precies ontstaan met als doel dergelijke wiskundige bewijzen te onderzoeken en een formele notie van bewijs te introduceren. Een bewijs volgens de methode die we hier zullen zien is wel een formeel wiskundig object: het is een boom van strings die opgesteld wordt volgens strikt wiskundige regels.

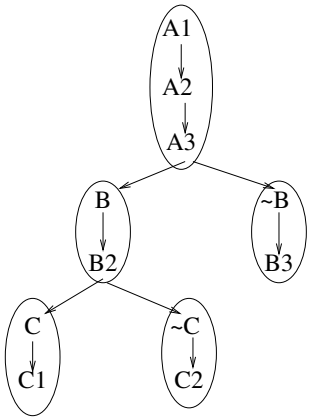
In de loop van de 20ste eeuw werden talloze formele bewijssystemen voorgesteld. We zullen één methode zien. Deze methode noemt men de KE-bewijsmethode (van *Klassiek Eliminatie-bewijs*). Omdat deze methode wel formeel is bestaan er computerprogramma's die de correctheid van zo'n bewijs kunnen verifiëren. Zo'n programma is beschikbaar in LogicPalet. In principe is het zelfs mogelijk om computerprogramma's te bouwen die zo'n bewijs automatisch kunnen genereren.

De methode dient om de inconsistentie van een theorie te bewijzen. Dat volstaat echter om ook andere logische redeneertaken op te lossen, namelijk die herleid kunnen worden tot bewijs van inconsistentie:

- Bewijs dat een zin A logisch waar is: immers A is logisch waar asa $\neg A$ is inconsistent.
- Bewijs dat een zin B een logisch gevolg is van een logische theorie $T = \{A_1, \ldots, A_n\}$. Immers, dit is het geval asa $\{A_1, \ldots, A_n, \neg B\}$ inconsistent is.
- Bewijs dat een zin B inconsistent is met de logische theorie T. Dat is het geval asa $\{A_1, \ldots, A_n, B\}$ inconsistent is.

De KE-bewijsmethode dient om inconsistentie te bewijzen, maar soms kan ze ook gebruikt worden om de consistentie van een theorie te bewijzen. Niet altijd evenwel zoals we later zullen zien.

Een KE-redenering uit een theorie $T = \{A_1, A_2, \ldots A_n\}$ bestaat uit boom van logische zinnen. Figuur 3.2 toont de structuur van een dergelijke boom. Hierin zijn A_i, B_i, C_i zinnen. De deelrijen komen overeen met de 5 omcirkelde rijen zinnen, bv. $A1, A2, A3$. Waar de boom splitst begint één tak met een formule en de andere met de negatie van deze formule. Anders gezegd, aan het eind van een rij kan de rij opgesplitst worden in twee deelrijen die met tegengestelde formules $A, \neg A$ beginnen.



Figuur 3.2: Structuur van een KE-bewijs

De lege boom is een KE-redenering. Een tak van een KE-redenering kan onderaan uitgebreid worden. Er zijn 2 manieren:

- door een nieuwe zin toe te voegen die ofwel afkomstig is uit T, ofwel bekomen wordt door toepassing van een *propagatie-regel* op één of meerdere formules hogerop in de tak; deze propagatie-regels worden verderop uitgelegd;
- ofwel door een gevalsonderscheid te maken voor een zin A en $\neg A$ en dus de tak op te splitsen.

Voorbeeld 3.5.1. We willen bewijzen dat de logische zin uit Voorbeeld 3.1.2 logisch waar is:

$$[(\forall x)(P(x, R1) \vee P(x, R2)) \wedge \neg P(C, R1)] \Rightarrow P(C, R2)$$

Onze theorie bestaat dus uit de negatie van deze zin. Het KE-bewijs is de formele versie van het bewijs[3] uit het ongerijmde in Voorbeeld 3.1.2.

$\neg\{[(\forall x)(P(x, R1) \vee P(x, R2)) \wedge \neg P(C, R1)] \Rightarrow P(C, R2)\}$	1. Theorie
$\neg P(C, R2)$	2. $\neg\Rightarrow$-prop(1)
$[(\forall x)(P(x, R1) \vee P(x, R2)) \wedge \neg P(C, R1)]$	3. $\neg\Rightarrow$-prop(1)
$(\forall x)(P(x, R1) \vee P(x, R2))$	4. $\wedge$-prop(3)
$\neg P(C, R1)$	5. $\wedge$-prop(3)
$(P(C, R1) \vee P(C, R2))$	6. $\forall$-prop(4)
Case A	
$P(C, R1)$	7. Case $P(C, R1)$
Contradictie 7,5	
Case B	
$\neg P(C, R1)$	8. Case $\neg P(C, R1)$
$P(C, R2)$	9. $\vee$-prop(6,8)
Contradictie 2,9	

Bv. de tweede stap $\neg(C, R2)$ werd afgeleid door de $\neg\Rightarrow$-prop propagatieregel toegepast op stap 1. De uitleg van deze regel volgt. Aan het eind gebeurt een gevalsonderscheid op $P(C, R1)$,

[3]Er bestaat overigens een verkorte versie van dit bewijs waarin geen gevalsonderscheid voor komt: namelijk door de $\vee$-prop regel toe te passen op 2,6 bekomt men meteen contradictie.

de eerste disjunct van (6). Uit de negatie ervan in Case B volgt de tweede disjunct, die in contradictie is met de zin in stap 2. Alle takken leiden dus tot een contradictie. Dus is de oorspronkelijke zin inderdaad inconsistent en zijn negatie logisch waar.

Vervolgens introduceren we de uitbreidingsregels van KE-bewijzen.

Hypothesen-regel. Elke zin van T mag toegevoegd worden onderaan een tak. We spreken af dit eerst te doen, vooralleer we andere regels toepassen (indien T eindig is).

Propositionele propagatie-regels. Een zin mag toegevoegd worden aan een tak indien deze kan afgeleid worden uit formules hogerop in de tak door middel van één van de propagatie-regels hieronder. De propagatie-regels vertrekken allemaal van hetzelfde idee: als we een samengestelde formule A hebben dan kan uit de waarheidswaarde van A en sommige van zijn deelformules de waarheidswaarde van andere deelformules afgeleid worden. Bv. als $A \wedge B$ waar is, dan is A waar en ook B. Als $A \wedge B$ onwaar is en A waar, dan moet B onwaar zijn.

De Propagatie-regels zijn de volgende:

$\neg$	$\neg\neg A$	$\longrightarrow$	A	$\neg\neg$-prop
$\wedge$	$A \wedge B$	$\longrightarrow$	A, B	$\wedge$-prop
	$\neg(A \wedge B)$, B	$\longrightarrow$	$\neg A$	$\neg\wedge$-prop
	$\neg(A \wedge B)$, A	$\longrightarrow$	$\neg B$	$\neg\wedge$-prop
$\vee$	$A \vee B$, $\neg A$	$\longrightarrow$	B	$\vee$-prop
	$A \vee B$, $\neg B$	$\longrightarrow$	A	$\vee$-prop
	$\neg(A \vee B)$	$\longrightarrow$	$\neg A$, $\neg B$	$\neg\vee$-prop
$\Rightarrow$	$A \Rightarrow B$, A	$\longrightarrow$	B	$\Rightarrow$-prop
	$A \Rightarrow B$, $\neg B$	$\longrightarrow$	$\neg A$	$\Rightarrow$-prop
	$\neg(A \Rightarrow B)$	$\longrightarrow$	$A, \neg B$	$\neg\Rightarrow$-prop
$\Leftrightarrow$	$A \Leftrightarrow B$, A	$\longrightarrow$	B	$\Leftrightarrow$-prop
	$A \Leftrightarrow B$, B	$\longrightarrow$	A	$\Leftrightarrow$-prop
	$A \Leftrightarrow B$, $\neg A$	$\longrightarrow$	$\neg B$	$\Leftrightarrow$-prop
	$A \Leftrightarrow B$, $\neg B$	$\longrightarrow$	$\neg A$	$\Leftrightarrow$-prop
	$\neg(A \Leftrightarrow B)$, A	$\longrightarrow$	$\neg B$	$\neg\Leftrightarrow$-prop
	$\neg(A \Leftrightarrow B)$, B	$\longrightarrow$	$\neg A$	$\neg\Leftrightarrow$-prop
	$\neg(A \Leftrightarrow B)$, $\neg A$	$\longrightarrow$	B	$\neg\Leftrightarrow$-prop
	$\neg(A \Leftrightarrow B)$, $\neg B$	$\longrightarrow$	A	$\neg\Leftrightarrow$-prop

Merk op dat elke regel uit een samengestelde zin een deelzin of zijn negatie afleidt.

We voeren de volgende afspraak in: als een propagatie een formule van de vorm $\neg\neg A$ produceert, dan wordt de dubbele negatie meteen verwijderd. Dat bespaart ons een toepassing van de $\neg\neg$-prop regel.

Oefening 3.5.1. *Elk van deze propagatie-regels komt voort uit een tautologie, namelijk door $\longrightarrow$ te vervangen door $\Rightarrow$ en "," door $\wedge$. Bv. $\neg\neg A \Rightarrow A$. Bewijs deze.*

ERule : $\exists$-regel. Indien $(\exists x)A[x]$ voorkomt in een tak, dan mag men $A[c]$ toevoegen aan die tak, met c een nieuwe constante die nog niet voorkomt in deze tak noch in T. Indien $\neg(\forall x)A[x]$ voorkomt in een tak, dan mag men $\neg A[c]$ toevoegen aan die tak voor c opnieuw zo'n nieuwe constante.

Opmerking 3.5.1. $A[x]$ duidt een formule aan waarin x een vrij voorkomen heeft. Veronderstel dat t een term is die geen enkel symbool y bevat die gekwantificeerd voorkomt in $A[x]$. Dan is $A[t]$ de formule bekomen door elk vrij voorkomen van x te vervangen door t.

Let op: het is altijd en overal verboden om vrije voorkomens van x in $A[x]$ te vervangen door een term die gekwantificeerde variabelen uit $A[x]$ bevat. Bv. neem voor $A[x]$ de formule $Q(x) \vee (\forall y)P(x,y)$, dan mag x nooit vervangen worden door y. Dit zou $Q(y) \vee (\forall y)P(y,y)$ produceren: het probleem is dat de y binnen en buiten de kwantor eigenlijk een verschillend object aanduidt. We mogen x evenmin door $f(y), g(a,y), \ldots$ vervangen.

ARule: $\forall$-regel. Indien $(\forall x)A[x]$ voorkomt in een tak en t is een term dan mag men $A[t]$ toevoegen aan die tak. Indien $\neg(\exists x)A[x]$ voorkomt, dan mag men $\neg A[t]$ toevoegen aan die tak.

Er zijn twee voorwaarden op t: (1) t bevat geen constanten die gekwantificeerd zijn in $A[x]$; (2) elk constante of functie-symbool in t moet eerder vrij voorkomen in de tak, of in de theorie T. (Als er geen constanten in de tak of in T voorkomen, mag je één nieuwe constante introduceren).

Gelijkheid-regels. Zij t een term zonder variabelen. Dan mag men $t = t$ toevoegen.

Zij t_1, t_2 termen zonder variabelen. Indien de zinnen $t_1 = t_2$ en A_1 voorkomen in een tak waarbij A_1 een voorkomen van t_1 bevat, dan mag men A_2 toevoegen aan die tak. Hierbij is A_2 bekomen door het vervangen van één of meerdere voorkomens van t_1 door t_2 in A_1.

Ter herinnering: $t_1 \neq t_2$ is een afkorting van $\neg t_1 = t_2$.

Voorbeeld 3.5.2. We willen bewijzen dat

$$\neg\left((\exists y)[Triangle(y) \wedge (\forall x)\neg LeftOf(x,y)]\right)$$

een logisch gevolg is van

$$(\exists x)(\forall y)[Triangle(y) \Rightarrow LeftOf(x,y)].$$

Dit redeneerprobleem kan herleid worden tot het bewijzen van inconsistentie van de volgende theorie

$$\left\{ \begin{array}{l} (\exists x)(\forall y)[Triangle(y) \Rightarrow LeftOf(x,y)], \\ \neg\left[\neg\left((\exists y)[Triangle(y) \wedge (\forall x)\neg LeftOf(x,y)]\right)\right] \end{array} \right\}.$$

Hier is een KE-bewijs zonder gevalsonderscheid.

$(\exists x)(\forall y)[Triangle(y) \Rightarrow LeftOf(x,y)]$	1. Hypothesis
$\neg\,[\neg\,((\exists y)[Triangle(y) \wedge (\forall x)\neg LeftOf(x,y)])]$	2. NegatedConclusion
$(\exists y)[Triangle(y) \wedge (\forall x)\neg LeftOf(x,y)]$	3. $\neg\neg$-regel(2)
$(\forall y)[Triangle(y) \Rightarrow LeftOf(a,y)]$	4. ERule(1)
$Triangle(b) \wedge (\forall x)\neg LeftOf(x,b)$	5. ERule(3)
$Triangle(b)$	6. $\wedge$-prop(5)
$(\forall x)\neg LeftOf(x,b)$	7. $\wedge$-prop(5)
$Triangle(b) \Rightarrow LeftOf(a,b)$	8. ARule(4)
$LeftOf(a,b)$	9. $\Rightarrow$-prop(6,8)
$\neg LeftOf(a,b)$	10. ARule(7)
Contradiction. 9,10	

Je kunt verifiëren dat elke stap correct is zonder zelfs te weten wat deze zinnen betekenen. Dus kan ook een computer dit soort bewijzen verifiëren. Daarentegen, om zo'n bewijs zelf op te stellen is het voor ons meestal noodzakelijk om bij de les te zijn en goed te begrijpen wat de zinnen betekenen, en zelf eerst een argument op te stellen waarom deze theorie inconsistent is. Het formele bewijs zal dan een herwerking van dat argument zijn.

In dit geval drukt de hypothese uit dat er een Geo-figuurtje bestaat (noem het a) dat links staat van elke driehoek. De te bewijzen zin specifieert dat er geen driehoek is waar geen enkel object links van voorkomt, t.t.z. elk object ligt minstens op gelijke hoogte of rechts ervan.

Bovenstaand formeel bewijs loopt geheel volgens de lijnen van het volgende gedetailleerd maar informeel bewijs uit het ongerijmde:

Gegeven: er bestaat een Geo-figuurtje dat links staat van elke driehoek.

T.B. : er is geen driehoek waar geen enkel object links van ligt.

Bew. (uit het ongerijmde)

Gegeven is dat een figuurtje links staat van elke driehoek (stap 1). Noem dat figuurtje a (stap 4).

Veronderstel dat het te bewijzen onwaar is (stap 2). Dus, er is een driehoek waar geen enkel object links van ligt (stap 3). Noem deze driehoek b (stap 5).

b is dus een driehoek (stap 6). Dus staat a links van b (stap 8+9).

Maar b heeft de eigenschap dat geen enkel object er links van staat (stap 7). Dus staat a niet links van (b) (stap 10).

Contradictie. Q.E.D.

Gevalsonderscheiding-regel Een gevalsonderscheiding maken we door onderaan een tak twee nieuwe deelrijen te openen, één beginnend met een formule A, de andere met $\neg A$.

Hier mag A eender welke zin zijn, maar meestal zal A een deelzin zijn van een zin die reeds voorkomt in de tak.

Voorbeeld 3.5.3. Toon aan met een KE-bewijs dat

$$(\exists x)(\forall y)[Triangle(y) \Rightarrow Square(x)]$$

een logisch gevolg is van

$$(\exists x)Triangle(x) \Rightarrow (\exists x)Square(x)$$

KE-bewijs:

$(\exists x)Triangle(x) \Rightarrow (\exists x)Square(x)$	1. Hypothesis
$\neg\{(\exists x)(\forall y)[Triangle(y) \Rightarrow Square(x)]\}$	2. NegatedConclusion
Case A	
$(\exists x)Triangle(x)$	3. Case A
$(\exists x)Square(x)$	5. $\Rightarrow$prop(1,3)
$Square(a)$	6. ERule(5)
$\neg(\forall y)[Triangle(y) \Rightarrow Square(a)]$	7. ARule(2)
$\neg(Triangle(b) \Rightarrow Square(a))$	8. ERule(7)
$Triangle(b)$	9. PropRule(8)
$\neg Square(a)$	10. PropRule(8)
$Contradiction.10, 6$	
Case A	
$\neg(\exists x)Triangle(x)$	4. Case B
$\neg\{(\forall y)[Triangle(y) \Rightarrow Square(a)]\}$	12. ARule(2)
$\neg(Triangle(b) \Rightarrow Square(a))$	13. ERule(12)
$Triangle(b)$	14. PropRule(13)
$\neg Square(a)$	15. PropRule(13)
$\neg Triangle(b)$	16. ARule(4)
Contradiction: 14,16	

Merk op dat we voor een bewijs van een logisch gevolg onderscheid maken tussen theorie-stappen die een hypothese introduceren en een theorie-stap die de negatie van het gevolg introduceert.

Voorbeeld 3.5.4. We construeren een bewijs dat de formule $(\forall x)P(x, y) \Rightarrow (\exists x)P(x, y)$ uit Oefening 3.1.1 logisch waar is.

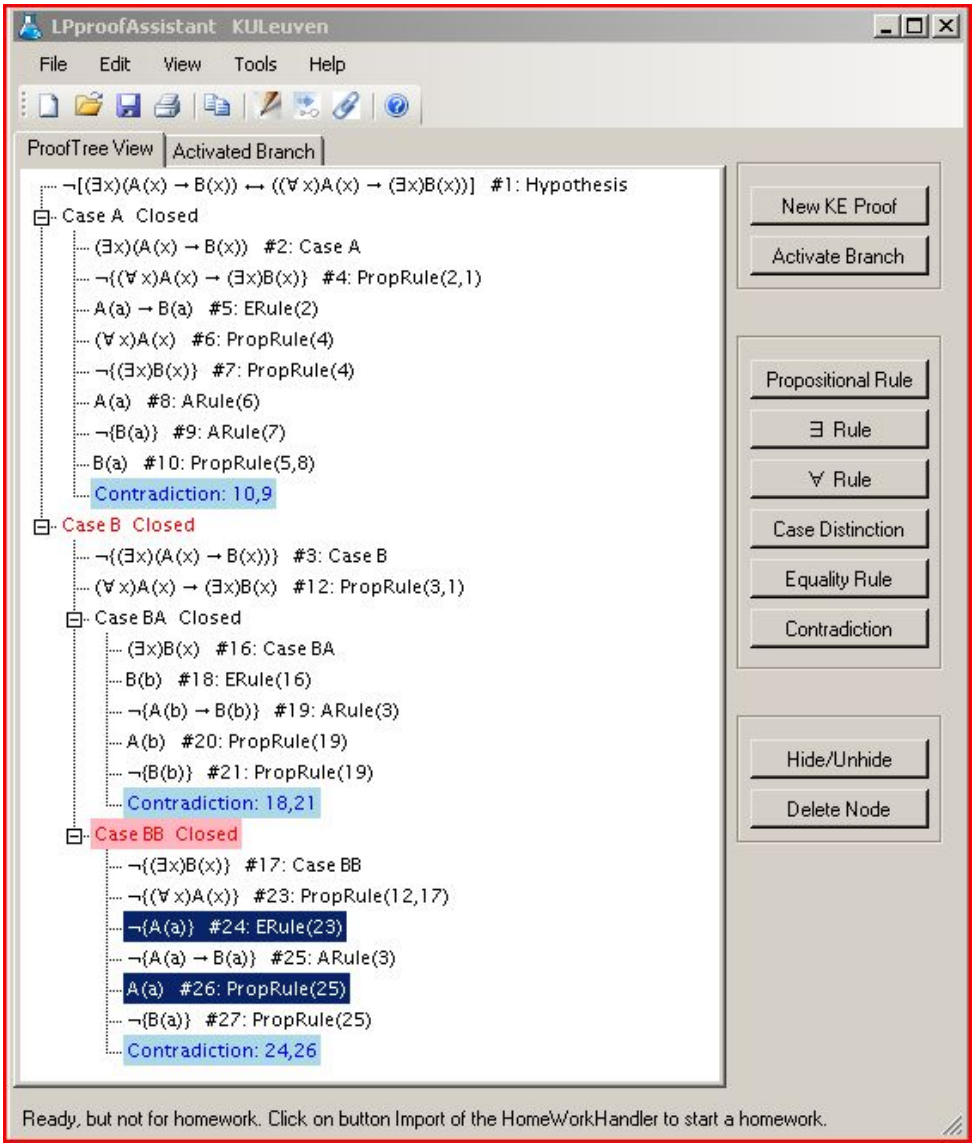
$\neg(\forall y)[(\forall x)P(x, y) \Rightarrow (\exists x)P(x, y)]$	1. NegatedConclusion
$\neg[(\forall x)P(x, c_1) \Rightarrow (\exists x)P(x, c_1)]$	2. Erule(1)
$(\forall x)P(x, c_1)$	3. $\neg\Rightarrow$-prop(2)
$\neg(\exists x)P(x, c_1)$	4. $\neg\Rightarrow$-prop(4)
$P(c_1, c_1)$	5. Arule(3)
$\neg P(c_1, c_1)$	6. Arule(4)
Contradictie(5,6)	

Definitie 3.5.1. Een tak van een KE-redenering is *gesloten* indien er een contradictie in voorkomt: een zin A en ook $\neg A$.

Een KE-redenering uit T is een KE-bewijs van inconsistentie van T als elke tak gesloten is.

Het tool LPproofAssistant van de software LogicPalet is een handig hulpmiddel om KE-bewijzen te construeren. Er zijn knoppen voorzien voor elk van de KE-regels. Daarop klikken past de regel toe en schrijft automatisch het resultaat. Er is een foutmelding wanneer de regel niet toepasbaar is waardoor het onmogelijk wordt om een foutief bewijs te construeren.

In Figuur 3.3 staat een KE-bewijs dat met de proof-assistant van LogicPalet is uitgewerkt.



Figuur 3.3: Een KE-bewijs van $(\exists x)(A(x) \Rightarrow B(x)) \Leftrightarrow ((\forall x)A(x) \Rightarrow (\exists x)B(x))$

Correctheid en volledigheid.

Propositie 3.5.1. *Als een logische theorie T een model $\mathfrak{A}$ heeft, dan geldt voor elke KE-redenering dat er minstens 1 tak bestaat en een uitbreiding $\mathfrak{A}_1$ van $\mathfrak{A}$ die alle nieuwe constanten c in die tak interpreteert, zodat alle zinnen van de tak waar zijn in $\mathfrak{A}_1$.*

Bew. Dit kan gemakkelijk door inductie op de lengte van het KE-redenering bewezen worden. Veronderstel dat $\mathfrak{A}$ een model is van T.

De stelling is voldaan voor KE-redeneringen van lengte 0. Immers, neem $\mathfrak{A}_1 = \mathfrak{A}$. Een dergelijke KE-redenering bevat geen enkele formule, en dus zijn "alle" formules ervan voldaan in $\mathfrak{A}_1$.

Veronderstel dat de stelling voldaan is voor KE-redeneringen van lengte n-1 (dit is de inductiehypothese). Dan bewijzen we dat het ook geldig is voor KE-redeneringen van lengte n.

Uit het feit dat de inductiehypothese voldaan was voor het bewijs tot en met stap $n-1$ weten we dat er een tak `t` en een structuur $\mathfrak{A}_1$ bestaan die alle formules in de tak waar maakt. Bij stap n is het ofwel de tak `t` die uitgebreid wordt, ofwel een andere. Als een andere tak uitgebreid wordt, dan geldt natuurlijk nog altijd dat $\mathfrak{A}_1$ alle formules van tak `t` waar maakt. Dus veronderstel dat het tak `t` is die uitgebreid wordt.

Het bewijs is door een gevalstudie van stap n. We bekijken niet alle gevallen.

- Veronderstel dat stap n een theorie-stap is en een zin $A \in T$ toevoegt aan `t`. Aangezien $\mathfrak{A}_1$ een uitbreiding is van $\mathfrak{A}$ geldt dat $\mathfrak{A}_1 \models A$. Dus geldt de inductiehypothese voor n.

- Veronderstel dat stap n een gevalsonderscheid is voor een formule A. Ofwel geldt $\mathfrak{A}_1 \models A$, ofwel $\mathfrak{A}_1 \models \neg A$, dus is de inductiehypothese voldaan voor één van de twee deeltakken.

- Veronderstel dat stap n een toepassing is van een propositionele propagatie-regel. Aangezien achter elke propagatie-regel $A, B \longrightarrow C$ een tautologie schuilt $A \wedge B \Rightarrow C$ en aangezien $\mathfrak{A}_1 \models A \wedge B$, geldt $\mathfrak{A}_1 \models C$. Hetzelfde geldt voor propagatie-regels $A \longrightarrow C$. Dus is de inductiehypothese voor n voldaan.

- Erule: veronderstel dat `t` een formule $(\exists x)A[x]$ bevat en $A[c]$ wordt toegevoegd, met c een nieuwe constante. Merk op dat niet geldt dat $\mathfrak{A}_1 \models A[c]$, immers, c is een nieuwe constante en dus niet geïnterpreteerd door $\mathfrak{A}_1$ (t.t.z. $c^{\mathfrak{A}_1}$ bestaat niet). Maar we kunnen $\mathfrak{A}_1$ wel uitbreiden voor c. Aangezien deze tak voldaan was in $\mathfrak{A}_1$, geldt $\mathfrak{A}_1 \models (\exists x)A[x]$, dus bestaat $d \in D_{\mathfrak{A}}$ zodat $\mathfrak{A}_1[x : d] \models A[x]$. We breiden $\mathfrak{A}_1$ uit door deze d te kiezen als interpretatie voor de nieuwe c. Dus, stel $c^{\mathfrak{A}_1} = d$. Met deze uitbreiding geldt $\mathfrak{A}_1 \models A[c]$.

- Arule: uit een formule $(\forall x)A[x]$ wordt $A[t]$ afgeleid, met t een term die enkel functie- en constante-symbolen die in T of in de tak voorkomen en door $\mathfrak{A}_1$ geïnterpreteerd worden. Veronderstel dat $t^{\mathfrak{A}_1} = d$, dan geldt $\mathfrak{A}_1[x : A] \models A[x]$ en bijgevolg geldt $\mathfrak{A}_1 \models A[t]$. Opnieuw is de inductiehypothese voldaan voor n.

- Enz.

Q.E.D.

Oefening 3.5.2. *De enige soort stap waarvan we de correctheid niet bewezen hebben zijn de gelijkheidsstappen. Bewijs de correctheid ervan.*

Stelling 3.5.1 (correctheid)**.** *Indien er een KE-redenering uit $T = \{A_1, \ldots, A_n\}$ bestaat waarvan alle takken gesloten zijn dan is T logisch inconsistent. Als T van de vorm is $\{A_1, \ldots, A_n, \neg B\}$ dan is B een logisch gevolg van $\{A_1, \ldots, A_n\}$. Die KE-redenering wordt dan een KE-bewijs van B uit $\{A_1, \ldots, A_n\}$ genoemd.*

Bew. Veronderstel dat T een gesloten KE-bewijs heeft. Mocht T consistent zijn, en dus een model hebben, dan volgt uit Propositie 3.5.1 dat er een tak is en een structuur $\mathfrak{A}_1$ die alle formules op die tak waar maakt. Dat kan niet want elke tak bevat een contradictie, dus is T niet consistent. Q.E.D.

Stelling 3.5.2 (Gödels volledigheidsstelling)**.** *Indien een eindige theorie T inconsistent is, dan bestaat er een KE-bewijs van de inconsistentie. Indien B een logisch gevolg is van T dan bestaat een KE-bewijs voor B uit T.*

Dit is een hele straffe stelling, waar men tientallen jaren naar heeft gezocht. Kurt Gödel gaf een wiskundig bewijs van deze stelling in 1930. Gödel bewees dit voor een ander soort van formele bewijzen, maar zijn methode kan aangepast worden voor KE-bewijzen.

Semi-onbeslisbaarheid van de predikatenlogica. We hebben in dit hoofdstuk een aantal nieuwe redeneertaken bestudeerd:

Bereken of theorie T inconsistent is.

Bereken of zin A een logisch gevolg is van theorie T.

Bereken of zin A inconsistent is met theorie T.

Merk op: als we één taak kunnen oplossen, dan kunnen we ze alle drie oplossen.

Definitie 3.5.2. Deze redeneervormen noemt men **deductief redeneren** of kortweg **deductie**.

Een belangrijke vraag is nu of er algoritmes bestaan die deze vragen kunnen oplossen. We hebben natuurlijk zonet de KE-bewijsmethode gezien, maar dit is wel een formele redeneermethode, maar het is geen algoritme om automatisch bewijzen te genereren.

Elk van deze drie taken heeft ook een duale taak:

Bereken of theorie T consistent is.

Bereken of zin A geen logisch gevolg is van theorie T.

Bereken of zin A consistent is met theorie T.

Een "ja" antwoord op een vraag betekent een "nee" antwoord op zijn duale vraag en omgekeerd. Je zou dus verwachten dat als we één probleem kunnen oplossen dat we ook zijn duale probleem kunnen oplossen, en dus alle zes. Merkwaardig genoeg blijkt dit niet helemaal het geval te zijn!

De volgende stelling volgt uit Gödels volledigheidsstelling.

Stelling 3.5.3. *Er bestaat een computerprogramma*[4] *dat bij input van zinnen* $T = \{A_1, \ldots, A_n\}$ *stopt na een eindig aantal stappen met output "*inconsistent*" indien* T *inconsistent is en niet stopt of antwoordt met output "*consistent*" als* T *consistent is.*

Dus, als dit programma eindigt dan is het antwoord correct, en als de theorie inconsistent is zal het ook eindigen. Als de theorie consistent is zal het mogelijks niet eindigen.

Bew. We hoeven ons van efficiëntie gelukkig niets aan te trekken. Veronderstellen we dat er een oneindige lijst is van nieuwe constanten $a, c_1, c_2 \ldots$. Bij elke Arule-stap waarbij voorheen geen constanten in de tak voorkwamen wordt de nieuwe constante a gebruikt en bij elke Erule-stap wordt steeds de eerste ongebruikte constante c_n gebruikt. Dus, bij de n-de Erule-stap gebruiken we altijd c_n. Elk KE-redenering kan op deze manier herbouwd worden.

We definiëren de grootte van een KE-redenering als de som van de lengtes van alle formules die erin voorkomen, en de lengte ervan als het aantal formules die ze bevat, dus het aantal redeneerstappen in de KE-redenering. De lengte is kleiner dan de grootte, dat spreekt vanzelf.

Veronderstel dat de KE-redenering K grootte kleiner dan of gelijk aan n heeft. Welke symbolen kunnen voorkomen in K? De symbolen uit T en een aantal nieuwe constanten die door Erule-stappen werden geïntroduceerd. De lengte van K is kleiner dan n, en dus behoren alle nieuwe constanten tot de verzameling $\{c_1, \ldots, c_n\}$. Dus is de verzameling van symbolen die voorkomen in K zeker een deelverzameling van de verzameling van alle symbolen uit T en uit $\{c_1, \ldots, c_n\}$.

We zien dus dat een KE-redenering van grootte n kan gezien worden als een string van lengte n bestaande uit een eindig aantal symbolen. Er zijn maar een eindig aantal zo'n strings, en dus maar een eindig aantal KE-redeneringen van grootte n. Bekijk volgend programma:

- Voor $n = 1, 2, 3, \ldots$ doe:
  - Genereer elke willekeurig string S van lengte n bestaande uit symbolen uit T en $\{c_1, \ldots, c_n\}$.
  - Voor elke S doe:
    - test of het een verzameling van zinnen is;
    - zo ja, test of het overeenkomt met een KE-redenering voor T;
    - zo ja, test of het een KE-bewijs van de inconsistentie van T;

* zo ja, stop met antwoord "inconsistent".
* In alle andere gevallen, neem de volgende string.

De enige manier voor dit programma om te eindigen is als het een KE-bewijs van de inconsistentie van T vindt. Door de correctheidsstelling volgt dat T dan inderdaad inconsistent is en dus is het antwoord "inconsistent" correct. Het programma eindigt nooit als de theorie consistent is.

Als de theorie inderdaad inconsistent is, dan bestaat volgens Gödels volledigheidsstelling een KE-bewijs van zijn inconsistentie, bv. van grootte N. Dus zal dit bewijs gevonden worden door ons programma, namelijk wanneer het $n = N$ bereikt. En dan zal het programma ook stoppen. Q.E.D.

Het algoritme in het bewijs is buitengewoon naïef en zou in de praktijk nooit werken. Maar hier zijn we nog niet geïnteresseerd in zo'n practische zaken zoals efficiëntie. Straks zullen we daarover iets meer zeggen.

Kan er een algoritme gevonden worden dat ook stopt als T consistent is? Het antwoord is negatief.

Stelling 3.5.4 (Stelling van Church (1936))**.** *Er bestaat geen eindigend computerprogramma dat de redeneertaak "*Bereken of een eindige input theorie T consistent is*" correct oplost, t.t.z., dat bij input T altijd stopt na een eindig aantal stappen met output:*

- *"consistent" indien T consistent is;*
- *"inconsistent" indien T inconsistent is.*

Dit betekent natuurlijk ook dat er geen eindigend algoritme bestaat dat voor input T en zin A antwoordt "*A is een logisch gevolg van T*" indien dat het geval is, en anders antwoordt "*A is geen logisch gevolg van T*".

De stelling betekent ook dat voor de drie duale redeneertaken (is T consistent, is A geen logisch gevolg van T, is A consistent met T?) geen programma bestaat dat voor elke input in eindige tijd het antwoord vindt indien dat antwoord positief is. Daarentegen bestaat er wel een dergelijk programma dat altijd zal eindigen indien het antwoord negatief is.

Men vat de stelling van Church ook wel eens samen door te zeggen : "*Predikatenlogica is onbeslisbaar*". Daarmee bedoelt men dus exact wat in deze stelling uitgedrukt staat. De logicus Church[5] heeft dit resultaat voor het eerst bewezen.

Samengevat stellen we het volgende vast. Voor de redeneertaak "*Is een eindige theorie T inconsistent?*" bestaat er wel een algoritme dat correct "inconsistent" antwoordt en eindigt indien T inconsistent is, en anders "consistent" antwoordt of niet eindigt. Deze eigenschap wordt ook kortweg de *semi-beslisbaarheid van predikatenlogica* genoemd.

[5]Let wel, hij heeft ook bewezen dat er geen computerprogramma kan bestaan die voor elke zin kan beslissen of deze waar is in de natuurlijke getallen. Deze stelling heet: de onbeslisbaarheid van de natuurlijke getallen. Hiernaar werd verwezen in Hoofdstuk 2. Verwar deze twee stellingen niet!

Stelling 3.5.5. *Predikatenlogica is onbeslisbaar maar wel semi-beslisbaar.*

Consistentie bewijzen met KE-redeneringen Het algoritme in het bewijs van Stelling 3.5.3 zal nooit eindigen indien T consistent is. Er kunnen immers steeds nieuwe gevalsonderscheiden gemaakt worden op steeds andere formules, of dezelfde redeneerstap kan oneindig keer herhaald worden. Zoiets moet in een practische implementatie natuurlijk vermeden worden.

Een *strategie* bestaat uit het opleggen van beperkingen aan hoe KE-redeneringen gevormd mogen worden. Zo was het opleggen van de beperking dat bij de $n$de toepassing van de Erule de constante c_n gebruikt moet worden een strategie-regel.

Hier zijn er nog een paar zinnige strategie-regels:

- Leidt nooit een zin af die al eerder voorkwam in de tak.
- Pas nooit een gevalsonderscheiding toe voor een formule A als A of $\neg A$ al eerder voorkwam in de tak.
- Pas de ERule maar 1 keer toe, dus voor slechts één constante c.

Elke bijkomende regel vermindert het aantal KE-redeneringen, en maakt de procedure om inconsistentie te bewijzen potentieel efficiënter.

Definitie 3.5.3. We noemen een strategie S *volledig* indien geldt: elke KE-redenering voor een inconsistente theorie T kan vervolledigd worden met strategie S tot een KE-bewijs van inconsistentie.

Voor bepaalde strategieën kan het zijn dat in bepaalde takken, hoewel ze niet gesloten zijn (dus geen contradictie bevatten), er toch geen enkele redeneerstap nog mogelijk is. Als de strategie volledig is betekent dat dat deze KE-redenering dus niet meer uitgebouwd kan worden tot een KE-bewijs van inconsistentie. Bijgevolg is de input theorie T consistent!! Op die manier kunnen we dus soms toch consistentie bewijzen.

Stelling 3.5.6. *Als een KE-redenering uit theorie T werd gebouwd met een volledige strategie en deze redenering heeft een volledige maar niet gesloten tak, dan is T consistent.*

Een voorbeeld van een strategie die volledig is indien er geen gelijkheden of functiesymbolen voorkomen in T bestaat uit alle regels die we tot nog toe zagen plus de volgende:

- Propagatieregels en Erule toepassen van zodra mogelijk.
- Bij toepassing van de $\forall$-regel alleen maar een term t substitueren die reeds voorkomt in de huidige tak, tenzij er nog geen termen in voorkomen.

- De huidige tak moet eerst gesloten zijn vooralleer in een andere tak te werken.
- Gevalsonderscheiding alleen maar toepassen als drie voorwaarden voldaan zijn:
  - als er geen andere regels kunnen toegepast worden in de huidige tak
  - enkel op een deelzin A van een zin in dezelfde tak;
  - en alleen maar als dit een zin levert die vervolgens kan gebruikt worden als tweede hypothese voor een propagatie-regel.
    Merk op dat in elke propagatieregel die 2 hypotheses heeft, de tweede hypothese een deelformule is van de eerste hypothese (of de negatie van zo'n deelformule).

Men kan wiskundig bewijzen dat deze strategie volledig is tenminste als, zoals gezegd, er geen gelijkheden (of ongelijkheden) of functiesymbolen in T voorkomen. We gaan daar niet verder op in.

De combinatie van al deze regels zorgt ervoor dat heel wat KE-bewijzen volledige maar niet gesloten takken zullen bevatten. Het is bovendien zo dat met zo'n tak een model geconstrueerd kan worden uit de literals die voorkomen in die tak.

Voorbeeld 3.5.5. Veronderstel dat we zouden willen bewijzen dat $\neg(P \Rightarrow Q) \Leftrightarrow (R \vee Q)$ logisch waar is. We doen dat door te bewijzen dat de negatie onwaar is.

$\neg[\neg(P \Rightarrow Q) \Leftrightarrow (R \vee Q)]$	1. Theorie
Case A	
$R \vee Q$	2. Case A
$(P \Rightarrow Q)$	3. $\neg\Leftrightarrow$-prop(1,2)
Case A.A	
$\neg Q$	4. Case A.A
R	5. $\vee$-prop(2,4)
$\neg P$	6. $\Rightarrow$-prop(3,4)
Tak volledig; niet gesloten	

Op dit punt kan met onze strategie geen enkele uitbreidingsregel meer toegepast worden. Door het vinden van deze volledige maar niet gesloten tak, hebben we aangetoond dat de zin (1) niet inconsistent is en bijgevolg heeft het geen zin om verder te gaan. Bij inspectie van deze tak blijken we bovendien ook een structuur gevonden te hebben die voldoet aan (1): dat is de structuur $\mathfrak{A}$ waarin de literals in de tak waar zijn, namelijk $\neg P, \neg Q, R$.

In het algemeen kan de KE-methode gebruikt worden als een efficiënte methode om modellen van propositionele formules te construeren, efficiënter dan het construeren van een waarheidstabel.

Oefening 3.5.3. *Construeer een KE-redenering met een volledige doch niet gesloten tak voor de zin* $[(P \wedge R) \Rightarrow \neg Q] \wedge (Q \vee R)$.

3.6 Automated theoremproving (ATP)

In ATP worden automatische theoremprovers ontwikkeld, methodes die gebruikt kunnen worden om inconsistentie, logische waarheid of logisch gevolg aan te tonen.

Er bestaat een grote diversiteit aan formele bewijsmethoden. KE-bewijzen zien er heel anders uit dan de bewijsmethodes die ontwikkeld werden ten tijde van Gödel. In het college Artificiële intelligentie van 2BA wordt de *resolutiemethode* besproken die dateert van 1965 en wiens computerimplementatie soms sneller werkt maar waarvan de geleverde formele bewijzen veel minder begrijpbaar zijn voor ons, mensen.

Hedendaagse theoremprovers zijn zeer complexe systemen waarin vaak veel verschillende methodes en technieken gecombineerd worden. Bv. het systeem Spass dat gebruikt wordt in LogicPalet is gebaseerd op een combinatie van de resolutiemethode met een methode die zeer verwant is aan de KE-methode. De geleverde formele bewijzen zijn echter moeilijk begrijpbaar voor mensen. Naast SPASS bestaan er nog veel andere systemen zoals bijvoorbeeld OTTER of Vampire.

Eén van de eerste successen van theoremprovers was het oplossen van het Robbins probleem in Boolse algebra, een open wiskundig probleem sinds 1933, dat in 1996 in 8 dagen werd opgelost door de theorem prover EQP en daarbij de New York Times haalde. Maar hoewel er sinds 1996 nog zeer grote vooruitgang geboekt werd is, in tegenstelling tot andere intellectuele activiteiten zoals bv. schaken, het bewijzen van wiskundige stellingen vooralsnog een domein waar mensen, wiskundigen, veruit superieur zijn aan computers.

Toch worden theoremprovers steeds vaker ingezet voor misschien minder moeilijke maar vaak voorkomende deelproblemen in de context van informatica-toepassingen.

Veronderstel dat een bedrijf een grote databank wil gaan opstellen. In de ontwerp-fase wordt een vocabularium en een verzameling integriteitsbeperkingen opgesteld. Daarin sluipen gemakkelijk allerlei kleine fouten die door een theoremprover ontdekt kunnen worden. Bv. iemand wil specifiëren dat een student geslaagd is voor een examen asa zijn of haar score een slaagscore is, en schrijft per abuus:

$$(\forall x)(\forall c)(\forall s)(Grade(x,c,s) \wedge PassingGrade(s) \Leftrightarrow SucceedsCourse(x,c))$$

Hier wordt dezelfde fout gemaakt als in het "koorts"-voorbeeld in Sectie **??**. Deze zin impliceert dat niemand slaagt voor een cursus (want anders zijn alle objecten in het universum slaagscores). Dergelijke fouten zijn bijna onvermijdbaar - iedereen maakt ze. Een automatische theoremprover zal de inconsistentie van zo'n theorie ontdekken. Theoremprovers kunnen ook gebruikt worden om aan te tonen dat een integriteitsbeperking een logisch gevolg is van andere. Of om aan te tonen dat een bepaalde transactie de integriteit van een databank bewaart, enz.

Theoremprovers hebben toepassingen in velden zoals:

- Deductieve databanken (zie Hoofdstuk 4).
- Software-verificatie (zie Hoofdstuk 4). Bv. theoremprovers worden door Siemens gebruikt in toepassingen zoals de ontwikkeling van de software (confer Metro 14 in Hoofdstuk 1).
- Hardware-verificicatie. Het ACL2 systeem werd gebruikt om de correctheid te bewijzen van floating point deling in AMD's PENTIUM-like AMD5K86 microprocessor. (Dit was overigens kort nadat Intel een miljardenverlies leedt doordat een fout in hun floating point processor was ontdekt, waarna ze al die processoren gratis moesten vervangen.)
- Correctheids-bewijzen van programmas.
- Programmeertalen die volledig gebaseerd zijn op predikatenlogica en automated reasoning, bijvoorbeeld Prolog.
- Theoremproving wordt ook gebruikt in het semantisch web.

Een goed vertrekspunt om meer informatie tevinden is via de webpagina: `http://www.cs.miami.edu/~tptp/OverviewOfATP.html`.

3.7 Samenvatting

In dit hoofdstuk werden fundamentele concepten van het denken bestudeerd: logische waarheid, logisch gevolg en logische (in)consistentie. We bestudeerden een aantal wetten van correct denken, in de vorm van logisch ware equivalenties. We hebben gezien hoe logische waarheid of gevolg bewezen kan worden, op een informele manier. We hebben ook een formele bewijsmethode gezien, de KE-bewijsmethode die correct en volledig is.

Vervolgens werd aandacht besteed aan een aantal nieuwe redeneertaken:

- bepalen of een zin logisch waar is
- bepalen of een zin inconsistent is
- bepalen of een zin een logisch gevolg is van een theorie.

Deze vormen van redeneren worden vaak *deductief redeneren* genoemd. Elk van deze taken heeft een duale taak (niet logisch waar, consistent, geen logisch gevolg).

In propositielogica blijken al deze taken oplosbaar (propositielogica is beslisbaar!). In predikatenlogica echter blijken deze taken maar half oplosbaar. Er bestaan wel methodes die in eindige tijd het antwoord "ja" op deze vragen kan produceren, maar geen enkele correcte methode kan in eindige tijd antwoorden met "ja" of "nee". Dit noemt men de *onbeslisbaarheid* en *semi-beslisbaarheid* van predikatenlogica.

Let wel, in feite is de uitspraak "*predikatenlogica is onbeslisbaar*" misleidend, omdat deze zin niet expliciet maakt over welke redeneertaak het gaat. Vergeet niet dat andere redeneertaken (bv. berekenen van waarheid of queries in een structuur, berekenen van modellen met een gegeven eindig domein, enz.) ongetwijfeld veel meer dagdagelijkse toepassingen hebben in de informatica dan het bewijzen of een formule logisch waar is of inconsistent of een logisch gevolg. En er bestaan wel eindigende en zelfs zeer efficiënte programma's om die taken te verrichten voor predikatenlogica, of voor talen die daarvan afgeleid zijn. Databanksystemen zijn voorbeelden daarvan.

Historisch gezien is er al zeer veel te doen geweest rond de onbeslisbaarheid van predikatenlogica. Er zijn nogal wat mensen waarvoor predikatenlogica bijna *synoniem* is voor de studie van deductief redeneren[6]. Daar is een historische reden voor: herinner u uit Hoofdstuk 1 dat predikatenlogica ontstond uit de studie van het wiskundig, deductief redeneren. Aangezien gebruikers van softwaresystemen nood hebben aan eindigende programma's die het antwoord op onze vragen vinden, lijkt predikatenlogica voor mensen met deze visie onbruikbaar voor practisch gebruik in de informatica! Maar dit is een te enge visie op predikatenlogica, en de conclusie strookt gewoon niet met de feiten: de vele toepassingen van andere vormen van redeneren, en de efficiëntie waarmee deze geïmplementeerd kunnen worden.

[6]Predikatenlogica wordt door sommigen ook wel *deductive logic* genoemd

Hoofdstuk 4

Modelleren en redeneren in informatica-toepassingen

We bestuderen een aantal informatica-domeinen. Welke informatie is daarin beschikbaar? Hoe kunnen we deze informatie voorstellen in logica? Welke informaticaproblemen kunnen opgelost worden met die informatie? Welke vorm van redeneren hebben we daarbij nodig? Dit zijn de vragen die we in dit hoofdstuk bestuderen.

De droom hierachter is: stel de informatie in uw toepassingsgebied voor in logica, en laat uw probleem oplossen door een logisch redeneerssysteem! Dat is het doel van een deel van Artificiële Intelligentie. Het is ook de "dream, the quest, the holy grail" waarover Bill Gates het had in het citaat pagina 8 van deze cursus.

We zullen ook een eigenschap zien die logica (en formele modelleertalen in het algemeen) onderscheidt van alle programmeertalen: de mogelijkheid om verschillende informaticaproblemen op te lossen op basis van *dezelfde* logische theorie.

4.1 Modelleren in logica: inleiding

De eerste stap bij het gebruik van logica in een toepassing is altijd dezelfde: het modelleren van informatie over het toepassingsgebied in een logische theorie.

Om dergelijke informatie te modelleren in formele talen zoals predikatenlogica zijn er twee grote stappen die ondernomen moeten worden:

1. het kiezen van een vocabularium Σ om "objecten" of "concepten" uit het toepassingsgebied te noteren;
2. het uitdrukken van een aantal informele eigenschappen over het toepassingsgebied in formele zinnen over Σ.

Laat ons dat even illustreren in de context van een eenvoudig domein: namelijk de verschillende familierelaties. Waar we over willen spreken is over kinderen en ouders, broers, zussen, neven en nichten, nonkels en tantes, grootouders en voorouders, echtgenoten en echtgenotes. We moeten ook een onderscheid maken tussen mannen en vrouwen. Misschien (maar niet noodzakelijk) willen we ook spreken over bepaalde specifieke mensen, zoals mezelf en mijn directe familieleden.

Nu kiezen we hiervoor een vocabularium waarin elk symbool overeenkomt met één van die relaties of personen. Moeilijk is dat hier niet, bv. $Man/1, Vrouw/1, OuderVan/2, GetrouwdMet/2$, $ZoonVan/2, DochterVan/2, BroerVan/2, OomVan/2, TanteVan/2, NeefVan/2, \ldots, Marc$, $Roel$, $Edgar, An, \ldots$. Toch zijn er verschillende mogelijkheden:

- We hebben niet al die symbolen nodig want het is duidelijk dat sommige relaties uitgedrukt of *gedefinieerd* kunnen worden in termen van andere. Bv. een zuster is een vrouw waarmee je twee gemeenschappelijke ouders hebt. We zouden kunnen kiezen voor een minimale verzameling van symbolen en alle andere relaties uitdrukken door middel van formules, ofwel kunnen we symbolen introduceren voor elke relatie maar dan moeten we de logische verbanden tussen die symbolen uitdrukken, t.t.z. dan moeten we de *definities* van die symbolen uitdrukken met logische formules.
- Maar we hadden ook andere symbolen kunnen kiezen. Bv. aangezien iedereen een unieke vader en moeder heeft, hadden we ook functiesymbolen $Vader/1 :, Moeder/1 :$ kunnen kiezen.

Oefening 4.1.1. *Laat zien dat de volgende symbolen volstaan om alle andere familierelaties te beschrijven:* $Man/1, Vrouw/1, OuderVan/2, GetrouwdMet/2$. *Schrijf een formule* $A[x, y]$ *met vrije variabelen* x, y *zodat* $\{(x, y) : A[x, y]\}$ *een relatievoorschrift is voor de volgende relaties:*

- x *is moeder, vader, stiefmoeder, schoonmoeder van* y*;*
- x *is zoon, dochter, stiefzoon van* y*;*
- x *is zuster, broer, halfbroer, stiefbroer van* y*;*
- x *is oom, tante van* y*;*
- x *is grootouder van* y*;*
- x *is voorouder van* y*.*

Oefening 4.1.2. *Neem het volledige vocabularium en druk logische verbanden uit met de basisconcepten. Bv.*

$$(\forall x)(\forall y)(StiefMoederVan(x, y) \Leftrightarrow \begin{array}{l} Vrouw(x) \wedge \neg OuderVan(x, y) \wedge \\ (\exists z)(GetrouwdMet(x, z) \wedge OuderVan(z, y))) \end{array}$$

Hierbij kun je de formules uit de vorige oefening natuurlijk hergebruiken.

Oefening 4.1.3. *Herformuleer een paar van de logische formules van de vorige oefeningen gebruik makend van* $Moeder/1 :, Vader : /1$ *in plaats van* $OuderVan/2$.

Het resultaat van deze eerste fase is dus een vocabularium met formele symbolen waaraan we een specifieke betekenis hechten (bv. $Man/1$ betekent mannelijk). Bijgevolg kunnen we elke logische zin nu gaan interpreteren als een Nederlandse zin over het toepassingsdomein waarin we geïnteresseerd zijn.

Wat betekent dat nu precies, dat we "betekenis" hechten aan een symbool? Betekent dit dat we een structuur geven waarin de waarde van dit symbool gegeven is? Absoluut niet. Als je kijkt naar de meeste voorbeelden in de cursus, inclusief dat van de familierelaties, dan hebben we helemaal geen concreet universum gespecificieerd noch waardes voor de symbolen. Wat we eigenlijk doen is: we associëren de formele symbolen met informele concepten uit het toepassingsdomein. Door het verband te leggen tussen de formele symbolen en de informele concepten kunnen we wiskundige structuren van ons vocabularium zien als wiskundige voorstellingen van mogelijke of onmogelijke toestanden van dat toepassingsdomein.

Bv. een structuur $\mathfrak{A}$ met domein $\{d\}$ en $OuderVan^{\mathfrak{A}} = \{(d,d)\}$ stelt een situatie voor met maar 1 mens die zijn eigen ouder is. Dat is onmogelijk. Bv. als we P kiezen als "de trein is te laat" en Q als "John mist zijn bus", dan komen de 4 wiskundige structuren van $\Sigma = \{P, Q\}$ overeen met de 4 situaties waarin de trein al dan niet te laat is en John al dan niet zijn bus mist. Eén van die situaties is onmogelijk, namelijk die waarin John zijn bus niet mist terwijl de trein te laat is. Deze onmogelijke situatie komt overeen met de structuur $\{P\}$.

De volgende fase van het modelleren bestaat erin om logische zinnen te formuleren waarvan we weten dat ze waar zijn in het toepassingsdomein. Maar dat volstaat niet, want anders zouden we tautologieën kunnen uitschrijven. Het heeft maar zin om een nieuwe zin toe te voegen aan onze theorie als er een klasse van onmogelijke structuren bestaat, waarin elke zin die we tot nog toe hadden waar is maar waarin de nieuwe zin onwaar is. M.a.w., de nieuwe zin sluit deze klasse van onmogelijke structuren uit. Bv., door $P \Rightarrow Q$ toe te voegen elimineren we de onmogelijke wereld waarin de trein te laat is maar John zijn bus niet mist. En zo gaan we door totdat tenslotte de modellen overeen komen met wat we beschouwen als de mogelijke situaties van het toepassingsdomein. Dat is een kunst, het is niet altijd zo eenvoudig, maar je kunt het leren, en dat is één van de bedoelingen van deze cursus.

Oefening 4.1.4. *Beschouw het vocabularium Σ bestaande uit relatiesymbolen $OuderVan/2$ en $Vrouw/1$. We interpreteren $OuderVan$ en $Vrouw$ op de manier gesuggereerd door de namen. Druk de volgende eigenschappen uit in Σ:*

- *Niemand is vader van iedereen.*
- *Iedereen heeft een moeder.*
- *Wie een dochter heeft, heeft een zoon.*

Is de volgende zin logisch waar?

$$(\forall x)(\forall y_1)(\forall y_2)(\forall y_3)[\ OuderVan(y_1,x) \wedge OuderVan(y_2,x) \wedge OuderVan(y_3,x) \Rightarrow y_1 = y_2 \vee y_2 = y_3 \vee y_3 = y_1]$$

Is deze zin waar in het toepassingsdomein?

Oefening 4.1.5. *Een groep $\langle G, e, +/2\rangle$ is een structuur bestaande uit een verzameling, een neutraal element e en een binaire operator $+$ die aan de volgende eigenschappen voldoet: $+$*

is een totale functie, associatief, e is links en rechts neutraal element en elk element heeft een invers element. Kies een vocabularium en schrijf deze eigenschappen uit in logica. Merk op dat je met DecaWorld van LogicPalet nu zelf groepen kunt genereren van hoogstens 10 elementen.

In de volgende secties zullen we in de context van een aantal informaticaproblemen het modelleren van informatie in logica bestuderen, zowel het opstellen van een vocabularium als van de logische zinnen.

4.2 Wat is inferentie?

Een basisidee van deze cursus is dat informatie kan uitgedrukt worden in logica, en dat informaticaproblemen en taken opgelost kunnen worden met behulp van die informatie door middel van bepaalde vormen van inferentie. In de cursus hebben we verschillende vormen van inferentie gezien. Hieronder geven we een algemene definitie van wat we bedoelen met inferentie.

Definitie 4.2.1. Een inferentie-probleem is een probleem met een input en een gewenste output. De input bestaat uit één of meerdere logische objecten. Hiermee bedoelen we

- een symbool of vocabularium
- een waarde uit een domain
- een structuur: een toekenning van waardes aan symbolen
- een logische uitdrukking: een term, een verzamelingenuitdrukking, een formule of zin, een theorie.

De output bestaat uit één of meerdere logische objecten die aan een bepaalde logische voorwaarde in termen van de invoer voldoen.

Deze algemene definitie illustreren we met een aantal voorbeelden die eerder in de cursus voorkwamen.

- Een query-probleem:
  - Input: een structuur $\mathfrak{A}$, een zin A over het vocabularium van $\mathfrak{A}$.
  - Output: $\mathbf{t}$ indien $\mathfrak{A} \models A$, $\mathbf{f}$ indien $\mathfrak{A} \not\models A$.
- Het veralgemeende query-probleem:
  - Input: een structuur $\mathfrak{A}$, een verzamelingexpressie $\{(x_1, \ldots, x_n) : A\}$ over het vocabularium van $\mathfrak{A}$.
  - Output: $\{(x_1, \ldots, x_n) : A\}^{\mathfrak{A}}$, de relatie voorgesteld door de verzamelingexpressie in $\mathfrak{A}$.

- Een satisfieerbaarheids-probleem
  - Input: een zin of theorie T
  - Output: **t** indien T een model heeft, **f** indien T geen model heeft.
- Een modelgeneratie-probleem:
  - Input: een zin of theorie T
  - Output: één of meerdere of alle structuren $\mathfrak{A}$ zodat $\mathfrak{A} \models T$.
- Een deductie-probleem:
  - Input: een theorie T, een logische zin A
  - Output: **t** indien $T \models A$; **f** indien $T \not\models A$.
- Een modelexpansie-probleem:
  - Input: een zin of theorie T, een structuur $\mathfrak{A}_i$ die een deel van de symbolen van T interpreteert.
  - Output: een structuur $\mathfrak{A}$ zodat $\mathfrak{A} \models T$ en $\mathfrak{A}$ is een expansie van $\mathfrak{A}_i$, t.t.z., $\mathfrak{A}_i$ and $\mathfrak{A}$ hebben hetzelfde domein, en voor elk geïnterpreteerd symbool σ van $\mathfrak{A}_i$ geldt $\sigma^{\mathfrak{A}_i} = \sigma^{\mathfrak{A}}$.

Voor elk van deze problemen en vormen van inferentie hebben we reeds eerder voorbeelden in de cursus gezien. Bv. in Hoofdstuk 1 zagen we het query-probleem. Theoremprovers zijn gespecialiseerd in het oplossen van deductie-problemen. Het grafenkleuringsprobleem (Voorbeeld 2.5.2) is een voorbeeld van een modelexpansie-probleem: de input is een structuur $\mathfrak{A}_i$ die een grafe en een domein van kleuren specifieert. De output is een structuur $\mathfrak{A}$, of meer precies de relatie $C^{\mathfrak{A}}$ waarbij $\mathfrak{A}$ een model is van een theorie en een uitbreiding van $\mathfrak{A}_i$.

Een basisidee van deze cursus is dat veel informaticaproblemen op een natuurlijke manier geklassifieerd of gereduceerd kunnen worden tot een beperkt aantal inferentieproblemen. In het vervolg van dit hoofdstuk zien we nog meer voorbeelden hiervan.

4.3 Databanken revisited

Bij het opstellen van een databank moet een vocabularium gekozen worden. Eens zover ontstaat een databank door het creëren, aanvullen en wijzigen van tabellen. Er stellen zich verschillende modelleertaken:

- Het modelleren van integriteitsbeperkingen; dit is de verantwoordelijkheid van de databankmanager.
- Het modelleren van queries: een query is immers een symbolisch relatievoorschrift die precies de gezochte eigenschap of relatie van de gebruiker moet specifiëren. De verantwoordelijkheid voor het opstellen ligt natuurlijk bij de gebruikers van de databank.

In deze sectie bekijken we in meer detail naar een aantal logische aspecten van databanken.

4.3.1 Een databank als een logische theorie

In deze cursus hebben we twee uitgangspunten gezien die bij nader toezicht enigszins tegenstrijdig zijn:

- Enerzijds werden logische zinnen en logische theorieën voorgesteld als dé manier om informatie formeel uit te drukken.
- Anderzijds werd een databank gezien als een structuur.

Er is hier iets vreemds aan de gang. Het is onmiskenbaar dat een databank zeer veel informatie bevat. Waarom kiezen we er dan voor om deze informatie voor te stellen door een structuur in plaats van door een logische theorie?

In deze sectie bestuderen we de volgende logische modelleertaak: hoe kunnen we de informatie in een databank modelleren met logische formules?

Wat is het nut daarvan? We willen eenvoudigweg begrijpen welk soort informatie voorgesteld kan worden in een databank, en welke niet. We zullen merken dat er een paar belangrijke en onverwachte lessen te leren zijn over het modelleren van informatie in logica.

Nemen we als voorbeeld onze studentendatabank $\mathfrak{A}$ uit Figuur 2.1. We beschikken over een vocabularium Σ en kunnen ons dus de vraag stellen: bestaat er een logische theorie met dezelfde informatie-inhoud als deze structuur? Hoe ziet die eruit?

Op het eerste zicht ziet het eenvoudig genoeg uit. We stellen gewoon 1 grote formule T_{DB} op die een conjunctie is van alle atomen die waar zijn in de databank:

$$\begin{gathered} Instructor(Ray, CS230) \wedge \cdots \wedge Instructor(Pat, CS238) \wedge \\ Prerequ(CS230, CS238) \wedge \cdots \wedge Prerequ(M100, M200) \wedge \\ Enrolled(Jill, CS230) \wedge \cdots \wedge Enrolled(Flo, M200) \\ Grade(Sam, CS148, AAA) \wedge \cdots \wedge Grade(Flo, M100, B) \wedge \\ PassingGrade(AAA) \wedge \cdots \wedge PassingGrade(C) \end{gathered}$$

Klaar is kees. Of niet?

Als deze theorie echt alle informatie uit $\mathfrak{A}$ bevat, dan zou een theoremprover en een databanksysteem queries op dezelfde manier moeten kunnen beantwoorden. T.t.z. als het databanksysteem $\mathbf{t}$ antwoordt op query A moet de theoremprover antwoorden dat A een logische gevolg is van T_{DB} en omgekeerd. Als dat niet het geval is, dan is onze theorie fout of onvolledig, t.t.z., we hebben niet alle informatie waarvan de databank uitgaat uitgedrukt in de theorie.

Hier is een reeks van queries waarop een databanksysteem $\mathbf{t}$ antwoordt, maar die geen logisch gevolg zijn van T_{DB} en die dus ook niet bewezen kunnen worden door een theoremprover.

- $\neg Ray = Hec$
- $\neg Instructor(Sue, CS230)$
- $\neg(\exists x) Prerequ(x, x)$ - een integriteitsbeperking
- $(\forall x)(Prerequ(x, CS238) \Rightarrow Instructor(Ray, x))$

Bew. Elk van deze queries is een zin die waar is in $\mathfrak{A}$, en die door een databanksysteem met $\mathbf{t}$ zou beantwoord worden. Om aan te tonen dat het geen logische gevolgen zijn van T_{DB}, zullen we voor elke query in deze lijst een model $\mathfrak{B}$ van T_{DB} vinden die niet voldoet aan de query. Voor de eerste drie queries kiezen we $\mathfrak{B}$ als volgt:

- $D_{\mathfrak{B}} = \{a\}$; m.a.w. er is 1 domeinelement.
- $Ray^{\mathfrak{B}} = Hec^{\mathfrak{B}} = \cdots = CS230^{\mathfrak{B}} = \cdots = AAA^{\mathfrak{B}} = \cdots = Sam^{\mathfrak{B}} = \cdots = a$: a is de interpretatie van elke constante.
- $PassingGrade^{\mathfrak{B}} = \{a\}$.
- $Instructor^{\mathfrak{B}} = Prerequ^{\mathfrak{B}} = Enrolled^{\mathfrak{B}} = \{(a, a)\}$.
- $Grade^{\mathfrak{B}} = \{(a, a, a)\}$.

Het valt gemakkelijk in te zien dat $\mathfrak{B}$ een model is van T_{DB}. Immers, elke atomaire formule van Σ is waar in $\mathfrak{B}$, dus T_{DB} waar in $\mathfrak{B}$. Ook elk gelijkheidsatoom zoals $Ray = Hec$ is waar in $\mathfrak{B}$. Dus zijn de eerste drie queries onwaar in $\mathfrak{B}$. Bv. $\mathfrak{B} \not\models \neg(\exists x)Prerequ(x, x)$ want $\mathfrak{B} \models Prerequ(a, a)$.

De query $(\forall x)(Prerequ(x, CS238) \Rightarrow Instructor(Ray, x))$ is wel voldaan in $\mathfrak{B}$ want voor 'elk' domeinelement (t.t.z. a) geldt de conclusie. Dus moeten we een andere structuur vinden die wel aan T_{DB} maar niet aan deze formule voldoet.

We passen $\mathfrak{B}$ aan door een tweede element b toe te voegen aan het domein en één koppel toe te voegen aan $Prerequ^{\mathfrak{B}}$:

$$Prerequ^{\mathfrak{B}} = \{(a, a), (b, a)\}$$

Na deze aanpassing voldoet $\mathfrak{B}$ nog altijd aan T_{DB} en er geldt $\mathfrak{B}[x : b] \not\models Prerequ(x, CS238) \Rightarrow Instructor(Ray, x)$, dus is de vierde query niet voldaan. Q.E.D.

Als we kijken naar de queries uit Hoofstuk 1 dan kunnen we bewijzen dat geen enkele query die $\Rightarrow, \neq, \Leftrightarrow, \forall$ bevat te bewijzen valt uit T_{DB}. Enkel de allereenvoudigste ware queries kunnen beantwoord worden met een theoremprover uitgaande van T_{DB}.

Oefening 4.3.1. *Selecteer een aantal queries die $\Rightarrow, \neq, \Leftrightarrow$ of $\forall$ bevatten en construeer een structuur waarin de theorie wel en de query niet waar is.*

Wat is hier aan de hand? We botsen hier tegen een **uiterst belangrijk aspect** van het modelleren van informatie in logica. Dat is dat mensen beschikken over allerlei impliciete kennis die zo evident is voor ons dat we er niet eens aan denken om deze kennis te expliciteren. Maar logica weet niets, en als je die impliciete informatie niet uitdrukt, dan "weet" uw logische theorie die ook niet.

Er zijn drie soorten informatie die niet uitgedrukt werden in T_{DB} waar wij en het databanksysteem toch van uitgaan maar zonder dewelke een theoremprover onmogelijk kan weten dat de queries waar zijn.

- In een databank kiest men verschillende strings om verschillende objecten aan te duiden. We weten dus dat $Ray, Hec, Sam, CS230, AAA, \ldots$ allemaal verwijzen naar verschillende elementen van het domein. Dat werd niet uitgedrukt in T_{DB}. Dus kan de theoremprover niet weten dat $\neg Ray = Hec$.
- We hebben ook niet uitgedrukt dat er geen andere objecten in het universum van onze toepassing zijn dan degene die overeenkomen met het databankdomein

$$\{Ray, \ldots, Sam, \ldots, CS230, \ldots, AAA, \ldots\}.$$

- In onze logische theorie werd enkel aangegeven welke koppels wel in de tabellen zitten maar niet welke koppels niet in de tabellen zitten. Bv. de theorie drukt niet uit dat $\neg Prerequ(CS230, CS230)$. Dus kan een theoremprover ook niet bewijzen dat geen enkele cursus voorkennis van zichzelf is.

Hieronder zullen we drie soorten axioma's introduceren om de noodzakelijke informatie uit te drukken. Zij $S = \{C_1, \ldots, C_n\}$ een willekeurige niet lege verzameling van constanten.

Definitie 4.3.1. De **Unique Name Axioma's** voor S, notatie $UNA(S)$, is de verzameling van alle formules $\neg C_i = C_j$ voor elk paar van verschillende constanten $C_i, C_j \in S$.

Propositie 4.3.1. *In elk model $\mathfrak{A}$ van $UNA(S)$ geldt voor elk paar van onderling verschillende constanten $C_i, C_j \in S$ dat $C_i^{\mathfrak{A}} \neq C_j{}^{\mathfrak{A}}$.*

Bew. Evident.

Definitie 4.3.2. Het Domain Closure Axioma voor S, notatie $DCA(S)$, is de zin:

$$(\forall x)(x = C_1 \vee \cdots \vee x = C_n)$$

Propositie 4.3.2. *In elk model $\mathfrak{A}$ van $DCA(S)$ bestaat voor elk domeinelement $d \in D_{\mathfrak{A}}$ een constante C_i zodat $C_i^{\mathfrak{A}} = d$.*

Bew. Triviaal uit de $\forall$-regel en de $\vee$-regel van de definitie van waarheid.

Voorbeeld 4.3.1. Zij $S = \{Obama, Silvio, Herman\}$ en $\Sigma = S \cup \{President\_of\_US\}$. Dan geldt in elke structuur $\mathfrak{A}$ van Σ die voldoet aan $DCA(S)$ dat

$$President\_of\_US^{\mathfrak{A}} \in \{Obama^{\mathfrak{A}}, Silvio^{\mathfrak{A}}, Herman^{\mathfrak{A}}\}$$

Of ook :

$$DCA(S) \models President\_of\_US = Obama \vee \cdots \vee President\_of\_US = Herman$$

M.a.w, de interpretatie van elke constante buiten S is identiek aan de interpretatie van minstens 1 van de constanten van S.

Propositie 4.3.3. *Een structuur $\mathfrak{A}$ is een model van $UNA(S) \wedge DCA(S)$ asa $D_{\mathfrak{A}}$ n elementen heeft zodat voor elk domeinelement $d \in D_{\mathfrak{A}}$ een unieke constante $C_i \in S$ bestaat zodat $C_i^{\mathfrak{A}} = d$. Voor elk paar van modellen $\mathfrak{A}, \mathfrak{B}$ van deze theorie bestaat er een bijectie $b : D_{\mathfrak{A}} \to D_J$ zodat voor elke $d \in D_{\mathfrak{A}}$ geldt dat als $C^{\mathfrak{A}} = d$ dan is $C^{\mathfrak{B}} = b(d)$.*

Bew. Door combinatie van de twee vorige proposities.

Vanaf nu beperken we ons opnieuw tot databankstructuren $\mathfrak{A}$ en het bijbehorende vocabularium Σ. Neem $S = \{C_1, \ldots, C_n\}$ de verzameling van constanten die overeenkomen met het domein van $\mathfrak{A}$. Dus, $D_{\mathfrak{A}} = \{C_1, \ldots, C_n\}$ en $C_i{}^{\mathfrak{A}} = C_i$.

Propositie 4.3.4. *Een databankstructuur $\mathfrak{A}$ is een model van $UNA(S) \wedge DCA(S)$.*

Definitie 4.3.3. Zij $P/k \in \Sigma$ een predikaatsymbool zodat $P^{\mathfrak{A}}$ bestaat uit m koppels $(C_{1,1}, \ldots, C_{1,k}), \ldots, (C_{m,1}, \ldots, C_{m,k})$. De *definitie* van P, notatie $Def(P)$, is de logische zin:

$$\begin{aligned}(\forall x_1)\ldots(\forall x_k)(P(x_1,\ldots,x_k) \Leftrightarrow\ & (x_1 = C_{1,1} \wedge \cdots \wedge x_k = C_{1,k}) \\ & \vee \cdots \vee \\ & (x_1 = C_{m,1} \wedge \cdots \wedge x_k = C_{m,k}))\end{aligned}$$

Voorbeeld 4.3.2. $Def(Prerequ)$ is:

$$\begin{aligned}(\forall x)(\forall y)[Prerequ(x,y) \Leftrightarrow\ & (x = CS148 \wedge y = CS230) \vee \\ & (x = CS230 \wedge y = CS238) \vee \\ & (x = M100 \wedge y = M200)]\end{aligned}$$

De volgende stelling gaat niet over de databankstructuur $\mathfrak{A}$ maar over willekeurige modellen $\mathfrak{B}$ van $Def(P)$.

Propositie 4.3.5. *Een structuur $\mathfrak{B}$ is een model van $Def(P)$ asa er geldt dat voor elk koppel domeinelementen $(d_1, \ldots, d_k) \in P^{\mathfrak{B}}$ een koppel $(C_{i,1}, \ldots, C_{i,k}) \in P^{\mathfrak{A}}$ bestaat zodat ${C_{i,1}}^{\mathfrak{B}} = d_1, \ldots, {C_{i,n}}^{\mathfrak{B}} = d_n$. De databankstructuur $\mathfrak{A}$ is een model van $Def(R)$.*

Bew. Direct uit toepassing van de $\forall$-, $\Leftrightarrow$-, en $\vee$-regel op $Def(P)$.

Definitie 4.3.4. Definieer de theorie van $\mathfrak{A}$, notatie $Theo(\mathfrak{A})$, als de verzameling van logische zinnen:

- $UNA(S)$;
- $DCA(S)$;
- $Def(P)$, voor elk predikaatsymbool $P \in \Sigma$.

$Theo(\mathfrak{A})$ is de manier om de informatie in een databank te modelleren in logica. Wat we zullen bewijzen is dat deze theorie alle informatie bevat uit $\mathfrak{A}$, in de zin dat een theoremprover elke query op dezelfde manier kan beantwoorden uit $Theo(\mathfrak{A})$ als dat een databanksysteem zou doen uit $\mathfrak{A}$. Dat doen we door te bewijzen dat $Theo(\mathfrak{A})$ essentieel maar één model heeft, namelijk $\mathfrak{A}$ zelf.

Dit klopt niet helemaal. De theorie $Theo(\mathfrak{A})$ heeft wel degelijk oneindig veel modellen, maar ze hebben allemaal exact dezelfde inwendige structuur als $\mathfrak{A}$. We zeggen dat alle modellen *isomorf* zijn met $\mathfrak{A}$.

Definitie 4.3.5. Twee structuren $\mathfrak{A}, \mathfrak{B}$ van hetzelfde vocabularium Σ zijn **isomorf** asa er een bijectie b bestaat van $D_{\mathfrak{A}}$ naar $D_{\mathfrak{B}}$ die de interpretatie van elk symbool bewaart:

- als $C^{\mathfrak{A}} = d$ dan is $C^{\mathfrak{B}} = b(d)$, voor elke constante $C \in \Sigma$.
- als $F^{\mathfrak{A}}(d_1, \ldots, d_n) = d$ dan is $F^{\mathfrak{B}}(b(d_1), \ldots, b(d_n)) = b(d)$, voor elke functie $F \in \Sigma$;
- als $\mathfrak{A} \models P(d_1, \ldots, d_n)$ dan geldt $\mathfrak{B} \models P(b(d_1), \ldots, b(d_n))$, voor elk predikaat $P \in \Sigma$.

Eigenschap 4.3.1. In isomorfe structuren zijn dezelfde formules waar.

We zullen dit niet bewijzen maar het bewijs is niet moeilijk. Het is een inductiebewijs geheel analoog aan het bewijs van de Generaliseereigenschap 3.3.1.

Oefening 4.3.2. *Bewijs deze stelling.*

Stelling 4.3.1. *Zij gegeven een databankstructuur $\mathfrak{A}$ met vocabularium Σ.*
(a) Een structuur $\mathfrak{B}$ van Σ is een model van $Theo(\mathfrak{A})$ asa $\mathfrak{B}$ en $\mathfrak{A}$ isomorf zijn.

(b) Voor elke logische zin A van Σ geldt dat $Theo(\mathfrak{A}) \models A$ asa $\mathfrak{A} \models A$.

In woorden: A is een logisch gevolg van de theorie $Theo(\mathfrak{A})$ asa A waar is in de structuur $\mathfrak{A}$. Dat betekent dat -in principe- een theoremprover uit $Theo(\mathfrak{A})$ dezelfde queries kan beantwoorden als een databanksysteem uit $\mathfrak{A}$. Het betekent ook dat we een datanksysteem kunnen beschouwen als een superefficiënte theoremprover voor de theorie $Theo(\mathfrak{A})$.

Bew.

(a) Uit Stelling 4.3.3 volgt al dat $\mathfrak{B}$ een model is van $UNA(S) \wedge DCA(S)$ asa er een bijectie bestaat tussen het domein van $\mathfrak{B}$ en van $\mathfrak{A}$ zodat $b({C_i}^{\mathfrak{B}}) = C_i = {C_i}^{\mathfrak{A}}$. Uit Stelling 4.3.5 volgt ook dat een dergelijke $\mathfrak{B}$ voldoet aan de definitie $Def(P)$ voor $P \in \Sigma$ asa de afbeelding b toegepast op de koppels van $P^{\mathfrak{B}}$ precies de koppels zijn van $P^{\mathfrak{A}}$.

(b) Aangezien in isomorfe structuren dezelfde formules waar zijn, volgt (b) uit (a). Q.E.D.

Deze stelling garandeert ons dat $Theo(\mathfrak{A})$ alle informatie bevat die in de databank $\mathfrak{A}$ vervat zit.

Deze informatie is *logisch consistent* en *volledig*. De informatie is consistent want er is een model, en de informatie is volledig want het model is uniek (modulo isomorfie) en dus voor elke zin A van Σ geldt dat ofwel A, ofwel $\neg A$ een logisch gevolg is van $Theo(DB)$. Dit volgt direct uit Stelling 4.3.1 en het feit dat $\mathfrak{A} \models A$ ofwel $\mathfrak{A} \models \neg A$.

Volledige versus onvolledige informatie. We hebben nu een goed inzicht in welke soort informatie aanwezig is in een databank. Het gaat over *feitelijke* informatie, over het domein (DCA), over de identiteit van de domeinelementen (UNA) en over de relaties in het domein. Deze informatie is zowel consistent als volledig.

Er zijn heel wat toepassingen waarin mensen niet over zo'n volledige informatie beschikken, ook in de context van databanken. Bv. het zou kunnen dat je weet dat Jill examen heeft afgelegd

van CS148 zonder haar score te kennen. Of dat Flo een B of een D heeft voor haar examen M100 zonder te weten hetwelk.

Het is een belangrijk onderzoeksprobleem over hoe databankensystemen uit te breiden om met dergelijke onvolledige kennis te redeneren. De meest voorkomende manier is door middel van null-waardes. Stel dat er in een tabel van de databank één of meerdere waardes ontbreken. In de bestaande databanksystemen zet men dan de NULL op die plaatsen. Men noemt dit een null-waarde. Het betekent dat de waarde voor die positie niet gekend is. Bv.

Instructor

Ray	NULL
Hec	CS230
Sue	M100
NULL	M200
Pat	CS238

Intuïtief betekent dit het volgende: de tabel stelt alle informatie over wie welke vakken doceert behalve dat we niet weten welke vak Ray doceert, en niet wie M200 doceert. Merk op dat het Ray zou kunnen zijn die M200 doceert.

Zo'n null-waardes maken het beantwoorden van queries moeilijker. Bv. bij het beantwoorden van de vraag wie CS230 doceert, moet Ray als mogelijk antwoord gegeven worden. We zullen ons daar nu niet mee bezig houden. De vraag die we ons nu stellen is: hoe moet $Theo(DB)$ aangepast worden zodat we een theoremprover als Spass kunnen gebruiken om correct queries mee te beantwoorden, voor kleine databankjes tenminste.

Het basisidee is het volgende: we stellen voor om voor elk voorkomen van NULL een nieuwe logische constante n_i te introduceren. Dus we kunnen de constanten van Σ nu opsplitsen in de verzameling S van de oorspronkelijke constanten waarvan we de identeit kennen, en de nieuwe null-constanten $n_1, \ldots, n_l$. De theorie wordt als volgt aangepast:

- $UNA(S)$: dit axioma verandert niet maar heeft enkel betrekking op constanten met bekende ideniteit; de theorie sluit wel uit dat $Ray = Hec$ maar niet dat $Ray = n_2$.
- $DCA(S)$: het impliceert dat n_i gelijk moet zijn aan één van de constanten uit S met gekende identiteit.
- $Def(P)$, voor elk predicaat $P \in \Sigma$: ook deze formules veranderen niet. Ze impliceren dat Ray het vak n_1 geeft, en dat iemand n_2 het vak $M200$. De formule ziet er als volgt uit:

$$\begin{aligned}(\forall x)(\forall y)(Instructor(x,y) \Leftrightarrow\ & (x = Ray \wedge y = n_1) \vee \\ & (x = Hec \wedge y = CS230) \vee \\ & (x = Sue \wedge y = M100) \vee \\ & (x = n_2 \wedge y = M200) \vee \\ & (x = Pat \wedge y = CS238))\end{aligned}$$

We kunnen nu bijkomende constraints opleggen aan n_1 en n_2 zoals $n_1 \neq n_2$ of:

$$n_1 = CS230 \vee n_1 = CS238$$

Zoiets is onmogelijk in klassieke databanken.

Oefening 4.3.3. *Uitwerking met SPASS. Voor een eenvoudige databank DB met null-waardes, plaats in tekstvak A van LogicPalet de conjunctie van alle zinnen van $TheoDB$. Plaats een query in tekstvak B. Vraag Spass of B een logisch gevolg is van A. Indien je geen positief antwoord krijgt vul dan de negatie van de query in in B en roep opnieuw Spass op. Als noch de query noch haar negatie een logische gevolg is van $Theo(DB)$ dan bevat de logische databank te weinig informatie om de query te beantwoorden, t.t.z. dan bestaat er een model van $Theo(DB)$ waarin de query waar is, en ook een model van $Theo(DB)$ waarin de query onwaar is.*

Opmerking 4.3.1. De rest van Sectie 4.3.1 is ter informatie en moet je niet kennen voor het examen.

Open domeinen. In Hoofdstuk 1 zagen we een query voor de studentendatabank waarvoor het antwoord in logica en in SQL-databanken verschillend zou zijn: *Heeft John examen voor CS230 afgelegd?*

$$(\exists s)Grade(John, CS230, s)$$

"John" komt niet voor in het databankdomein, en de databankstructuur $\mathfrak{A}$ heeft geen interpretatie voor de constante $John$. Dus is de waarheid van deze zin niet bepaald in de structuur $\mathfrak{A}$.

Wat een echt databanksysteem doet is opzoeken of er koppel is van de vorm $(John, CS230, s)$ in de tabel van $Grade$ en aangezien zo'n koppel niet bestaat zal het antwoorden "onwaar".

We kunnen de logische theorie $Theo(\mathfrak{A})$ gemakkelijk aanpassen om met deze visie rekening te houden. Veronderstel dat we een query $\{(x_1, \ldots, x_n) : A\}$ stellen aan de databank, en A bevat een aantal nieuwe constanten S_A die niet in het databankdomein voorkoken. Neem de theorie $Theo(\mathfrak{A}, A)$ bestaande uit $UNA(S \cup S_A)$ en $Def(P)$, voor elk predikaatsymbool $P \in \Sigma$. Voor de query $(\exists s)Grade(John, CS230, s)$ is $S_A = \{John\}$. De resulterende theorie impliceert inderdaad dat deze query onwaar is.

De nieuwe theorie stelt nu geen beperkingen meer aan het universum. M.a.w., het universum is onbekend en kan een willekeurige (eventueel oneindige) superset zijn van de elementen die in de databank voorkomen. Daarom is de nieuwe theorie zwakker en sommige queries kunnen er niet in beantwoord worden.

Voorbeeld 4.3.3. Neem volgende twee queries:

$$\{x : x = x\} \text{ en } \{x : \neg PassingGrade(x)\}$$

Wat is het antwoord erop in twee structuren: de oorspronkelijke databankstructuur $\mathfrak{A}$, en in de structuur $\mathfrak{B}$ die je bekomt uit $\mathfrak{A}$ door een nieuw object, bv. $John$, toe te voegen aan $D_{\mathfrak{A}}$. Het antwoord op beide queries verschilt in beide doordat het antwoord in $\mathfrak{B}$ het nieuwe domeinelement $John$ bevat.

We kunnen ons nu de vraag stellen hoe databanken omgaan met dergelijke queries? Het antwoord is dat dergelijke queries eenvoudigweg niet gesteld kunnen worden in de huidige databanksystemen!

Databanksystemen stellen bepaalde syntactische beperkingen aan queries A zodat het antwoord erop identiek is in elk model van $Theo(\mathfrak{A}, A)$. M.a.w., de queries zijn *domeinonafhankelijk*. Dergelijke queries worden *veilige queries* genoemd. In Sectie **??** zullen we dit soort queries beschrijven.

Deductieve databanken. Dat is de naam van het onderzoeksdomein waarin veralgemeende databanken bestudeerd worden. Onderwerpen die hierin aan bod komen zijn o.a.:

- integriteitsbeperkingen;
- definities, t.t.z. views, eventueel recursief;
- onvolledige kennis zoals null waardes.

In dit onderzoeksdomein worden technieken ontwikkeld die vervolgens in databanksystemen geïntegreerd worden, in de vorm van uitbreidingen van SQL. Zo zijn views, null-waardes, en bepaalde eenvoudige types van integriteitsbeperkingen in databanksystemen toegevoegd.

In de context van databanken en deductieve databanken wordt - in tegenstelling tot wat de naam suggereert - een brede klasse van redeneertaken bestudeerd - niet alleen deductieve. We zagen al een aantal:
- Bereken de interpretatie van een relatievoorschrift in een structuur.
- Bereken een equivalente query die sneller opgelost kan worden dan de oorspronkelijke.
- Bereken een gedefinieerd predikaat in een structuur.
- Pas de interpretatie van een gedefinieerd predicaat aan na een wijziging van de structuur.

Er zijn er nog andere. Bv. het *update* probleem:

Pas de databank (minimaal) aan zodat een formule voldaan wordt.

Veronderstel dat volgens de databank van de univ ik morgen les moet geven, maar ik ben ziek, dan zou men willen zoeken naar een gewijzigde databank die voldoet aan de formule dat ik morgen geen les geef zonder heel de databank van nul her op te bouwen. Dit is een zeer moeilijk redeneerprobleem.

4.3.2 Een query-algoritme

In Figuur 4.1 schetsen we een algoritme in pseudocode om queries te evalueren in een databank $\mathfrak{A}$. De inductieve Definitie 2.2.4 van waarheid in een structuur is eigenlijk zelf al bijna een algoritme is om queries te berekenen. De functie eval is gewoon een herformulering ervan. Aangezien Definitie 2.2.4 inductief is, en dus is deze procedure recursief: ze roept zichzelf op.

Merk op: wanneer een **return** instructie uitgevoerd wordt, stopt de uitvoering van dit programma. Ook zullen we hieronder veronderstellen dat we elke formule $(\forall x)A$ vervangen hebben door de equivalente formule $\neg(\exists x)\neg A$. We gebruiken ook de volgende notatie: als A een formule is met vrije variabele x, dan is $A[d/x]$ de formule bekomen door de vrije voorkomens van x te vervangen door d.

Opmerking 4.3.2. In deze pseudo-code worden booleaanse operatoren gebruikt: && staat voor $\wedge$, || staat voor $\vee$ en ! staat voor $\neg$. Je ziet dat concepten uit de propositionele logica ook in programmeertalen gebruikt worden!

```
function eval(Zin Q, Databank 𝔄):Boolean {
   // Input: Een zin Q, een databank 𝔄 die Q interpreteert.
   // Output: t of f.
   Herschrijf Q door het elimineren van ⇔, ⇒, ≠, ∀;
   if ( Q is een atoom P(d_1, ..., d_n)) then {
      if ( (d_1, ..., d_n) ∈ P^𝔄 ) then return t
      else return f;
      }
   else if ( Q heeft vorm d_1 = d_2 ) then {
      if (d_1 en d_2 zijn identiek) then return t
      else return f;
      }
   else if (Q heeft vorm A ∧ B) then return (eval(A) && eval(B))
   else if ( Q heeft vorm A ∨ B ) then return (eval(A) || eval(B))
   else if ( Q heeft vorm ¬A ) then return (!eval(A))
   else if ( Q heeft vorm (∃x)A ) then {
      for ( Object d ∈ dom_DB) do { //voor elk element d in het universum doe:
         Zij B := A[d/x];
         if eval(B) then return t;
         }
      return f; // voor geen enkele d voldoet B
      }
   }

function answer(Query {(x_1, ..., x_n) : A}, Databank 𝔄): n-voudige relatie {
   // Input: Een relatievoorschrift, een databank 𝔄 die A interpreteert.
   // Output: een n-voudige relatie.
   Initialiseer R = ∅;
   for all d_1, ..., d_n ∈ dom_𝔄 do {
      Zij B := A[d_1/x_1, ..., d_n/x_n];
      if eval(B) then R = R ∪ {(d_1, ..., d_n)};
      }
   return R
   }
```

Figuur 4.1: Query-algorithm

Voorbeeld 4.3.4. Een voorbeeld van een uitvoering van de oproep eval($(\exists c)(Grade(Sam, c, AAA) \vee Grade(Sam, c, AA)), \mathfrak{A}$)

- **for** ($\mathbf{d} \in dom_{\mathfrak{A}}$) **do** {
  // d : Ray, ..., CS230, ..., AAA,..., !! $B := Grade(Sam, \mathbf{d}, AAA) \vee Grade(Sam, \mathbf{d}, AA)$;
  Call eval($B, \mathfrak{A}$)
  Indien antwoord **t** dan **return t**;
  Indien antwoord **f** dan neem volgend domein element.
  }
  Indien het eind van de lus bereikt wordt, **return f**

- eval($Grade(Sam, \mathbf{d}, AAA) \vee Grade(Sam, \mathbf{d}, AA), \mathfrak{A}$):
  - Call eval($Grade(Sam, \mathbf{d}, AAA), \mathfrak{A}$); stop antwoord in v
    Call eval($Grade(Sam, \mathbf{d}, AA), \mathfrak{A}$); stop antwoord in w
    return v || w
  - eval($Grade(Sam, \mathbf{d}, AAA), \mathfrak{A}$)
    - Verifieer of $(Sam, \mathbf{d}, AAA) \in Grade^{\mathfrak{A}}$
      Zo ja **return t**
      Zo niet **return f**
  - eval($Grade(Sam, \mathbf{d}, AAA), \mathfrak{A}$)
    - ...

Opmerking 4.3.3. In LogicPalet zit een tool om uit te leggen waarom een formule waar of onwaar in een structuur. Experimenteer met dit tool. Dan zul je zien dat dit tool eigenlijk de functie **eval** stapsgewijs uitvoert.

Het algoritme in Figuur 4.1 is zeer naïef en is zeker niet in staat is om queries te beantwoorden in grote databanken. De query-methodes die geïmplementeerd zijn in databanken kunnen beschouwd worden als supergeoptimaliseerde versies van dit algoritme.

De belangrijkste optimalisatie is dat de databankalgoritmes nooit alle domeinelementen van de databank overlopen: ze doorlopen enkel koppels in tabellen. Bv. om de query $(\exists c)(Grade(Sam, c, AAA)$ te berekenen, volstaat het de koppels van *Grade* af te lopen. Dergelijke methodes behoren tot het domein van de *relationele algebra*. Dit levert een enorme tijdswinst op.

Op zo'n manier is het echter niet mogelijk om alle queries op te lossen. Bv. het antwoord op de query $\{x : \neg Prerequ(x, x)\}$ kan onmogelijk berekend worden alleen gebruik makend van de koppels in de tabel van *Prerequ*. Ook het antwoord op $\{x : x = x\}$ kan niet berekend worden: een database houdt geen tabel bij van $=$.

De oplossing in databanken is om beperkingen op te leggen op de vorm van queries. *Veilige queries* hebben de eigenschap dat ze berekend kunnen worden met relationele algebra, dus, enkel gebruikmakend van de tabellen. Bovendien zijn ze *domeinonafhankelijk* (zie vorige sectie) en, zoals we zullen zien in Sectie 4.3.5, ook *impliciet getypeerd*. Hier worden dus drie vliegen in één klap geslagen. Hebben ze ook nadelen? Bv. een verlies van expressiviteit? Bestaan er ook interessante queries die niet "veilig" zijn? In principe is dat mogelijk, maar in dagdagelijks gebruik ondervindt men blijkbaar geen hinder van deze beperking.

Oefening 4.3.4. *We hebben ervoor gekozen om* $\Rightarrow, \Leftrightarrow, \forall$ *weg te vertalen, maar dat is helemaal niet nodig. Voeg expliciete code toe voor* $\Rightarrow, \Leftrightarrow, \forall$.

Opmerking 4.3.4. Je moet dit algoritme begrijpen, maar je moet het niet kunnen uitschrijven.

4.3.3 Query optimalisatie

Zij A, B logisch equivalente formules (eventueel modulo de theorie van integriteitsbeperkingen) met vrije variabelen $x_1, \ldots, x_n$. Dan hebben de queries $\{(x_1, \ldots, x_n : A\}$ en $\{(x_1, \ldots, x_n) : B\}$ hetzelfde antwoord in elke (zinvolle) databank/structuur. Het is echter best mogelijk dat het een databanksysteem véél meer tijd kost om het antwoord op één query te berekenen dan op het andere, hoewel de antwoorden uiteindelijk identiek zijn.

Voorbeeld 4.3.5. Is er een docent die ooit score AAA gegeven heeft voor een voorkennisvak dat hij nu doceert? Deze kan op verschillende manieren uitgedrukt worden. Als Q1:

$$(\exists x)(\exists y)(\exists s)(\exists t)(Instructor(x, y) \wedge Grade(s, y, AAA) \wedge Prerequ(y, t) \wedge Instructor(x, t))$$

versus als Q2:

$$(\exists x)(\exists y)[Instructor(x, y) \wedge (\exists s)Grade(s, y, AAA) \wedge (\exists t)(Prerequ(y, t) \wedge Instructor(x, t))]$$

Het is gemakkelijk te zien dat beide equivalent zijn. In het query-algoritme in Figuur 4.1 vereist Q1 vier geneste for-lussen met toekenningen aan achtereenvolgens x, y, s, t, terwijl Q2 eerst twee geneste for-lussen heeft voor x, y, en daarbinnen een geneste lus voor s en apart voor t: dus maximaal drie geneste lussen. Dat maakt de tweede query Q2 veel efficiënter dan Q1 voor het recursief query algoritme.

Om een idee te geven, veronderstel dat er 1000 domeinelementen zijn in de database. Dan zal het algoritme voor de eerste query $1000^4 = 10^{12}$, dus 1000 miljard toekenningen doen aan de binnenste variabele t. Voor de tweede formule zijn er twee binnenste variabelen s en t en aan elk zijn er slechts 1 miljard toekenningen. 500 keer minder. Een gigantisch verschil!

De meeste hedendaagses databanksystemen hebben een krachtige optimizer die queries omzet naar logisch equivalente queries die minder rekenwerk vergen, zonder dat de gebruiker daar iets van merkt. Zo'n optimizer verricht de volgende logische redeneertaak:

> Voor query $\{(x_1, \ldots, x_n\} : A\}$ bereken B zodat A, B logisch equivalent zijn, maar het query-algoritme is efficiënter voor B dan voor A.

De taak van de optimizer is afhankelijk is van het query-algoritme. Toch zijn er een paar algemene regels waar alle optimizers aan voldoen. Merk op dat in het voorbeeld Q1 de prenex-normaalvorm is van Q2. Als het gaat om efficiënte query answering is omzetten naar prenex-normaalvorm zowat het slechtste wat men kan doen. Integendeel, bij het optimaliseren van een query zal men juist proberen om kwantoren zo diep mogelijk in formules te drijven.

Er zijn overigens andere toepassingen waarvoor de prenex-normaalvorm wel nuttig is. In het vak Artificiële Intelligentie van de tweede bachelor wordt de prenex-normaalvorm gebruikt in de context van resolutie, een techniek voor theorem proving.

Opmerking 4.3.5. Van deze sectie moet je onthouden wat query optimalisatie is, en je zou ook het argument moeten kunnen uitleggen in Voorbeeld 4.3.5. Merk op dat je daarvoor toch het algoritme moet begrijpen.

4.3.4 Correct-getypeerde databanken

In alle toepassingsdomeinen behalve de allereenvoudigste (zoals de theorie van groepen) bestaat het universum uit een verschillende types van objecten. Dit is zeker zo in databanken. In een zinvolle databank verwachten we dat de relaties goed getypeerd zijn, t.t.z. in elke kolom staan objecten van het correcte type. Laat ons eens bekijken hoe we dit kunnen uitdrukken in logica.

Bv. in het geval van de studentendatabank kunnen we vier types onderscheiden. We introduceren daarvoor vier nieuwe 1-voudige type-predikaatsymbolen $Student/1, Docent/1, Score/1, Vak/1$. In een zinvolle databank zouden de volgende type-integriteitsbeperkingen voldaan zouden moeten zijn:

$$(\forall x)(\forall y)(Instructor(x,y) \Rightarrow Docent(x) \wedge Vak(y))$$
$$(\forall x)(\forall y)(Enrolled(x,y) \Rightarrow Student(x) \wedge Vak(y))$$
$$(\forall x)(\forall y)(Prerequ(x,y) \Rightarrow Vak(x) \wedge Vak(y))$$
$$(\forall x)(\forall y)(\forall z)(Grade(x,y,z) \Rightarrow Student(x) \wedge Vak(y)) \wedge Score(z))$$
$$(\forall x)(PassingGrade(x) \Rightarrow Score(x))$$

Deze zinnen zijn niet geïnterpreteerd in de studentendatabank $\mathfrak{A}$. Immers, $\mathfrak{A}$ bevat geen tabellen voor de type-predicaten, dus is de waarheid van deze zinnen onbepaald in $\mathfrak{A}$ -in tegenstelling tot andere integriteitsbeperkingen. Om deze formules om te zetten tot verifieerbare queries moeten de type-predicaten vervangen worden door *definiërende formules*: formules die omschrijven welke objecten van de databank tot dat type behoren. In de voorbeelden van hoofdstuk 1 komen verschillende zo'n definiërende formules voor in queries:

- Bv. we hebben $Docent(t)$ gedefinieerd door de formule $(\exists c)Instructor(t,c)$.
- Bv. we kunnen $Vak(c)$ definiëren door $(\exists t)Instructor(t,c)$.

Als we vervolgens een type-predikaat in de type-integriteitsbeperkingen vervangen door de definiërende formule, dan bekomen we logische zinnen die wel geïnterpreteerd zijn door de databank en die we dus kunnen verifiëren in de databank, bv.:

$$(\forall x)(\forall y)(Enrolled(x,y) \Rightarrow (\exists t)Instructor(t,y))$$

Oefening 4.3.5. *Stel definiërende formules voor om $Student, Score$ te definiëren. Geef aan onder welk voorwaarde op het design van de databank ze correct zijn. Voor $Score$ is dat vervelend, niet?*

In de volgende secties zullen we ervan uitgaan dat de databank goed getypeerd is. Dus dat deze voldoet aan de volgende soort formules:

- Type-integriteitsbeperkingen:

$$(\forall x_1) \ldots (\forall x_n)(P(x_1, \ldots, x_n) \Rightarrow T_1(x_1) \wedge \cdots \wedge T_n(x_n))$$

- Goedgetypeerde objecten:

$$Vak(CS230) \wedge \cdots \wedge Docent(Ray) \wedge \cdots \wedge Student(Sam) \wedge \cdots \wedge Score(AAA) \wedge \ldots$$

4.3.5 Expliciet en impliciet getypeerde queries

In natuurlijke taal kwantificeren we nooit over het hele universum maar altijd over een deelklasse of een deeltype. Dat geldt ook voor queries. Predikatenlogica is geen getypeerde taal. SQL, de taal die in databanksystemen gebruikt wordt evenmin. Dat betekent dat we modelleertechnieken nodig hebben om te zorgen dat onze queries goed getypeerd zijn.

Veronderstel dat ons vocabularium expliciete type-predikaten bevat zoals in Sectie 4.3.4.

Definitie 4.3.6. Een *expliciet getypeerde logische formule* is van de vorm waarin elke gekwantificeerde deelformule eruit ziet als $(\exists x)(T(x) \wedge A)$ of $(\forall x)(T(x) \Rightarrow A)$ waarbij T het type is van de variabele x. Een *expliciet getypeerde n-voudige query* is een relatievoorschrift van de vorm $\{(x_1, \ldots, x_n) : T_1(x_1) \wedge \cdots \wedge T_n(x_n) \wedge A\}$, met A een expliciet getypeerde formule.

Voorbeeld 4.3.6. Expliciet getypeerde queries over de studentendatabank zijn:

$$(\exists c)(Vak(c) \wedge Grade(Sam, c, AAA))$$
$$(\forall y)[Docent(y) \Rightarrow (\exists w)(Vak(w) \wedge Instructor(y, w) \wedge Enrolled(Jack, w))]$$
$$\{x : Vak(x) \wedge \neg Prerequ(x, x)\}$$

Elke query die we aan de databank willen stellen komt intuïtief overeen met zo'n expliciet getypeerde query. Maar natuurlijk kunnen we een expliciet getypeerde query niet rechtstreeks aan de databank stellen, want er zijn geen tabellen voor de type-predikaten.

We bekomen evalueerbare queries door de type-predikaten te vervangen door definiërende formules. Maar soms doen we dat niet en is dat ook niet nodig.

Voorbeeld 4.3.7. Neem de query:

$$(\exists c)Grade(Sam, c, AAA)$$

De expliciet getypeerde versie is:

$$(\exists c)(Vak(c) \wedge Grade(Sam, c, AAA))$$

Door $Vak(c)$ te vervangen door zijn definiërende formule bekomen we een query die evalueerbaar is in de databank.

$$(\exists c)((\exists y)Instructor(y, c) \wedge Grade(Sam, c, AAA))$$

Maar het antwoord op beide queries zal identiek zijn: immers als de eerste query waar is, dan is het omdat er een koppel (Sam, d, AAA) in de tabel van $Grade$ zit, en aangezien we ervan uitgaan dat de databank goed getypeerd is, zal d een vak zijn.

Voor andere queries loopt het echter grondig mis als we niet expliciet typeren.

Voorbeeld 4.3.8. Het antwoordt op de query $\{x : \neg Prerequ(x, x)\}$ produceert niet alleen de vakken die geen voorkennis zijn van zichzelf, maar ook alle studenten, docenten en scores. Dit is niet wat je wenst. Een correcte formulering van deze query bekom je door de definiërende formule voor $Vak(x)$ eraan toe te voegen:

$$\{x : (\exists t)Instructor(t, x) \wedge \neg Prerequ(x, x)\}$$

Voorbeeld 4.3.9. Neem de query dat Jack les volgt bij elke docent, poging 1:

$$(\forall y)(\exists w)(Instructor(y, w) \wedge Enrolled(Jack, w))$$

De types van y en w zijn respectievelijk *Docent* en Vak. Ze zijn niet uitgedrukt. Deze zin is waar in de databank asa elk object in het domein, inclusief elke score, studente en vak, een cursus doceert waarvoor Jack ingeschreven is. Dat is niet wat bedoeld wordt en deze zin zal ook nooit voldaan zijn. De correcte query is, poging 2:

$$(\forall y)[(\exists w)Instructor(y, w) \Rightarrow (\exists w)(Instructor(y, w) \wedge Enrolled(Jack, w))]$$

Je ziet dat we de definiërende formule voor *Docent* hebben toegevoegd.

Dus soms moeten we type-informatie toevoegen aan een query en soms niet. Hoe komt dat? En hoe kunnen we zien of we al dan niet type-informatie moeten toevoegen? Dat is belangrijk, niet alleen in de context van databanken, maar in de context van alle modelleertoepassingen van logica. Dat zullen we nu nader bestuderen.

Er is een eenvoudige manier om te herkennen of je query goed getypeerd is. Deze is gebaseerd op de type-integriteitsbeperkingen en de volgende tautologie:

$$(P \Rightarrow Q) \Rightarrow (P \Leftrightarrow (P \wedge Q))$$

In een goed getypeerde databank waarin dus de type-integriteitbeperkingen voldaan zijn,

$$(\forall x_1) \ldots (\forall x_n)(P(x_1, \ldots, x_n) \Rightarrow T_1(x_1) \wedge \cdots \wedge T_n(x_n))$$

mogen we elk atoom $P(t_1, \ldots, t_n)$ vervangen door $T_1(t_1) \wedge \ldots T_n(t_n) \wedge P(t_1, \ldots, t_n)$, waarbij T_i het type-predikaat is van het i-de argument van P. Deze transformatie zullen we de *type-insertie* noemen. Als we dan een expliciet getypeerde formule bekomen (of een formule die gemakkelijk kan omgezet worden in de expliciet getypeerde formule) dan is de query in orde.

Voorbeeld 4.3.10. Voor de query

$$(\exists c)Grade(Sam, c, AAA)$$

krijgen we :

$$(\exists c)(Vak(c) \wedge Student(Sam) \wedge Score(AAA) \wedge Grade(Sam, c, AAA))$$

welke equivalent is met de expliciet getypeerde query:

$$(\exists c)(Vak(c) \wedge Grade(Sam, c, AAA))$$

Voorbeeld 4.3.11. De query

$$\{x : \neg Prerequ(x, x)\}$$

wordt getransformeerd in

$$\{x : \neg(Vak(x) \wedge Prerequ(x, x))\}$$

wat zeker niet equivalent is met de expliciet getypeerde query:

$$\{x : Vak(x) \wedge \neg Prerequ(x, x)\}.$$

Voorbeeld 4.3.12. De query

$$(\forall y)(\exists w)(Instructor(y, w) \wedge Enrolled(Jack, w))$$

wordt door type-insertie:

$$(\forall y)(\exists w)(Docent(y) \wedge Vak(w) \wedge Student(Jack) \wedge Instructor(y, w) \wedge Enrolled(Jack, w))$$

wat niet equivalent is met de expliciet getypeerde query:

$$(\forall y)(Docent(y) \Rightarrow (\exists w)(Vak(w) \wedge Instructor(y, w) \wedge Enrolled(Jack, w))$$

Definitie 4.3.7. Een **impliciet getypeerde query** is er een zonder type-predikaten (dus enkel databankpredicaten) die logisch equivalent is met zijn expliciet getypeerde query modulo de type-integriteitsbeperkingen.

Modulo de type-integriteitsbeperkingen? Dat betekent dat in alle modellen van de type-integriteits-beperkingen (zoals bv. $(\forall x)(\forall y)(Enrolled(x, y) \Rightarrow Student(x) \wedge Vak(y))$) beide formules dezelfde waarheidswaarde hebben.

Impliciet getypeerde queries zullen correct getypeerde antwoorden teruggeven, tenminste als we ervan uit mogen gaan dat de type-integriteitsbeperkingen voldaan zijn. Queries die niet impliciet getypeerd zijn zullen nonsens als antwoord geven.

Voorbeeld 4.3.13. Een impliciet getypeerde query zal toch slecht getypeerde antwoorden teruggeven indien de databank niet goed getypeerd is. Bv. als we het tuple $(Bill, AAA, AAA)$ toevoegen aan $Grade^{\mathfrak{A}}$, dan zal de impliciet getypeerde query $\{c : Grade(Bill, c, AAA)\}$ het slecht getypeerde antwoord $\{AAA\}$ terug geven.

Om te bepalen of een query impliciet getypeerd is kunnen we nu gewoon een theoremprover oproepen (zoals Spass in LogicPalet).

Voorbeeld 4.3.14. Indien de query $\{x : Prerequ(x, x)\}$ impliciet getypeerd is zou Spass het volgende moeten kunnen bewijzen:

$$\begin{array}{l}(\forall x)(\forall y)(Prerequ(x, y) \Rightarrow Vak(x) \wedge Vak(y)) \\ \models \\ (\forall x)(\neg Prerequ(x, x) \Leftrightarrow (Vak(x) \wedge \neg Prerequ(x, x)))\end{array}$$

Spass kan dit niet bewijzen, dus de query is niet impliciet getypeerd.

Maar er is een nuttig en belangrijk syntactisch criterium om zo'n queries te herkennen.

Definitie 4.3.8. Een **veilige query** $\{(x_1, \ldots, x_n) : A\}$ voldoet aan de volgende voorwaarden:

- A is een atomaire formule $P(\ldots)$ die elke variabele $x_1, \ldots, x_n$ bevat. Of A is een conjunctie zodat elke x_i in een atomaire conjunct $P(\ldots)$ voorkomt.

- Voor elke deelformule $(\exists x_1)\dots(\exists x_n)B$ geldt een soortgelijke voorwaarde: B is een atomaire formule $P(\dots)$ die elke variabele $x_1, \dots, x_n$ bevat. Of B is een conjunctie zodat elke x_i in een atomaire conjunct $P(\dots)$ voorkomt.
- Voor een universele formule $(\forall x_1)\dots(\forall x_n)B$ is B een implicatie $C \Rightarrow D$ zodat C een atoom $P(\dots)$ is waarin elke x_i voorkomt, of het is een conjunctie zodat elke x_i in een atomaire conjunct $P(\dots)$ van C voorkomt.

Voorbeeld 4.3.15.

- $(\exists c)Grade(Sam, c, AAA)$ is een veilige query.
- $\{x : \neg Prerequ(x, x)\}$ is geen veilige query.
- $(\forall y)(\exists w)(Instructor(y, w) \wedge Enrolled(Jack, w))$: is geen veilige query omwille van $(\forall y)$.

Eigenschap 4.3.2. Een veilige query is impliciet getypeerd.

We zullen deze eigenschap niet bewijzen, maar het is niet moeilijk om dit in te zien. Als je de type-predikaten toevoegt aan een veilige query, dan zie je zo dat voor elke variabele een type-predicaat op een juiste plaats staat: in een conjunctie voor een existentiële variabele, en in de premisse van een implicatie voor een universele variabele.

In hedendaagse databanken worden queries beperkt tot *veilige (safe)* queries. Zoals we eerder besproken hebben heeft dat ook nog andere voordelen: veilige queries zijn domeinonafhankelik en kunnen veel efficiënter berekend worden.

Oefening 4.3.6. *Ga na of de volgende query goed getypeerd is:*

$$(\forall y)[(\exists w)Instructor(y, w) \Rightarrow (\exists w)(Instructor(y, w) \wedge Enrolled(Jack, w))]$$

Oefening 4.3.7. *Formuleer volgende queries en integriteitsbeperkingen voor de Studenten-Databank. Let op of uw query impliciet getypeerd is, en zoniet, voeg expliciete typering toe. Je mag daarvoor de type-predikaten gebruiken; immers het is niet moeilijk om nadien een type-atoom zoals $Vak(x)$ te vervangen door zijn definiërende formule.*

1. *Zijn alle studenten die ingeschreven zijn voor CS230 geslaagd voor dat vak? (Opgepast: een student kan meerdere scores voor een vak hebben! Zijn beste score telt.)*
2. *Is er een student die ingeschreven is voor alle vakken die Sue doceert? Uw query moet ook juist zijn als Sue geen enkel vak doceert.*
3. *Is er een student die ingeschreven is voor meer dan twee vakken?*
4. *Zijn Ray en Hec docenten voor CS230 en tevens de enigen hiervoor?*

5. *Gevraagd: elke student die geslaagd is voor alle door hem afgelegde vakken? (Opgepast: een student kan meerdere scores voor een vak hebben! De hoogste telt.)*

6. *Gevraagd: alle docenten die meer dan één vak doceren.*

7. *Integriteitsbeperking: Niemand mag ingeschreven zijn voor een vak waarvoor hij/zij al geslaagd is.*

8. *Integriteitsbeperking: Iedereen die ingeschreven is voor een vak moet geslaagd zijn voor alle vakken die (directe) nodige voorkennis zijn voor dat vak.*

Oefening 4.3.8. *1. Toon aan door te rekenen met logische equivalenties dat de volgende twee queries logisch equivalent zijn:*

$$\{x : \ (\exists y)[Enrolled(x,y) \vee (\exists z)Grade(x,y,z)] \wedge \\ (\forall y)(\forall z)[Grade(x,y,z) \Rightarrow PassingGrade(z)]\} \tag{4.1}$$

$$\{x : (\forall z)(\exists t)[\ [(\exists y)Enrolled(x,y) \vee (\exists s)Grade(x,s,t)] \wedge \\ [(\exists y)Grade(x,y,z) \Rightarrow PassingGrade(z)]]\} \tag{4.2}$$

2. *Verifieer dat ook met AskSpass van LogicPalet. (Dit stelt geen probleem alhoewel x een vrije variabele is.)*

3. *Formuleer de query 4.1 zo kort mogelijk in het Nederlands.*

4. *Welke van de twee queries zal het efficiënts uitgevoerd worden door het query-algoritme?*

5. *Formuleer volgende query zo kort mogelijk in het Nederlands:*

$$\{x : \ [(\exists y)Enrolled(x,y) \vee (\exists y)(\exists z)Grade(x,y,z)] \wedge \\ (\forall y)[(\exists z)Grade(x,y,z) \Rightarrow (\exists z)(Grade(x,y,z) \wedge PassingGrade(z))\} \tag{4.3}$$

6. *Toon aan door een redenering dat (4.1) en (4.3) logisch equivalent zijn modulo de volgende integriteitsbeperking*

$$\neg(\exists x)(\exists w)(\exists s)(\exists t)[Grade(x,w,s) \wedge Grade(x,w,t) \wedge s \neq t]. \tag{4.4}$$

(Spass verifieert dit in minder dan 0,1 seconde!)

Oefening 4.3.9. *Voor queries met geneste kwantoren zijn er soms meerdere redeneerstappen nodig om ze na de type-insertie om te zetten in een equivalente expliciet getypeerde formule. Neem de query of er studenten bestaan die van elke docent een cursus volgen:*

$$(\exists x)(\forall y)[(\exists w)Instructor(y,w) \Rightarrow (\exists w)(Instructor(y,w) \wedge Enrolled(x,w)))]$$

Doe de type-insertie en toon aan dat je die kunt omzetten naar een expliciet-getypeerde query.

Opmerking 4.3.6. Van deze sectie moet je onthouden wat expliciet en impliciet getypeerde queries zijn, en je moet kunnen herkennen door middel van de type-transformatie of een query impliciet getypeerd is.

4.3.6 Definities

In vrijwel alle niet triviale modelleertoepassingen blijken bepaalde concepten op te duiken die weliswaar uit te drukken zijn in termen van het bestaande vocabularium Σ door middel van een definiërende formule, maar die zo vaak voorkomen dat het nuttig zou zijn om een nieuw predikaatsymbool ervoor te introduceren.

- De definiërende formules van de types in de studentendatabank waren een voorbeeld. Bv., de definiërende formules voor $Vak(v)$:

  $$(\exists t)Instructor(t, v)$$

  Of die voor $Docent(t)$:

  $$(\exists v)Instructor(t, v)$$

- Student x is geslaagd voor vak v

  $$(\exists s)(Grade(x, v, s) \wedge PassingGrade(s))$$

Het is mogelijk om, in elke query waarin die afgeleide concepten voorkomen, telkens zijn definiërende formule in te voegen, maar dat maakt formules lang en onoverzichtelijk, en het is een bron van fouten (mensen zijn niet geschikt om lange formules uit te schrijven). De oplossing hiervoor is om expliciete *definities* toe te voegen aan de taal. Definities drukken de betekenis van een nieuw predikaatsymbool uit in termen van de bestaande, t.t.z. in termen van basisconcepten. Nadien kunnen de nieuwe symbolen vrij gebruikt worden in formules en/of queries.

In predikatenlogica kunnen we definities uitdrukken met behulp van het equivalentiesymbool. Bv.

$$\begin{aligned}(\forall x)(\forall v)[GeslaagdVoor(x, v) &\Leftrightarrow (\exists s)Grade(x, v, s) \wedge PassingGrade(s))]\\ (\forall v)[Vak(v) &\Leftrightarrow (\exists x)Instructor(x, v)]\\ (\forall v)[BasisVak(v) &\Leftrightarrow Vak(v) \wedge \neg(\exists v1)Prerequ(v1, v)]\end{aligned}$$

Door het introduceren van nieuwe, gedefinieerde predikaten worden veel queries eenvoudiger.

Voorbeeld 4.3.16. De query of er studenten zijn die van elke docent een vak hebben gehad waarvoor ze geslaagd zijn:

$$(\exists x)(\forall y)[(\exists w)Instructor(y, w) \Rightarrow (\exists w)(Instructor(y, w) \wedge (\exists s)(Grade(x, y, s) \wedge Passinggrade(s)))]$$

wordt:

$$(\exists x)(\forall y)[Docent(y) \Rightarrow (\exists w)(Instructor(y, w) \wedge GeslaagdVoor(x, w))]$$

Definities worden in veel formele talen gebruikt. Vaak krijgen ze een andere naam, en worden andere notaties gebruikt, maar ze staan voor hetzelfde logische concept: een definitie[1]. In databanken worden gedefinieerde predikaatsymbolen *view-predikaten* genoemd.

[1] Terwijl logici misschien soms wat te weinig notaties gebruiken (zoals $\models$) heeft men in de informatica duidelijk last van het omgekeerde: in elk domein krijgt hetzelfde logische concept een andere naam en een andere notatie. Op die manier zie je tenslotte door de bomen het bos niet meer.

Opmerking 4.3.7. De rest van deze sectie gaat over nieuwe vormen van redeneren in de context van definities en over inductieve definities. Dit is ter informatie en moet je niet kennen voor het examen.

In de context van databanken met definities/views stellen zich nieuwe redeneerproblemen.

Bereken een query die gedefinieerde predicaten bevat in een structuur $\mathfrak{A}$.

Bereken de interpretatie van een gedefinieerd predicaat in een structuur $\mathfrak{A}$.

Beide taken zijn gemakkelijk op te lossen: bv. we kunnen een gedefinieerd predicaat altijd vervangen door zijn definiërende formule. In de databankwereld wordt het tweede ook wel *view materialisation* genoemd.

Wanneer de interpretatie van een gedefinieerd predicaat bijgehouden wordt door een databanksysteem, dan moet wanneer de basispredikaten gewijzigd worden ook de interpretatie van de gedefinieerde predikaten aangepast worden. Bv. in de huidige studentendatabank zou Sam in de tabel van een excellente studenten zitten (enkel AAA's en AA's). Als we *Grade* uitbreiden met $(Sam, M200, B)$ moet Sam uit die tabel geschrapt worden.

Pas de interpretatie van een gedefinieerd predicaat aan na een wijziging van de structuur.

Een slimme manier om dat te doen zal niet heel de tabel opnieuw uitrekenen, maar de bestaande tabel aanpassen. In de databank wereld noemt men dit wel eens *incremental update of materialized views.*

Oefening 4.3.10. *Veronderstel dat we de relatie van tuples van studenten en vakken (s, v) hebben berekend zodat student s geslaagd is voor vak v. Veronderstel dat we de studentengegevensbank aanpassen: we voegen nieuwe koppels toe aan of verwijderen bestaande koppels uit de 5 basisrelaties. Ga na hoe dit de tabel van de geslaagde examens beinvloedt.*

- *Bv. heeft toevoegen of verwijderen van een koppel in Enrolled een invloed?*
- *Heeft toevoegen of verwijderen van een koppel uit Grade een invloed? En zo ja, welke?*

Inductieve definities In deze cursus (net als in vele andere cursussen met wiskundige inslag) staat het vol van definities, maar dan niet in logica uitgedrukt (bv. definitie van logische waarheid of logische gevolg in termen van waarheid in een structuur). In wezen is de rol van definities in zo'n wiskundige tekst niet anders dan de rol van een definitie in een databanksysteem of een andere informaticatoepassing: het creëren van een nieuw concept in termen van de bestaande.

In deze cursus komen verschillende definities voor van een type dat niet in SQL databankensystemen kan uitgedrukt worden: *inductieve definities.* Voorbeelden zijn: definitie van wat een term is, wat een formule is, definitie van de interpretatie van termen $t^{\mathfrak{A}}$, definitie van waarheid $\mathfrak{A} \models A$.

Je kunt je afvragen hoe zo'n definities in logica uitgedrukt kunnen worden, en of er nuttige toepassingen zijn voor dit type van definitie.

We hebben al twee concrete situaties gezien waar een inductieve definitie van toepassing zou zijn. De eerste is om te definiëren wanneer een cursus c directe of indirecte voorkennis is van een andere cursus c'? Dat is wanneer (c, c') in de $Prerequ$ tabel zit, maar meer algemeen als er tuples $(c, c_1), (c_1, c_2), \ldots, (c_n, c')$ in de tabel zitten, voor willekeurige $n, c_1, \ldots, c_n$. We kunnen dit uitdrukken door de volgende inductieve definitie:

Definitie 4.3.9. We zeggen dat cursus c (indirecte) voorkennis is van vak c' als de volgende inductieve regels gelden:

- als c directe voorkennis is van c' - dat is het basisgeval;
- als er een vak c_1 bestaat zodat c directe voorkennis is van c_1 en c_1 is (indirecte) voorkennis van c'.

Een analoge toepassing van een inductieve definitie is om te definiëren wat een voorouder is: een ouder (het basisgeval) of een ouder van een voorouder (het inductieve geval).

In SQL kunnen we geen inductieve definities neerschrijven. In de cursus "Modelleren van complexe systemen" in de eerste master, zullen we bewijzen dat dit ook onmogelijk is in predikatenlogica. In sommige andere logische talen is het wel mogelijk. Bv. in de taal van het IDP-systeem dat in LogicPalet zit, kan men deze inductieve definitie formeel uitschrijven:

$$\left\{ \begin{array}{l} (\forall x)(\forall y)(Voorkennis(x,y) \leftarrow Prerequ(x,y)), \\ (\forall x)(\forall y)(Voorkennis(x,y) \leftarrow (\exists z) Prerequ(x,z) \wedge Voorkennis(z,y)) \end{array} \right\}$$

Het ziet eruit als twee implicaties, maar dat is schijn. Hier staat dat de relatie $Voorkennis$ bekomen wordt door iteratief deze regels toe te passen. Anders gezegd, het is de kleinste relatie die voldoet aan deze twee regels. Als dit enkel twee implicaties zou zijn, dan zou er staat dat $Voorkennis$ om het even welke relatie is die aan deze voorwaarden voldoet.

4.3.7 SQL

Bijna alle commerciële databanksystemen gebruiken de SQL-taal voor het formuleren van queries. SQL is gebaseerd op de principes van de predikatenlogica maar heeft een totaal andere syntax. Toch is het niet moeilijk het verband met logica te zien. In 2Ba is er een afzonderlijk vak over databanken, met alle details over SQL. We beperken ons hier tot enkele voorbeelden en een bespreking van het verband met logica.

Een eerste belangrijk verschil is dat in een SQL database het vocabularium (het *database-schema* genoemd) gegeven wordt in de vorm van een aantal tabel-declaraties waarin ook de kolommen een naam krijgen. Ze zijn van de vorm

$$Tab(TabVeld_1, TabVeld_2, \ldots, TabVeld_n)$$

waarbij Tab de naam van de tabel is, en $TabVeld_i$ de naam van de i-de kolom van de tabel. Vaak kiest men het type van die positie als naam, maar dat is niet altijd mogelijk.

Voorbeeld 4.3.17. In een SQL-databank zouden we $Grade/3$ en $Enrolled/2$ kunnen declareren als $Grade(student, vak, score)$ en $Enrolled(student, score)$. De relatie $Grafe/2$ kun je declareren als $Grafe(vertex1, vertex2)$.

In SQL-queries staan er verwijzingen zowel naar de tabelnaam als naar de kolomnamen. In SQL wordt de existentiële kwantor aangeduid met EXISTS. Op het eerste zicht een belangrijk verschil met logica is dat deze operator loopt over de rijen van een tabel, niet over de elementen van het databankdomein. Er bestaat geen universele kwantor. Deze moet je uitdrukken met NOT EXISTS NOT.

Voorbeeld 4.3.18. Alle koppels (x,y) zodat student x geslaagd is voor vak y.

$$\{(x, y) : (\exists z)(Grade(x, y, z) \wedge PassingGrade(z))\}$$

De SQL-query:

$$\begin{array}{l} SELECT\ student, vak \\ FROM\ Grade \\ WHERE\ EXISTS\ (SELECT\ score \\ FROM\ PassingGrade \\ WHERE\ Grade.score = score) \end{array}$$

Deze query selecteert de waardes uit de kolommen *student* en *vak* in rijen van *Grade* waarvan de waarde in de kolom *score* behoort tot de tabel *PassingGrade*.

Het verband met logica. Het verband tussen SQL-databanken en logica is niet eenduidig. T.t.z., er zijn meerdere mogelijkheden om een SQL-databank te zien als een logische databank, namelijk, dit is afhankelijk van hoe we het logische vocabularium Σ afleiden uit het databaseschema:

- De "klassieke" manier: we kiezen een n-voudig predikaatsymbool per tabel met n kolommen.
- De "object georiënteerde" manier: we kiezen de tabelnaam Tab als een unair type-predikaat en de kolomnamen $TabVeld_i$ als 2-voudige relaties.

Dit heeft natuurlijk een zeer grote impact op hoe SQL-queries vertaald worden in logica.

Voorbeeld 4.3.19. De SQL-tabel $Grade(student, vak, score)$ kunnen we voorstellen als een 3-voudige relatie zoals in de studentendatabank. Het alternatief is om deze tabel te beschouwen als verzameling van relaties in Figuur 4.3.19 voor de predikaat-symbolen $Grade/1$, $GradeStudent/2$, $GradeVak/2$ en $GradeScore/2$. In deze tabellen staan de objecten G_i voor de koppels in de oorspronkelijke tabel. Je kunt ze interpreteren als individuele examens van studenten.

In het nieuwe logische vocabularium wordt de query voor alle koppels (x, y) zodat student x geslaagd is voor vak y:

$$\begin{array}{l} \{(x, y) : (\exists g)[\ Grade(g) \wedge GradeStudent(g, x) \wedge GradeVak(g, y) \wedge \\ \qquad (\exists s)(Passinggrade(s) \wedge GradeScore(g, s))]\} \end{array}$$

Deze query heeft dezelfde structuur als de SQL query. Merk op dat de existentiële quantor $(\exists g)$ nu ook loopt over wat voorheen "rijen" waren. Deze query kan vereenvoudigd worden door het type-predikaat te laten vallen:

$$\{(x, y) : (\exists g)[GradeStudent(g, x) \wedge GradeVak(g, y) \wedge (\exists s)(Passinggrade(s) \wedge GradeScore(g, s))]\}$$

Grade		
Sam	CS148	AAA
Bill	CS148	D
Jill	CS148	A
Jack	CS148	C
Flo	CS230	AA
May	CS230	AA
Bill	CS230	F
Ann	CS230	C
Jill	M100	B
Sam	M100	AA
Flo	M100	D
Flo	M100	B

$\longrightarrow$

$\mathbf{Grade}^{\mathfrak{A}}$
G1
G2
G3
G4
G5
G6
G7
G8
G9
G10
G11
G12

$\mathbf{GradeStudent}^{\mathfrak{A}}$	
G1	Sam
G2	Bill
G3	Jill
G4	Jack
G5	Flo
G6	May
G7	Bill
G8	Ann
G9	Jill
G10	Sam
G11	Flo
G12	Flo

$\mathbf{GradeVak}^{\mathfrak{A}}$	
G1	CS148
G2	CS148
G3	CS148
G4	CS148
G5	CS230
G6	CS230
G7	CS230
G8	CS230
G9	M100
G10	M100
G11	M100
G12	M100

$\mathbf{GradeScore}^{\mathfrak{A}}$	
G1	AAA
G2	D
G3	A
G4	C
G5	AA
G6	AA
G7	F
G8	C
G9	B
G10	AA
G11	D
G12	B

Figuur 4.2: Een database als alternatieve structuur

Wat is nu de beste manier om een SQL-databank te vertalen in een logische structuur? Of wat is de beste manier om het logisch vocabularium te kiezen. Dit hangt af van het aantal kolommen in de SQL-tabel! Hoe groter, hoe nuttiger het wordt om voor het alternatieve vocabularium te kiezen.

Het heeft totaal geen zin[2] om de alternatieve manier toe te passen op 1-voudige en 2-voudige tabellen zoals *PassingGrade*, *Enrolled*, *Prerequ*, *Instructor*. 3-voudige tabellen zijn een tussengeval zoals we net zagen. Maar voor meer kolommen wordt het steeds nuttiger om het alternatieve vocabularium te gebruiken.

In databanken werd het gebruikelijk om tabellen met zeer veel kolommen te introduceren. Stel u een tabel voor om de werknemers in een bedrijf voor te stellen: *Werknemer(id,voornaam, naam, straat, straatnr, gemeente, zip, geboortedatum, job, loon, anciënniteit).* Dit is een relatie met 11 argumenten. Veronderstellen dat we de query willen stellen naar de werknemers die minstens 100.000 euro per jaar verdienen. In SQL wordt dit

$$\begin{array}{l} SELECT\ id \\ FROM\ Werknemer \\ WHERE\ Werknemer.loon \geq 100.000; \end{array}$$

Als we deze tabel modelleren met een 11-voudig predikaatsymbool $Werknemer$ wordt de query:

$$\begin{array}{l} \{id: \ (\exists vn)(\exists n)(\exists st)(\exists stnr)(\exists g)(\exists z)(\exists geb)(\exists j)(\exists l)(\exists a) \\ \qquad (Werknemer(id, vn, n, st, stnr, g, z, geb, j, l, a) \wedge l > 100.000)\} \end{array}$$

Men zou voor minder een indigestie krijgen. Als we daarentegen kiezen voor de alternatieve formulering krijgen we:

$$\{id : (\exists w)[WerknemerId(w, id) \wedge (\exists l)(WerknemerLoon(w, l) \wedge l \geq 100.000)]\}$$

Het gebeurt zeer frequent dat een SQL-tabel een kolom bevat die voor elke rij een unieke identifier voor die rij bevat. Zo'n kolommen krijgen vaak de naam "id" of zoiets. Men noemt dergelijke kolommen een *primary key* van de tabel. In zo'n geval kunnen we vanzelfsprekend deze waarde gebruiken in plaats van nieuwe objecten aan te maken voor elke rij.

Voorbeeld 4.3.20. In de Werknemer tabel is de eerste kolom *id* zo'n primary key. We kunnen deze tabel modelleren met 1 (redelijk overbodig) type-predikaatsymbool $Werknemer/1$ die bestaat uit de primary keys en 10 2-voudige predicaten. De query wordt:

$$\{id : (\exists l)(WerknemerLoon(id, l) \wedge l \geq 100.000)]\}$$

[2]Bv. de tabel voor *PassingGrade* zou omgezet worden naar een type-predikaat *PassingGrade* met interpretatie $\{pg1, \ldots, pg5\}$ en een 2-voudig predikaatsymbool met interpretatie $\{(pg1, AAA), \ldots, (pg5, C)\}$.

Conclusie: het kiezen van een vocabularium. Zoals we gezien hebben heeft de keuze van het vocabularium een enorme impact op de vorm en elegantie van de logische zinnen en queries.

De lessen die we hier geleerd hebben over het kiezen van een vocabularium zijn niet enkel van toepassing in de context van databanken maar in modelleringsproblemen in het algemeen. Kiezen we één groot predicaatsymbool met meerdere argumenten of introduceren we nieuwe types en splitsen we op in kleinere predicaten? Welke types van objecten beschouwen we? Men moet hierin een gulden middenweg zoeken met als doel dat logische zinnen zo compact en elegant mogelijk zijn.

We hebben gezien dat te grote predikaten beter opgesplitst worden, maar ook dat teveel opsplitsing (zoals bij $Grade/3$) evenmin goed is. Voor de studentdatabank geldt dat de oorspronkelijke modellering van de databank met $PassingGrade/1$, $Enrolled/2$, $Prerequ/2$, $Instructor/2$ wellicht de beste keuze was.

Oefening 4.3.11. *De OnderdelenVerdelersDatabank bestaat uit volgende tabellen:*

- $Ver(id, naam, stad)$ *met respectievelijk de identifier (een primary key), de naam en de stad van een verdeler.*
- $Ond(id, naam, kleur, stad)$*, met kolommen die de identifier (een primary key), naam, kleur en stand (waar het onderdeel opgestapeld is) van het onderdeel.*
- $Pro(id, naam, stad)$ *met id (primary key), naam en stad van een project dat onderdelen vergt.*
- $VOP(verdeler\_id, onderdeel\_id, project\_id)$*: wie levert wat aan welk project.*

Kies een geschikt logisch vocabularium, en formuleer volgende queries in predikatenlogica:

1. *Gevraagd: alle id's van onderdelen geleverd door een verdeler in London aan een project in London.*
2. *Gevraagd: alle id's van projecten waaraan minstens één verdeler uit een andere stad levert.*
3. *Gevraagd: alle id's van projecten waaraan geen enkele verdeler in London een rood onderdeel levert.*
4. *Gevraagd: alle id's van onderdelen die geleverd worden aan alle projecten in London.*
5. *Gevraagd: alle id's van verdelers die eenzelfde onderdeel leveren aan alle projecten.*

Opmerking 4.3.8. Je moet geen SQL kennen voor het examen. Wat je wel moet kennen zijn de klassieke en de object georiënteerde manier om te modelleren, en het effect van deze keuze op de queries. De rest van deze sectie is ter informatie, voor wie meer wil weten over SQL en het verband met logica.

Voorbeeld 4.3.21. Gevraagd: Alle excellente studenten. Daarmee bedoelen we studenten die minstens één examen afgelegd hebben en op alle afgelegde examens een AA of AAA behaalden.

$$\{x : (\exists y)(\exists z)Grade(x, y, z) \wedge (\forall y)(\forall z)[Grade(x, y, z) \Rightarrow z = AA \vee z = AAA]\}$$

Voor deze query worden in SQL twee rij-variabelen gebruikt, $Gr1$ en $Gr2$:

$SELECT\ student$
$FROM\ Grade\ AS\ Gr1$
$WHERE\ NOT\ EXISTS\ (\ SELECT\ score$
$FROM\ Grade\ AS\ Gr2$
$WHERE\ Gr1.student = Gr2.student\ AND\ NOT\ (score = \text{'AA'}\ OR\ score = \text{'AAA'}))$

Deze rij-variabelen staan voor rijen in een tabel. De query selecteert de waarde in het veld *student* in rijen $Gr1$ van de tabel *Grade* waarvoor de volgende tabel leeg is: de tabel bestaande uit de scores in rijen $Gr2$ in de tabel *Grade* waarvoor geldt dat $Gr1$ en $Gr2$ dezelfde student bevatten, en de score (in $Gr2$) niet minstens AA is.

Deze SQL query lijkt nu heel sterk op de volgende impliciet getypeerde query:

$$\{st : (\exists gr1)[\ GradeStudent(gr1, st) \wedge (\forall sc)(\forall gr2)(\ GradeScore(gr2, sc) \wedge GradeStudent(gr2, st) \Rightarrow (sc = \text{'AA'} \vee sc = \text{'AAA'}))]\}$$

De query in termen van het "klassieke" vocabularium was echter nog compacter en eenvoudiger:

$$\{x : (\exists y)(\exists z)Grade(x, y, z) \wedge (\forall y)(\forall z)[Grade(x, y, z) \Rightarrow z = AA \vee z = AAA]\}$$

Voorbeeld 4.3.22. Veronderstel in SQL twee tabellen $suppliers(supplier\_id, supplier\_name)$ en $orders(supplier\_id, quantity, price)$. De kolom $supplier\_id$ is een primary key. Hoe zetten we dit om naar een logisch vocabularium?

Voor de eerste tabel kiezen we twee predikaatsymbolen $Supplier/1$ en $SupplierSupplier\_name/2$. In de structuur geassocieerd met deze databank bestaat de interpretatie van *Suppliers* uit alle supplier-id's, t.t.z. de primary keys uit de eerste kolom. $SupplierSupplier\_name/2$ linkt supplier-id's met een supplier naam.

Voor *orders*, die geen primary key heeft, creëren we een nieuw type $Orders/1$ en de drie relaties $OrdersSupplier\_id/2, OrdersPrice/2$ en $OrdersQuantity/2$. De interpretatie van deze zijn zoals eerder werd aangegeven.

De volgende SQL-query vraagt om de prijs en hoeveelheid van de orders die door IBM geleverd worden:

$SELECT\ orders.quantity, orders.price$
$FROM\ orders$
$WHERE\ EXISTS(SELECT\ suppliers.supplier\_id$
$FROM\ suppliers$
$WHERE\ suppliers.supplier\_name =' IBM'$
$and\ suppliers.supplier\_id = orders.supplier\_id);$

In het logisch vocabularium wordt dit uitgedrukt als volgt:

$$\{(y, z) : (\exists o)[\ OrdersQuantity(o, y) \wedge OrdersPrice(o, z) \wedge (\exists x)(\ OrdersSupplier\_id(o, x) \wedge SuppliersSupplier\_name(x,' IBM'))]\}$$

SQL in het echt. Historisch gezien is SQL gegroeid uit logica en deze taal omvat een deeltaal die beschouwd kan worden als een syntactische variant van logica. Wellicht behoort de grote meerderheid van SQL queries die in de praktijk gesteld worden tot die deeltaal. Er wordt vaak aangenomen dat SQL veel intuïtiever is dan logica. Er zijn op zijn minst twee (slechte) redenen hiervoor. Ten eerste hebben 99% van de werkende informatici wel SQL gestudeerd maar geen of maar weinig logica. Ten tweede wordt SQL vrijwel altijd vergeleken met de "klassieke" vertaling naar logica. In elk geval, wie logica goed beheerst zal zeer snel ook de SQL syntax begrijpen. Ik ben benieuwd naar wat jullie studenten hier volgend jaar over zullen denken nadat jullie het vak over databanken hebben gehad.

In welk opzicht is SQL uitgebreid in vergelijking met predikatenlogica? De lijst is lang, maar om één voorbeeld te geven: een soort uitdrukking die zeer lastig uit te drukken is in predikatenlogica is "er zijn minstens (of hoogstens, of exact) n objecten waarvoor eigenschap A geldt". Dit is een tekort want dergelijke uitspraken komen frequent voor in informaticatoepassingen. In SQL kan men dergelijke uitspraken wel formuleren.

Om een expressieve modelleertaal te verkrijgen uit predikatenlogica moet deze uitgebreid worden. Het IDP-systeem, dat in LogicPalet gebruikt wordt, ondersteunt een sterk uitgebreide versie van predikatenlogica. Je kan bv. schrijven:

$$(\forall v)(\forall a)(CursusAuditorium(v,a) \Rightarrow \#\{x : Enrolled(x,v)\} \leq Capaciteit(a))$$

om uit te drukken dat elk vak door moet gaan in een auditorium met voldoende capaciteit. Het relatievoorschrift $\{x : Enrolled(x,v)\}$ duidt de verzameling van studenten van het vak v aan, en $\#\{x : Enrolled(x,v)\}$ is het aantal elementen van deze verzameling.

4.4 De oudste logische theorie: de Peano axioma's

Peano was een 19de eeuwse logicus en wiskundige die in 1898 een logische studie publiceerde over de natuurlijke getallen. De bedoeling van zijn theorie was om de structuur $\langle \mathbb{N}, 0, S/1, +, \times \rangle$ bestaande uit de natuurlijke getallen, met 0, de successor functie (die n op $n+1$ afbeeldt), som en product te beschrijven in logica.

We kunnen deze structuur als een soort oneindige databank bekijken, maar met functies in plaats van relaties (wat overigens geen groot verschil is). Dan was Peano's bedoeling gelijk aan die van de theorie $Theo(DB)$ van de vorige sectie, namelijk een theorie te construeren die alle informatie in de structuur bevat, en die hopelijk maar één model heeft (modulo isomorphisme), namelijk de natuurlijke getallen zelf. Peano's bedoeling was dus om $Theo(\mathbb{N})$ te construeren.

Beschouw Σ_P bestaande uit een constante $zero$ en drie functiesymbolen $Succ/1$, $Plus/2$ en $Maal/2$. Beschouw de volgende stuctuur $\mathfrak{A}$ van Σ_P.

- $Dom_{\mathfrak{A}} = \mathbb{N}$,
- $zero^{\mathfrak{A}}$ is het getal 0,
- $Succ^{\mathfrak{A}} = \mathbb{N} \to \mathbb{N} : n \to n+1$,
- $Plus^{\mathfrak{A}} = \mathbb{N}^2 \to \mathbb{N} : (n,m) \to n+m$,
- $\times^{\mathfrak{A}} = \mathbb{N}^2 \to \mathbb{N} : (n,m) \to n \times m$.

We zoeken dus een theorie in Σ_P om deze structuur te beschrijven, net als daarnet voor een databank-structuur $\mathfrak{A}$.

Het is gebruikelijk om de Peano's axioma's uit te drukken in de standaard symbolen van de

natuurlijke getallen zelf. Daar zullen we ons ook aan houden. Dus, in plaats van

$$Plus(zero, Succ(zero)) = Succ(zero)$$

schrijven we

$$0 + S(0) = S(0).$$

Dat betekent dat we nu hetzelfde wiskundig symbool dubbel gebruiken, enerzijds als een niet-logisch symbool uit Σ_P, anderzijds als zijn interpretatie. M.a.w., $0^{\mathfrak{A}} = 0$, $S^{\mathfrak{A}} = S$, $+^{\mathfrak{A}} = +$, $\times^{\mathfrak{A}} = \times$. Let goed op!

Een andere notatie is dat we een term $\underbrace{S(S(\ldots S}_{n \text{ keer } S}(0)\ldots))$ noteren als $S^n(0)$. Merk op dat de $(S^n(0))^{\mathfrak{A}} = n$.

Net als voor een databank zijn er drie soorten axiomas:

- UNA: we willen hier uitdrukken dat $0, S(0), S(S(0)), \ldots, S^n(0), \ldots$ allemaal verschillend zijn. Dit kunnen we doen door een oneindige verzameling van axioma's $\neg S^n(0) = S^m(0)$ voor elke $n \neq m \in \mathbb{N}$. Peano vond echter een eindige voorstelling van deze oneindige verzameling:
$$(\forall n)\neg 0 = S(n)$$
$$(\forall n)(\forall m)(S(n) = S(m) \Rightarrow n = m)$$

- DCA: we willen uitdrukken dat elk object gelijk is aan één van de termen $0, S(0), S^2(0), \ldots$. Het idee wordt gevat door de volgende oneindige "zin" analoog aan de DCA in $Theo(DB)$:
$$(\forall x)(x = 0 \vee x = S(0) \vee x = S(S(0)) \vee \ldots)$$
Maar helaas is dat geen logische zin. We komen hier verderop terug.

- De definities van $+$ en $\times$. Die kunnen we voorstellen door een oneindige verzameling van gelijkheidsatomen $S^n(0) + S^m(0) = S^{n+m}(0)$ en $S^n(0) \times S^m(0) = S^{n\times m}(0)$. Maar Peano vond een eindige manier om deze uit te drukken:
$$(\forall n)(0 + n = n)$$
$$(\forall n)(\forall m)(S(n) + m = S(n + m))$$
$$(\forall n)(0 \times n = 0)$$
$$(\forall n)(\forall m)(S(n) \times m = m + (n \times m))$$

We zijn klaar, behalve met DCA.

Oefening 4.4.1. *Druk elk van deze axioma's uit in termen van het gegeven logische vocabularium (met $Plus, Succ, \ldots$).*

Er is een groot probleem met DCA: dit axioma kunnen we niet uitdrukken in predikatenlogica[3]. Peano vond echter een manier om DCA uit te drukken in *tweede-orde-logica*, een uitbreiding van predikatenlogica. Zijn axioma noemt men het *inductie-axioma*. Men kan bewijzen dat alle

[3]Dit zal bewezen worden in de cursus Modelleren van Complexe Systemen in de 1ste Master.

modellen van de theorie die we op die manier bekomen (dus, 6 zinnen in predikatenlogica en 1 zin in tweede-orde-logica) isomorf zijn met $\mathfrak{A}$, de natuurlijke getallen.

Maar dit is dus geen theorie van predikatenlogica, omwille van het inductie-axioma. Er bestaat echter een "benadering" van dit axioma in predikatenlogica waarmee men bijna alle belangrijke eigenschappen van de natuurlijke getallen kan bewijzen. Deze "benadering" noemt men het *inductie-schema*. Het is een oneindige verzameling van logische zinnen. Interessant genoeg zijn de formules in dit inductie-schema zeer duidelijk logische voorstellingen van het principe van bewijs door inductie.

Definitie 4.4.1. Het **inductieschema** voor een vocabularium Σ dat minstens $0, S/1$ bevat, bestaat uit alle zinnen van de predikatenlogicataal van Σ van de volgende vorm:

$$(\forall y_1) \ldots (\forall y_n)(\varphi[0] \wedge (\forall x)(\varphi[x] \Rightarrow \varphi[S(x)])) \Rightarrow (\forall x)\varphi[x])$$

waarbij $\varphi[x]$ een willekeurige formule is van predikatenlogica met vrije constante-symbolen x en $y_1, \ldots, y_n$ die niet voorkomen in Σ.

Je merkt duidelijk het verband tussen een zin uit het inductieschema en het bewijsprincipe door inductie. Hierbij speelt $\varphi[n]$ de rol van de inductiehypothese $\mathcal{H}[n]$.

De oneindige theorie, verkregen door in de Peano axioma's het inductie-axioma te vervangen door het inductieschema, wordt **Peano rekenkunde** genoemd. De structuur $\mathfrak{A}$ is een model van deze theorie, maar deze oneindige theorie heeft modellen die niet isomorf zijn met de natuurlijke getallen.

Alle "klassieke" stellingen aangaande de natuurlijke getallen die als logische zinnen van Σ_P geformuleerd kunnen worden, zijn een logisch gevolg van Peano rekenkunde. Daarom dacht men vóór 1931 dat elke zin van Σ_P die waar is in $\mathfrak{A}$, bewijsbaar zou zijn uit deze theorie. Kurt Gödel (1931) bewees echter dat dit niet waar is (zie volgend Hoofdstuk).

Wat Gödel bewees is sterk gerelateerd tot het feit dat we al eerder bespraken in Hoofdstuk 1, namelijk de *onbeslisbaarheid van natuurlijke getallen* (of de stelling van Church-Turing).

De beperking van Peano's rekenkunde tot formules waarin $\times$ niet voorkomt noemt men *Presburger-rekenkunde*. Elke ware zin zonder $\times$ in de natuurlijke getallen is wel te bewijzen uit de Presburger rekenkunde (en dus ook uit Peano's rekenkunde). Dit resultaat is equivalent met de *beslisbaarheid van Presburger-rekenkunde.*

De rol van Peano's rekenkunde in de informatica is niet te onderschatten. In veel complexe informaticaproblemen moet geredeneerd worden over getallen. Bv. compileroptimalisatiesystemen gebruiken Presburger rekenkunde om de volgorde van operaties te optimaliseren, parallelisme mogelijk te maken en geheugengebruik te reduceren. Bv. bewijstechnieken om de correctheid van programma's bewijzen zijn gebaseerd op Peano's inductieschema. De (interactieve) theoremprover Isabelle is een systeem gespecialiseerd voor inductiebewijzen.

Opmerking 4.4.1. Van deze sectie moet je onthouden dat de Peano-axiomas bestaan en waarvoor ze dienen, want dat is ook van belang in het laatste hoofdstuk. Je moet weten hoe een zin van het inductieschema eruit ziet. Je moet de andere axiomas niet van buiten kennen.

4.5 Zoekproblemen en modelgeneratie

Veel informaticaproblemen bestaan uit het zoeken van "waardes" die aan bepaalde "beperkingen" voldoen.

- Een kleuring zoeken van nodes in een grafe zodat twee aanpalende nodes verschillend gekleurd zijn.
- Oplossingen voor logische puzzels, sudoku's, n koninginnen op een $n \times n$ schaakbord plaatsen zodat ze elkaar niet aanvallen.
- Uurroosterproblemen: lessenroosters, examenroosters, ziekenhuisroosters, enz.
- Planningsproblemen: een serie van acties berekenen om een gewenste doelstelling te bereiken.
- Logistieke problemen: DHL zoekt een verzameling van verzend- en verscheepacties om al hun postpakketten ter plaatse te brengen.

Heel vaak zoekt men oplossingen die aan bepaalde optimaliteitscondities voldoen of zo goed mogelijk voldoen.

- We zoeken een plan dat zo snel mogelijk het doel bereikt en met zo weinig mogelijk middelen, bv. grondstoffen, energieverbruik, ...
- DHL zoekt een plan dat zoveel mogelijk postpakketten zo snel mogelijk ter plaatse brengt, met zo laag mogelijke transportkosten.

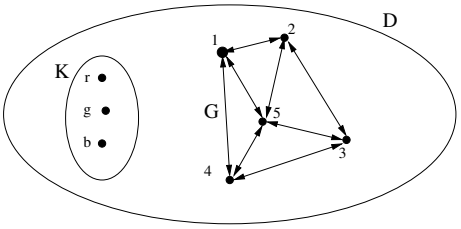
Dergelijke problemen worden ook wel *constraint problemen* genoemd en het wetenschapsdomein dat zich bezig houdt met dergelijke problemen wordt *Constraint Programming* genoemd. Men zoekt men waarde, functies, arrays, matrices, die aan een aantal beperkingen voldoen. Arrays en matrices in Logica? Dat zijn functies! Bv. array a[1..n] of int is een functie $\{1, \ldots, n\} \to \mathbb{N}$. Een booleaanse array b[1..n] of boolean is een predicaat met domein $\{1, \ldots, n\}$. "Beperkingen" zijn eigenschappen en ze moeten in een formele taal uitgedrukt worden. In logische termen, zoekt men dus naar *modellen* van een aantal logische *zinnen*. Vertaald naar het logisch perspectief is het precies dit wat er gebeurt in Constraint Programming maar ook in informaticadisciplines zoals scheduling en planning.

De logische werkwijze voor dergelijke problemen is: stel een vocabularium Σ en een logische theorie T van Σ met de gewenste eigenschappen van de structuur die we zoeken. Zeer vaak is reeds een deel van het model gegeven: bv. het domein en de interpretaties van een aantal predikaat- en functiesymbolen.

Voorbeeld 4.5.1. Het grafekleurprobleem, zie Voorbeeld 2.5.2. We kozen $\Sigma = \{G/2, K/1, C/1 : \}$. We zochten een structuur die aan de volgende formule voldoet:

$$(\forall x)K(C(x)) \wedge (\forall x)(\forall y)(G(x,y) \Rightarrow \neg C(x) = C(y))$$

De grafe en de kleuren zijn gegeven in de vorm van een structuur $\mathfrak{A}_i$ die K en G interpreteert.



Het probleem hier is dus om deze structuur uit te breiden voor het symbool $C/1$ zodat we een model $\mathfrak{A}$ van deze theorie krijgen. De functie $C^{\mathfrak{A}}$ is dan de gewenste oplossing.

Voorbeeld 4.5.2. Het n-koninginneprobleem. Plaats n koninginnen op een nxn bord zodat ze elkaar niet kunnen aanvallen. Het vocabularium bestaat uit een predikaatsymbool $Dim/1$ zodat $Dim(i)$ betekent dat i een rij- of kolom-index is (het bord is vierkant), een functiesymbool $Queen/1$ zodat $Queen(i)$ de kolompositie van de koningin van de i-de rij is, en verder $+$.

Een oplossing van dit probleem voldoet aan volgende axiomas:

- Als r een rij-index van het schaakbord is, dan ook $Queen(r)$.

$$(\forall r)(Dim(r) \Rightarrow Dim(Queen(r))$$

- Er staat hoogstens 1 koningin op een kolom:

$$(\forall r1)(\forall r2)(Dim(r1) \wedge Dim(r2) \wedge Queen(r1) = Queen(r2) \Rightarrow r1 = r2)$$

- Er staat hoogstens 1 koningin op een stijgende diagonaal:

$$(\forall r1)(\forall r2)(Dim(r1) \wedge Dim(r2) \wedge Queen(r1) + r2 = Queen(r2) + r1 \Rightarrow r1 = r2)$$

- Er staat hoogstens 1 koningin op een dalende diagonaal:

$$(\forall r1)(\forall r2)(Dim(r1) \wedge Dim(r2) \wedge Queen(r1) + r1 = Queen(r2) + r2) \Rightarrow r1 = r2)$$

We zoeken een model $\mathfrak{A}$ van deze theorie met domein $\mathbb{N}$ en $Dim^{\mathfrak{A}} = \{1, \ldots, n\}$. Het antwoord wordt gegeven door $Queen^{\mathfrak{A}}$ beperkt tot $\{1, \ldots, n\}$.

Voorbeeld 4.5.3. We bekijken een vereenvoudigd uurroosterprobleem in de context van de studentendatabank. We voegen een nieuwe relatie toe aan de databank: $Timeslot/1$ met alle time slots waarin lessen gegeven mogen worden in de campus. Het doel is om voor elk vak een tijdstip te bepalen zodat er geen conflicten optreden. We zullen deze -ongekende- relatie voorstellen door $Schedule/2$. We veronderstellen dat alle instructors van een vak steeds aanwezig willen zijn in de les. We maken ook gebruik van de type-predikaten $Vak/1, Docent/1, Student/1, Score/1$. We kunnen deze ofwel vervangen door hun definiërende formule, ofwel hun definitie toevoegen aan de theorie.

Aan welke voorwaarden voldoet een goed uurrooster?

- Typering van $Schedule$:

$$(\forall x)(\forall t)[Schedule(x,t) \Rightarrow Vak(x) \wedge Timeslot(t)]$$

- Elke vak vindt plaats op één tijdstip.

$$(\forall x)[Vak(x) \Rightarrow (\exists t)[Timeslot(t) \wedge Schedule(x,t) \wedge (\forall s)(Schedule(x,s) \Rightarrow s = t)]]$$

- Een instructor geeft geen twee vakken op hetzelfde tijdstip.

$$(\forall i)(\forall v)(\forall w)(\forall t)(Instructor(i,v) \wedge Instructor(i,w) \wedge Schedule(v,t) \wedge Schedule(w,t) \Rightarrow v = w)$$

- Een student moet geen twee vakken op hetzelfde tijdstip bijwonen.

$$(\forall i)(\forall v)(\forall w)(\forall t)(Enrolled(s,v) \wedge Enrolled(s,w) \wedge Schedule(v,t) \wedge Schedule(w,t) \Rightarrow v = w)$$

Hoe kunnen we nu een oplossing voor ons uurroosterprobleem berekenen met deze theorie? We hebben een databank $\mathfrak{A}$, een theorie T en we zoeken een structuur $\mathfrak{B}$ die een model is van T en die een uitbreiding is van $\mathfrak{A}$. In zo'n model geeft $Schedule^{\mathfrak{B}}$ de gewenste schedule. Anders gezegd zoeken we $\mathfrak{A}$ uit te breiden met een interpretatie $Schedule^{\mathfrak{A}}$ zodat we een model van T bekomen.

Er bestaan zeer krachtige constraintprogrammatie-systemen met veel industriële toepassingen. De talen in deze systemen worden steeds expressiever en lijken ook steeds meer op logica. Het IDP-systeem in LogicPalet is één van de weinige logische modelgeneratoren voor predikatenlogica en ondersteunt een zeer rijke uitbreiding ervan. Je kunt het systeem downloaden via `http://dtai.cs.kuleuven.be/krr/software/idp`.

Oefening 4.5.1. *De Einsteinpuzzel.*

Let us assume that there are five houses of different colors next to each other on the same road. In each house lives a man of a different nationality. Every man has his favorite drink, his favorite brand of cigarettes, and keeps pets of a particular kind.

- *The Englishman lives in the red house.*
- *The Swede keeps dogs.*
- *The Dane drinks tea.*
- *The green house is just to the left of the white one.*
- *The owner of the green house drinks coffee.*
- *The Pall Mall smoker keeps birds.*
- *The owner of the yellow house smokes Dunhills.*
- *The man in the center house drinks milk.*
- *The Norwegian lives in the first house.*
- *The Blend smoker has a neighbor who keeps cats.*
- *The man who smokes Blue Masters drinks bier.*
- *The man who keeps horses lives next to the Dunhill smoker.*
- *The German smokes Prince.*
- *The Norwegian lives next to the blue house.*
- *The Blend smoker has a neighbor who drinks water.*

The question to be answered is: Who keeps fish?

Kies een vocabularium om de basistypes (huizen, nationaliteiten, kleuren, cigaretten, huisdieren en drank) en hun interrelaties voor te stellen. Druk de eigenschappen uit. Een oplossing

voor het probleem bekom je door een model te berekenen van deze theorie en te kijken wie de fish *bezit.*

Zo'n raadsel wordt in een fractie van een seconde opgelost door het IDP-systeem.

Opmerking 4.5.1. Van deze sectie is er geen theorie te kennen. Er kunnen wel oefeningen gevraagd worden (zie de oefenzittingen hiervoor).

4.6 Formele methoden in Software Engineering

Meestal werken meerdere groepen mensen aan een groot software systeem. Elke groep werkt aan een onderdeel (component, klasse). Duidelijke en precieze communicatie tussen die groepen is essentiëel. Het Nederlands is soms niet precies genoeg. Code is te gedetailleerd. Ondubbelzinnige specificatie van wat elke component doet is noodzakelijk.

De oplossing: gebruik een formele taal voor specificaties. Deze zijn meestal gebaseerd op predikatenlogica.

Dit is op dit ogenblik vooral van belang voor systemen waar bugs zeer gevaarlijk zijn, op fysiek of economisch vlak.

- Controlesystemen voor de luchtvaart.
- Lanceren van ruimtetuigen[4]
- Medische toepassingen[5]
- Besturingssystemen voor kerncentrales.
- Besturingssystemen voor metros.
- Systemen die van primordiaal commercieel belang zijn voor een bedrijf[6]

Kritische systeemeigenschappen moeten met absolute zekerheid gegarandeerd kunnen worden.

Voorbeeld 4.6.1. Twee vliegtuigen mogen nooit opstijgen vanop dezelfde startbaan binnen een interval van 1 minuut.

Er zijn nog andere aspecten. De kosten om een bug op te lossen in software of hardware lopen exponentieel op met hoe ver de ontwikkeling van het systeem reeds gevorderd is. In een specificatie kost het vrijwel niets om een bug op te lossen. In een commercieel systeem kan de kostprijs gigantisch zijn en een firma op of over de rand van het failliet brengen.

Men zoekt dus naar methodes om de correctheid van systemen te bewijzen tijdens de ontwerpfase. Ideaal is om met theoremprovers te bewijzen dat kritische systeemeigenschappen een logisch gevolg zijn van de specificaties van de systeemcomponenten. Vaak is het ook zeer nuttig om een systeem te simuleren, eventueel onder bepaalde beperkingen.

In Belgische bedrijven worden dergelijke technieken nog niet vaak toegepast. Bij grote spelers zoals Microsoft, Siemens, Oracle, ... is de ontwikkeling van verificatietools en formele redeneersystemen in volle gang. Metro 14 (zie Hoofdstuk 1) dateert ondertussen al van 14 jaar geleden en de ontwikkeling is niet stil gestaan!

[4]Meer dan 1 raket is neergestort door bugs in software.
[5]Door bugs in software van bestralingssystemen zijn mensen omgekomen.
[6]Een bug in een Pentium chip kostte Intel ooit miljarden.

Als een toepassing van dit idee zullen we nu een algemene methode zien om transacties in een databank te beschrijven en de correctheid ervan te verfiëren.

Voorbeeld 4.6.2. Als voorbeeld behandelen we een zeer eenvoudig informatiesysteem voor het beheer van een bibliotheek, BibSys genaamd. BibSys bestaat uit:

- Een databank met de interpretatie voor de volgende relaties:
  - $InHoldings/1$: de boeken in bezit van de bib.
  - $OnLoan/2$: boek ... is uitgeleend aan persoon ...
  - $Available/1$: boeken aanwezig in bib.
- Software om bewerkingen (transacties) op de databank uit te voeren.

Er zijn een paar evidente integriteitsbeperkingen voor deze databank:

- $E1$: Een boek is in het bezit van de bib asa het aanwezig is of uitgeleend is. Formeel:

  $$(\forall x)[InHoldings(x) \Leftrightarrow Available(x) \vee (\exists u)OnLoan(x,u)]$$

  Deze eigenschap zal natuurlijk vaak niet voldaan zijn in een echte bib waar regelmatig boeken verloren gaan of gestolen worden. We maken daar abstractie van.

- $E2$: Geen enkel bibliotheekboek is tegelijkertijd uitgeleend en aanwezig. Formeel:

  $$(\forall x)[InHoldings(x) \Rightarrow \neg(\exists u)OnLoan(x,u) \vee \neg Available(x)]$$

- $E3$: Geen enkel bibliotheekboek is uitgeleend aan meer dan één persoon. Formeel:

  $$(\forall x)[InHoldings(x) \Rightarrow \neg(\exists u)(\exists w)(u \neq w \wedge OnLoan(x,u) \wedge OnLoan(x,w))$$

De theorie $\{E1, E2, E3\}$ (of de equivalente formule $E1 \wedge E2 \wedge E3$ zullen we verderop vaak nodig hebben. We duiden ze aan als IB, de verzameling van IntegriteitsBeperkingen van de databank.

In principe zou het bibliotheeksysteem als volgt kunnen werken. De bibliothecaris zou bij aankoop of verlies van een boek, bij elke uitlening of teruggave van een boek de databank kunnen aanpassen. Bv. bij een uitlening van boek b door lezer l voegt hij het koppel (b,l) toe aan $OnLoan^{\mathfrak{A}}$ en schrapt hij b van $Available^{\mathfrak{A}}$. Maar missen is menselijk. Binnen de kortste keren zit de databank dan vol fouten en inconsistenties. Daarom bundelt men de operaties die mogen gebeuren in *transacties*. Elke transactie past de databank op de juiste manier aan, t.t.z., zodat de integriteitsbeperkingen worden bewaard. Als we kunnen bewijzen dat de initiële databank voldoet aan de integriteitsbeperkingen, en dat elke transactie de waarheid van elke integriteitsbeperking bewaart, zullen de integriteitsbeperkingen altijd waar blijven, en moeten ze in principe maar één keer gecontroleerd worden! (Daar ging het over in Hoofdstuk 1.)

BibSys is een voorbeeld van een dynamisch systeem. De meeste informatiesystemen zijn dynamisch: hun toestand verandert voortdurend. Bewerkingen die de toestand van een informatiesysteem veranderen heten *transacties*. Transacties hebben *precondities*: een transactie mag alleen maar worden uitgevoerd indien de toestand van het systeem voldoet aan de precondities. Transacties hebben ook *postcondities*: dat zijn de eigenschappen waaraan de toestand van het systeem voldoet na uitvoering van de transactie. De combinatie van alle pre- en postcondities van een transactie geven een volledige beschrijving van de transactie. We zullen een eenvoudige manier illustreren hoe deze te beschrijven in logica, en hoe deze logische zinnen te gebruiken om te bewijzen dat de transactie de integriteitsbeperkingen bewaart.

De transacties van het bibliotheeksysteem zijn:

- $\texttt{Borrow}(\texttt{u},\texttt{x})$: persoon u leent boek x uit.
- $\texttt{Return}(\texttt{u},\texttt{x})$: persoon u geeft boek x terug aan bib.
- $\texttt{Remove}(\texttt{x})$: boek x wordt uit het bezit van de bib verwijderd.
- $\texttt{Add}(\texttt{x})$: boek x wordt aangekocht door bib.

Merk op: deze transacties zijn operaties op de databank. Het zijn geen predikaatsymbolen!

Laat ons even recapituleren. Het theoretische idee achter dit voorbeeld gaat als volgt. Gegeven is een logisch databankvocabularium Σ. De verschillende toestanden van de databank komen overeen met databankstructuren van Σ (eventueel uitgebreid met nieuwe constanten). We beschikken over een theorie in Σ van integriteitsbeperkingen IB die de zinvolle toestanden van de databank beschrijft. De databank kan enkel aangepast worden door middel van een aantal transacties $\texttt{Trans}(\texttt{x}_1,\ldots,\texttt{x}_\texttt{n})$. Voor deze transacties zouden we het volgende willen beschrijven:

- Voor elke transactie $\texttt{Trans}(\texttt{x}_1,\ldots,\texttt{x}_\texttt{n})$ de **pre-conditie** formule: $Pre_{\texttt{Trans}}[x_1,\ldots,x_n]$ is een formule in Σ met vrije variabelen $x_1,\ldots,x_n$ die beschrijft onder welke voorwaarden de transactie mag plaats vinden.
- We beschrijven voor elke transactie $\texttt{Trans}(\texttt{x}_1,\ldots,\texttt{x}_\texttt{n})$ ook de **post-conditie** formule: $Post_{\texttt{Trans}}[x_1,\ldots,x_n]$ met vrije variabelen $x_1,\ldots,x_n$. Deze beschrijft het effect op de databank.

De preconditie van een transactie beschrijven kan door middel van een formule in Σ. Maar op de postconditie te beschrijven volstaat Σ niet. Immers, een postconditie moet het verband beschrijven tussen de toestand vóór en de toestand ná de transactie. Je moet dus kunnen verwijzen naar twee toestanden van de databank in dezelfde formule!

Om dat te doen introduceren we een nieuw stel symbolen Σ_N: het zijn gewoon copieën van de symbolen van Σ, voorafgegaan door N. Bv. $NInHoldings, NOnLoan, NAvailable$. De "$N$" slaat op "Next". Een postconditie zal dan een formule zijn van $\Sigma \cup \Sigma_N$.

Voorbeeld 4.6.3. Terug naar BibSys. We bespreken pre- en postcondities van de vier transacties.

- $\texttt{Borrow}(\texttt{u},\texttt{x})$: $Pre_{\texttt{Borrow}}[u,x]$ is:
$$Available(x)$$
De postcondities informeel:
  - $InHoldings$ blijft onveranderd. I.e., $InHoldings$ en $NInHoldings$ zijn identiek.
  - Een boek y is uitgeleend aan persoon w na transactie asa ($y = x$ en $w = u$) of y is uitgeleend aan w vóór transactie.
  - Een boek y is in $Available$ na de transactie asa $y \neq x$ en y is available vóór de transactie.

  Postconditie formeel, $Post_{\texttt{Borrow}}[u,x]$:
$$\begin{array}{l}(\forall y)[NInHoldings(y) \Leftrightarrow InHoldings(y)]\wedge\\ (\forall y)(\forall w)[NOnLoan(y,w) \Leftrightarrow y = x \wedge w = u \vee OnLoan(y,w)]\wedge\\ (\forall y)[NAvailable(y) \Leftrightarrow y \neq x \wedge Available(y)]\end{array}$$
- $\texttt{Return}(\texttt{u},\texttt{x})$: Persoon u brengt boek x terug. De preconditie-formule $Pre_{\texttt{Return}}[u,x]$:
$$OnLoan(x,u)$$
De postcondities informeel:

- $Inholdings$ is onveranderd.
- Een boek y is uitgeleend aan persoon w na transactie asa $\neg(y = x$ en $w = u)$ en y is uitgeleend aan w vóór transactie.
- Een boek y is in $Available$ na transactie asa $y = x$ of y is available vóór transactie.

Postcondities formeel: $Post_{\texttt{Return}}[u, x]$:

$$(\forall y)[NInHoldings(y) \Leftrightarrow InHoldings(y)] \wedge$$
$$(\forall y)(\forall w)[NOnLoan(y, w) \Leftrightarrow \neg(y = x \wedge w = u) \wedge OnLoan(y, w)] \wedge$$
$$(\forall y)[NAvailable(y) \Leftrightarrow y = x \vee Available(y, s)]$$

- `Remove(x)`: boek x wordt verwijderd. $Pre_{\texttt{Remove}}[x]$:

$$Available(x)$$

Postcondities informeel:

- Een boek y is $InHoldings$ na de transactie asa $y \neq x$ en y is in $InHoldings$ vóór de transactie.
- $OnLoan$ is onveranderd.
- Een boek y is in $Available$ na de transactie asa $y \neq x$ en y is in $Available$ vóór de transactie.

Postcondities formeel: $Post_{\texttt{Remove}}[x]$:

$$(\forall y)[NInHoldings(y) \Leftrightarrow y \neq x \wedge InHoldings(y)] \wedge$$
$$(\forall y)(\forall w)[NOnLoan(y, w) \Leftrightarrow OnLoan(y, w)] \wedge$$
$$(\forall y)[NAvailable(y) \Leftrightarrow y \neq x \wedge Available(y)]$$

- `Add(x)`: boek x wordt aangeschaft. $Pre_{\texttt{Add}}[u, x]$:

$$\neg InHoldings(x)$$

Postcondities informeel:

- Een boek y is in $InHoldings$ na de transactie asa $y = x$ of y is in $InHoldings$ vóór de transactie.
- $OnLoan$ is onveranderd.
- Een boek y is in $Available$ na de transactie asa $y = x$ of y is in $Available$ vóór de transactie.

Postcondities formeel: $Post_{\texttt{Add}}[u, x]$:

$$(\forall y)[NInHoldings(y) \Leftrightarrow y = x \vee InHoldings(y)] \wedge$$
$$(\forall y)(\forall w)[NOnLoan(y, w) \Leftrightarrow OnLoan(y, w)] \wedge$$
$$(\forall y)[NAvailable(y) \Leftrightarrow y = x \vee Available(y)]$$

We moeten nu bewijzen dat als de transacties conform deze condities geïmplementeerd worden, zij de integriteitsbeperkingen IB zullen bewaren. Anders gezegd, we moeten bewijzen dat de integriteitsbeperkingen *invarianten* zijn van de transacties. Een logische zin is een invariant onder een transactie als geldt dat een toestand waarin de invariant waar is door de transactie wordt omgezet in een toestand waarin de invariant nog steeds geldt.

Zij IB onze theorie van integriteitsbeperkingen in Σ. Definieer IB_N de copie van deze theorie door elk Σ-symbool te vervangen door zijn copie in Σ_N.

Om aan te tonen dat eigenschap IB invariant is onder transactie $\texttt{Trans}(\texttt{x}_1, \ldots, \texttt{x}_n)$ volstaat het te bewijzen dat volgende formule logisch waar is:

$$(\forall x_1) \ldots (\forall x_n)(IB \wedge Pre_{\texttt{Trans}}[x_1, \ldots, x_n] \wedge Post_{\texttt{Trans}}[x_1, \ldots, x_n] \Rightarrow IB_N)$$

Dus, als IB waar is in een toestand, en de preconditie geldt en de postconditie geldt in de volgende toestand dan is IB voldaan in de nieuwe toestand.

Het bewijzen van logische waarheid van een formule is een deductieve redeneertaak. Hiervoor kunnen we Spass gebruiken, of andere theoremprovers.

In wat nu volgt schrijven we alle formules op die bewezen moeten worden door Spass om te bewijzen dat de transacties alle integriteitsbeperkingen bewaren. Je kunt deze formules copiëren van een electronische versie van deze tekst met cut en paste en inladen in LogicPalet.

Voorbeeld 4.6.4. Elk van de vier formules hieronder heeft dezelfde vorm, namelijk

$$(\forall x_1) \ldots (\forall x_n)(IB \wedge Pre_{\texttt{Trans}}[x_1, \ldots, x_n] \wedge Post_{\texttt{Trans}}[x_1, \ldots, x_n] \Rightarrow IB_N)$$

Ze verschillen dus enkel in de middelste gedeeltes.

Invariantie onder $\texttt{Borrow}[\texttt{u}, \texttt{x}]$**.** Te bewijzen: volgende formule is logisch waar:

$$
\begin{array}{l}
(\forall u)(\forall x)[\\
(\forall x)[InHoldings(x) \Leftrightarrow Available(x) \vee (\exists u)OnLoan(x,u)] \wedge \\
(\forall x)[InHoldings(x) \Rightarrow \neg(\exists u)OnLoan(x,u) \vee \neg Available(x)] \wedge \\
(\forall x)[InHoldings(x) \Rightarrow \neg(\exists u)(\exists w)(u \neq w \wedge OnLoan(x,u) \wedge OnLoan(x,w))] \\
\wedge \\
Available(x) \\
\wedge \\
(\forall y)[NInHoldings(y) \Leftrightarrow InHoldings(y)] \wedge \\
(\forall y)(\forall w)[NOnLoan(y,w) \Leftrightarrow y = x \wedge w = u \vee OnLoan(y,w)] \wedge \\
(\forall y)[NAvailable(y) \Leftrightarrow y \neq x \wedge Available(y)] \\
\Rightarrow \\
(\forall x)[NInHoldings(x) \Leftrightarrow NAvailable(x) \vee (\exists u)NOnLoan(x,u)] \wedge \\
(\forall x)[NInHoldings(x) \Rightarrow \neg(\exists u)NOnLoan(x,u) \vee \neg NAvailable(x)] \wedge \\
(\forall x)[NInHoldings(x) \Rightarrow \neg(\exists u)(\exists w)(u \neq w \wedge NOnLoan(x,u) \wedge NOnLoan(x,w))]]
\end{array}
$$

Invariantie onder $\texttt{Return}(\texttt{u}, \texttt{x})$**.** Te bewijzen: volgende formule is logisch waar:

$$
\begin{array}{l}
(\forall u)(\forall x)[\\
(\forall x)[InHoldings(x) \Leftrightarrow Available(x) \vee (\exists u)OnLoan(x,u)] \wedge \\
(\forall x)[InHoldings(x) \Rightarrow \neg(\exists u)OnLoan(x,u) \vee \neg Available(x)] \wedge \\
(\forall x)[InHoldings(x) \Rightarrow \neg(\exists u)(\exists w)(u \neq w \wedge OnLoan(x,u) \wedge OnLoan(x,w))] \\
\wedge \\
OnLoan(x,u) \\
\wedge \\
(\forall y)[NInHoldings(y) \Leftrightarrow InHoldings(y)] \wedge \\
(\forall y)(\forall w)[NOnLoan(y,w) \Leftrightarrow \neg(y = x \wedge w = u) \wedge OnLoan(y,w)] \wedge \\
(\forall y)[NAvailable(y) \Leftrightarrow y = x \vee Available(y)] \\
\Rightarrow \\
(\forall x)[NInHoldings(x) \Leftrightarrow NAvailable(x) \vee (\exists u)NOnLoan(x,u)] \wedge \\
(\forall x)[NInHoldings(x) \Rightarrow \neg(\exists u)NOnLoan(x,u) \vee \neg NAvailable(x)] \wedge \\
(\forall x)[NInHoldings(x) \Rightarrow \neg(\exists u)(\exists w)(u \neq w \wedge NOnLoan(x,u) \wedge NOnLoan(x,w))]]
\end{array}
$$

Invariantie onder `Remove(x)`. Te bewijzen: volgende formule is logisch waar:

$$
\begin{array}{l}
(\forall x)[\\
(\forall x)[InHoldings(x) \Leftrightarrow Available(x) \vee (\exists u)OnLoan(x,u)] \wedge \\
(\forall x)[InHoldings(x) \Rightarrow \neg(\exists u)OnLoan(x,u) \vee \neg Available(x)] \wedge \\
(\forall x)[InHoldings(x) \Rightarrow \neg(\exists u)(\exists w)(u \neq w \wedge OnLoan(x,u) \wedge OnLoan(x,w))] \\
\wedge \\
Available(x) \\
\wedge \\
(\forall y)[NInHoldings(y) \Leftrightarrow y \neq x \wedge InHoldings(y)] \wedge \\
(\forall y)(\forall w)[NOnLoan(y,w) \Leftrightarrow OnLoan(y,w)] \wedge \\
(\forall y)[NAvailable(y) \Leftrightarrow y \neq x \wedge Available(y)] \\
\Rightarrow \\
(\forall x)[NInHoldings(x) \Leftrightarrow NAvailable(x) \vee (\exists u)NOnLoan(x,u)] \wedge \\
(\forall x)[NInHoldings(x) \Rightarrow \neg(\exists u)NOnLoan(x,u) \vee \neg NAvailable(x)] \wedge \\
(\forall x)[NInHoldings(x) \Rightarrow \neg(\exists u)(\exists w)(u \neq w \wedge NOnLoan(x,u) \wedge NOnLoan(x,w))]]
\end{array}
$$

Invariantie onder `Add[x]`. Te bewijzen: volgende formule is logisch waar:

$$
\begin{array}{l}
(\forall x)[\\
(\forall x)[InHoldings(x) \Leftrightarrow Available(x) \vee (\exists u)OnLoan(x,u)] \wedge \\
(\forall x)[InHoldings(x) \Rightarrow \neg(\exists u)OnLoan(x,u) \vee \neg Available(x)] \wedge \\
(\forall x)[InHoldings(x) \Rightarrow \neg(\exists u)(\exists w)(u \neq w \wedge OnLoan(x,u) \wedge OnLoan(x,w))] \\
\wedge \\
\neg InHoldings(x) \\
\wedge \\
(\forall y)[NInHoldings(y) \Leftrightarrow y = x \vee InHoldings(y)] \wedge \\
(\forall y)(\forall w)[NOnLoan(y,w) \Leftrightarrow OnLoan(y,w)] \wedge \\
(\forall y)[NAvailable(y) \Leftrightarrow y = x \vee Available(y)] \\
\Rightarrow \\
(\forall x)[NInHoldings(x) \Leftrightarrow NAvailable(x) \vee (\exists u)NOnLoan(x,u)] \wedge \\
(\forall x)[NInHoldings(x) \Rightarrow \neg(\exists u)NOnLoan(x,u) \vee \neg NAvailable(x)] \wedge \\
(\forall x)[NInHoldings(x) \Rightarrow \neg(\exists u)(\exists w)(u \neq w \wedge NOnLoan(x,u) \wedge NOnLoan(x,w))]]
\end{array}
$$

SPASS bewijst elk van deze stellingen in minder dan 0,1 seconde!!

So what? Het kan misschien lijken alsof hier veel werk gedaan werd om iets evidents te bewijzen. Maar dat is een verkeerde indruk.

Voor dit kleine voorbeeldje kan het lijken alsof de integriteitsbeperkingen/invarianten evident zijn. Maar als je ze zelf zou moeten opstellen dan zou je bijna zeker fouten of onvolledigheden erin schrijven. Iedereen maakt daar fouten tegen of vergeet iets.

Dit voorbeeld is afkomstig uit een cursus die gegeven wordt aan de universiteit van Toronto. Toen Prof. Jan Denef dit voorbeeld uitwerkte met Spass, vond hij hiaten in de integriteitsbeperkingen, waardoor de invariantie ervan niet gegarandeerd kon worden. Voor zo'n eenvoudig databankje!

Het is des te moeilijker in echte toepassingen. Een fout in de specificatie leidt dan al gemakkelijk tot een fout in de implementatie en tot latere bugs. Zo'n fouten kunnen een hele tijd latent blijven, en plots een catastrofe veroorzaken.

Programming by contract. Programmeurs schrijven code voor elke transactie. Zij moeten de pre- en postcondities van een transactie beschouwen als een contract dat geïmplementeerd moet worden.

Correctheid van de code voor een transactie betekent: wanneer de code uitgevoerd wordt in toestand die aan de precondities voldoet dan stopt het programma na een eindig stappen in een toestand die voldoet aan de postcondities. M.a.w, het contract is correct geïmplementeerd.

Zoals we bespraken in Hoofdstuk 1 bestaan er twee methodes om de correctheid van code te garanderen:

- programmaverificatie: automatische correctheidsbewijzen van de code.
- programmageneratie: het genereren van code uit de specificatie van pre- en postcondities.

We gaan hier niet verder op in.

Zelfs als het contract van transacties correct werd gevolgd, de invarianten van de integriteitsbeperkingen werden bewezen en de initiële databank voldeed aan de integriteitsbeperkingen, dan biedt dit toch geen 100% garantie dat er geen fouten kunnen binnensluipen. Bv. een computer kan crashen midden in een transactie, met de transactie slechts half uitgevoerd. Databanksystemen voorzien oplossingen hiervoor.

Oefening 4.6.1. *Beschouw volgende transacties op StudentenDatabank:*
- *$AddGrade(student, vak, score)$: toevoegen aan tabel Grade.*
- *$EnrollStudent(student, vak)$: toevoegen aan tabel Enrolled.*

Formuleer de voor de hand liggende pre- en postcondities voor deze transacties, zodanig dat het volgende gegarandeerd is:
- *Een student die een score voor een vak verkrijgt moet daarvoor ingeschreven zijn.*
- *Een student die een score voor een vak verkrijgt moet automatisch voor dat vak uit de tabel Enrolled verdwijnen. Hij mag zich eventueel later terug inschrijven.*
- *Iedereen die ingeschreven is voor een vak moet geslaagd zijn voor alle vakken die nodige voorkennis zijn voor dat vak.*
- *Iemand die ingeschreven is of reeds geslaagd is voor een vak mag zich niet inschrijven voor dat vak.*

Alternatieve Redeneerproblemen Tot hiertoe hebben we naar 1 specifieke vorm van redeneren gekeken bij transacties: het bewijzen dat ze de invarianten bewaren (i.e., de integriteitsbeperkingen). Er zijn nog een paar alternatieve redeneerproblemen.

Zonder twijfel de meest evidente taak is: kunnen we een transactie doorvoeren zonder een programma ervoor te schrijven (zoals nu altijd gebeurt), t.t.z. puur op basis van redeneren over de gegeven logische pre- en postcondities van de transactie?

Gegeven een theorie $Trans$ met de logische pre- en postcondities voor alle transacties,
gegeven een databank $\mathfrak{A}$, en gegeven een gewenste transactie Tr,
bereken het resultaat $\mathfrak{A}'$ bekomen door Tr uit te voeren op $\mathfrak{A}$.

Dit is inderdaad mogelijk, namelijk met behulp van modelgeneratie. Dit zal gedemonstreerd worden in de oefenzittingen. Dit is zonder meer het meest dagdagelijkse redeneerprobleem: het toepassen van een concrete transactie op een databank. Het is belangrijk om te beseffen dat deze taak kan opgelost door middel van de logische specificatie van transacties en de gepaste vorm van redeneren. Het is evenwel niet deductie die we hier nodig hebben. Spass is onbruikbaar om concrete transacties te realiseren.

Een andere redeneertaak is als volgt. Meestal is het wenselijk dat een transactie deterministisch is, t.t.z., dat het resultaat ervan (m.a.w. de resulterende databank) eenduidig is bepaald. M.a.w., de transactie kan maar op 1 manier worden uitgevoerd. Dus is een zinvolle verificatietaak de volgende:

Gegeven een theorie $Trans$ met de logische pre- en postcondities voor alle transacties, bewijs dat de uitkomst van het toepassen van een willekeurige transactie deterministisch is.

Opmerking 4.6.1. Deze sectie is een gevorderde sectie voor de toepassing van logica in de informatica. Het materiaal ervan wordt grotendeels hernomen in de cursus Modelleren van Complexe Systemen van 1Ma. Wat je hiervan moet kennen voor het examen is: het algemene principe hoe een specificatie op te stellen van transacties, en hoe de correctheid van transacties te bewijzen.

4.7 Herbruiken van dezelfde modellering voor verschillende systemen.

In deze laatste sectie van dit hoofdstuk zullen we een principe demonstreren die logica en formele modelleertalen onderscheidt van programmeertalen:

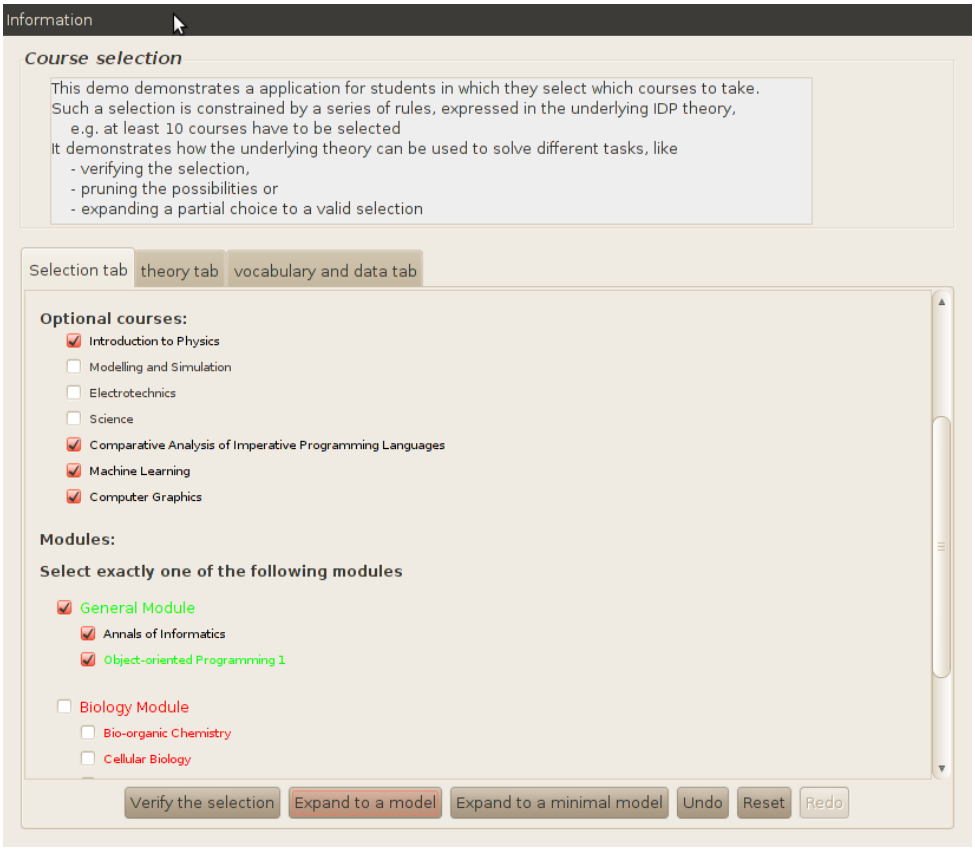
Gebruik van dezelfde theorie om diverse types van problemen binnen hetzelfde toepassingsdomein op te lossen door middel van verschillende vormen van redeneren.

In Figuur 4.7 wordt de interface van een prototype-systeem getoond dat illustreert hoe in een dagdagelijkse toepassing diverse types van redeneren van nut kunnen zijn als we tenminste de achtergrondinformatie expliciet modelleren in een theorie.

Deze toepassing werd ontwikkeld met het IDP-systeem. Het is een interactief configuratietool om een student van een fictieve studierichting toe te laten een studieprogramma samen te stellen.

De kennis van het toepassingebied werd gemodelleerd door een logische theorie T die de regels over toegelaten studiekeuzes uitdrukt. Deze theorie bevat regels zoals:

- Verplichte vakken moeten gekozen worden.
- Je moet exact één module kiezen.
- Je moet minstens 2 en hoogstens 4 optionele vakken kiezen.
- enz.



Figuur 4.3: Een interactief configuratietool

In het systeem worden vijf vormen van inferentie gebruikt om de student in verschillende fases van zijn keuze te ondersteunen:

- *waarheid berekenen*: bereken of de huidige selectie een model is van T
- *theoremproving*: bereken verplichte en verboden keuzes gegeven de huidige selectie t.o.v. T
- *modelgeneratie*: vervolledig huidige keuzes tot een correct studieprogramma (een model van T).
- *optimalisatie*: vervolledig huidige keuzes tot een optimaal studieprogramma (met een minimaal aantal studiepunten)
- *explanation*: leg uit waarom een keuze tot inconsistentie leidt.

Het systeem is geïmplementeerd als interface die informatie communiceert tussen de GUI en het IDP-systeem. Het aanpassen van de theorie resulteert onmiddelijk in aangepast gedrag van het systeem.

Toepassingen van deze soort hebben de neiging om snel te evolueren. Bv. elk jaar zijn de regels anders. Je kunt dit in Java implementeren, maar de inspanning zal misschien 10 keer groter zijn en meer, en elk jaar bestaat het risico dat de code op niet triviale manier aangepast moet worden.

Een voordeel van dit tool is dat dezelfde logische regels hergebruikt worden in elk van de 5 vormen van inferentie. Dezelfde logische zinnen worden gebruikt wanneer de student verifieert of de theorie voldaan is in zijn huidige keuze (Verify the selection), wanneer het systeem zijn huidige selectie probeert te vervolledigen tot een model van de theorie (Expand to a model) of tot een minimaal model, wanneer IDP berekent welke de gevolgen zijn van de keuzes, en wanneer het een uitleg genereert waarom een keuze niet gemaakt kan worden.

In een java-programma zou elke logische regel ontdubbeld moeten worden tot 5 apparte stukken code in de 5 procedures die telkens één van deze functionaliteiten implementeren.

Opmerking 4.7.1. Van deze sectie moet je maar één ding onthouden: dat logische modelleringen herbruikt kunnen worden om verschillende functionaliteiten te voorzien in een software systeem, waarbij verschillende vormen van redeneren worden gebruikt. Dat is het principe dat logische modelleertalen onderscheidt van programmeertalen.

4.8 Conclusie

In elk toepassingsdomein van de Informatica is er informatie. Overal kan men de volgende vragen stellen:

- Welke soort of soorten informatie is beschikbaar?
- Welke formele talen of taalconstructies hebben we nodig om deze uit te drukken?
- Welke soorten van redeneren hebben we nodig om concrete informaticaproblemen op te lossen met de gegeven informatie.

Als we een concreet informaticaprobleem kunnen herleiden tot een inferentie-probleem en we beschikken over een efficiënte generische solver voor dit type van inferentieprobleem, dan hoeven we geen programma meer te schrijven! Dat is de “dream, the quest, the holy grail” waarover Bill Gates het had in het citaat pagina 7 van deze cursus. Dit zijn daarom vragen van fundamenteel wetenschappelijk belang.

Het inferentie-probleem waar het gebruik van logica en generische solvers zeer sterk is doorgedrongen in de praktijk is het query-probleem. SQL (of toch een belangrijk fragment ervan) kan beschouwd worden als een expressieve logische taal; de kost om een programma te schrijven om een query-probleem op te lossen is een veelvoud van het formuleren ervan in SQL; en bovendien zal een databasesysteem de query meestal efficiënter kunnen oplossen dan een zelfgeschreven programma. Maar het query-probleem is een eenvoudig type van inferentie-probleem. Voor andere types van inferentie-problemen staan we nog lang zover niet.

In het algemeen is er nog verbazend weinig dat vast staat over deze vragen. Er bestaan veel fragmentarische resultaten, maar er is geen brede wetenschappelijke theorie die deze vragen beantwoordt voor een substantieel gedeelte van de informaticapraktijk. (Welk soort inferentieprobleem lossen programmeurs in bedrijven op? Welke achtergrondinformatie gebruiken ze daarbij? Wat is een natuurlijke weg om die problemen via logica en logische inferentie op te lossen? Zijn er beperkingen in deze logische weg? Vaak hebben we geen idee) Bovendien is nog veel ontwikkeling nodig om meer efficiënte solvers voor complexe inferentieproblemen te bouwen.

Om de droom te realiseren is dus nog veel weg af te leggen. In dit opzicht staat computerwetenschappen nog in zijn kinderschoenen. Maar in de wetenschappelijke wereld is de interesse en het onderzoek naar formele modelleertalen exponentieel aan het stijgen. Grote bedrijven zoals Siemens en Microsoft investeren in onderzoek rond formele talen en technieken en gebruiken nu reeds dergelijke methodes voor bepaalde types van problemen. Het is een kwestie van tijd voor hier nieuwe softwaretools uit ontstaan.

Hiermee sluiten we de toepassingen van logica en redeneren voor de informatica af. Het laatste hoofdstuk zal gaan over logica en de grondslagen van de wiskunde.

Hoofdstuk 5

Het Halting problem, Onbeslisbaarheid en de Onvolledigheidsstelling van Gödel

Wat is een berekening? Een eindige opeenvolging van elementaire berekeningsstappen, bv. in de natuurlijke getallen $+, \times, -, \div, \%, \ldots$. Als we if-then-else testen invoeren, kunnen we *bomen* van berekeningen introduceren door zo'n testen te nesten. Daarbij is elke tak een opeenvolging van testen en berekeningsstappen. Zo'n boom kan oneindig groot zijn en oneindige takken bevatten of oneindig veel takken bevatten. Wat is een *algoritme*? Een eindige beschrijving van een potentieel onbegrensde berekeningsboom. Bv. een algoritme dat bij invoer een getal n krijgt en berekent of n een priemgetal is, daarvan zal de berekeningsboom onbegrensd groot zijn aangezien voor steeds grotere n steeds meer stappen gezet moeten worden om te bepalen of n een priemgetal is. Hoe kunnen we onbegrensd grote berekeningsbomen op eindige manier voorstellen? Door middel van de while-lus, of for-lus of de GOTO. Hierdoor kunnen stukken berekening herhaald worden (op andere waardes).

Wat kan allemaal berekend worden? Bv. we kunnen som van twee eindige natuurlijke getallen of floating point getallen, of matrices berekenen. We kunnen in principe berekenen of een getal n al dan niet priem is. Kunnen we ook oneindige objecten berekenen. Bv. kunnen we de verzameling van alle priemgetallen berekenen? Ja, als we het programma oneindig lang mogen laten lopen.

Computergeheugens bestaan uit bits 0 of 1. Alle soorten objecten in een computergeheugen (strings, floating points, bools, matrices, arrays) bestaan uit bitvectoren. Bv. floating point getallen $0,26542 * 10^{23}$ worden intern voorgesteld door een combinatie van een eindige matisse 2654 en een exponent 23. Bitvectoren komen overeen met getallen of verzamelingen van natuurlijke getallen van begrensde grootte. Maw, alles wat te berekenen valt, valt te herleiden tot berekeningen van natuurlijke getallen. Dus is het logisch om de vraag "wat kan berekend worden?" te bestuderen in de context van de natuurlijke getallen.

Hierboven werden een aantal vragen gesteld over de fundamenten van de informatica (wat is een algoritme? wat kunnen we berekenen?). We zullen hier een paar van deze vragen beantwoorden.

5.1 Berekenbaarheid, Registermachines en de hypothese van Church

De eerste stap is een formele definitie van wat een "algoritme" nu precies is.

Definitie 5.1.1. Een registermachine bestaat uit

- Een eindig aantal *registers* $R_0, R_1, R_2, R_3, \cdots$ die elk een willekeurig groot natuurlijk getal kan bevatten. R_0 speelt een speciale rol en wordt de *accumulator* genoemd.
- Een *processor* die operaties kan uitvoeren op de getallen in de registers, en dit volgens een welbepaald *programma* dat ingebouwd is in de machine.
- Het programma van de machine is een genummerde eindige rij van *instructies*. Een instructie is een van de volgende bevelen:

Symbool	Betekenis
B R A n	Breng getal in Register R_n naar Accumulator. $R_0 := R_n$; ga naar de volgende instructie.
B A R n	Breng getal in Accumulator naar Register R_n. $R_n := R_0$; ga naar de volgende instructie.
C O N n	Breng het getal n (een concreet vast Constant getal) in de accumulator. $R_0 := n$; ga naar de volgende instructie.
O P T n	$R_0 := R_0 + R_n$; ga naar de volgende instructie.
A F T n	$R_0 := R_0 \dot{-} R_n$; ga naar de volgende instructie. – $R_0 \dot{-} R_n = 0$ indien $R_0 - R_n < 0$ – $R_0 \dot{-} R_n = R_0 - R_n$ indien $R_0 - R_n \geq 0$
V E R n	$R_0 := R_0 \times R_n$; ga naar de volgende instructie.
D E L n	$R_0 := R_0 \div R_n$; ga naar de volgende instructie. – $R_0 \div R_n = 0$ indien $R_n = 0$. – $R_0 \div R_n$ is gehele deling indien $R_n \neq 0$.
S P R n	(Sprong) Ga naar de instructie met nummer n.
V S P n	(Voorwaardelijke Sprong) If R0=0 then GOTO n else ga naar volgende instructie.
S T O P	Stop.

De n na een bevel is een concreet natuurlijk getal. Voor sommige bevelen moet n verwijzen naar een bestaand register; voor andere naar een bestaande instructie. Bijvoorbeeld **S P R** n met n groter dan het aantal instructies is niet toegelaten.

We spreken af dat de registers van een registermachine exact de registers zijn die gebruikt worden in zijn programma. Ook spreken we af dat **S T O P** de de laatste instructie isvan een programma en nergens anders voorkomt. (Vanop andere plaatsen waar je wilt stoppen spring je gewoon naar de laatste instructie).

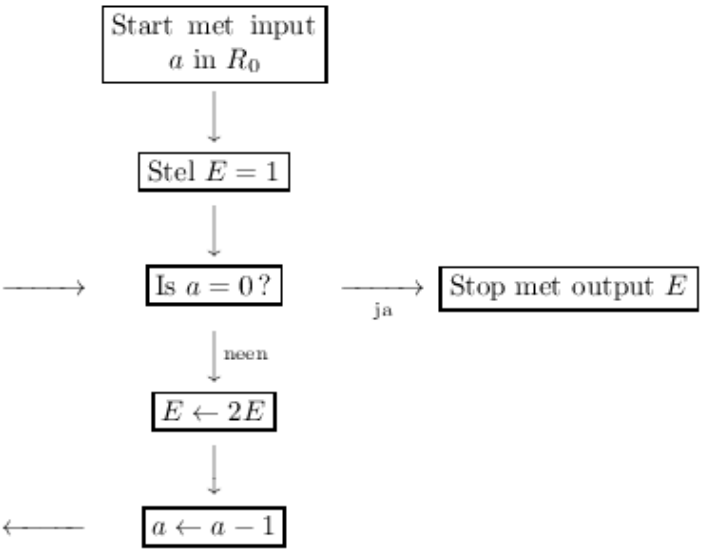
Opmerking 5.1.1. Een registermachine is een "denkbeeldige" computer. In een reële computer kunnen er in de registers slechts getallen met een beperkt aantal cijfers staan. (Anderzijds kan de hardware van een reële computer altijd aangepast worden met registers die grotere getallen kunnen bevatten naargelang de noden.)

Opmerking 5.1.2. Een registermachine werkt volgens één welbepaald programma. Met een ander programma komt een andere registermachine overeen. We beschouwen dus oneindig veel registermachines naargelang het programma. In feite komt een registermachine meer overeen met een programma in machinekode dan met een computer. (Maar ook in de vroegste computers was het programma vastgecodeerd.) Dit is maar een kwestie van terminologie.

Definitie 5.1.2. We zeggen dat een registermachine M bij *input* $a_0, a_1, a_2, \cdots, a_k$ stopt na een eindig aantal stappen met *output* b (in de accumulator) als, wanneer men de machine start met in registers $R_0, R_1, \cdots, R_k$ de natuurlijke getallen $a_0, a_1, \cdots, a_k$ *en in alle andere registers nul*, de machine stopt na een eindig aantal stappen met uiteindelijk b in register R_0 (en niet nader bepaalde getallen in de andere registers).

Voorbeeld 5.1.1. Geef een programma voor een registermachine die bij input $a \in \mathbb{N}$ stopt na een eindig aantal stappen met output 2^a.

Oplossing. Vooraleer we het programma schrijven, stellen we eerst een flowchart op:



Om dit te encoderen kiezen we registers die overeenkomen met de "variabelen" van deze flowchart. We plaatsen a in register R_1, E in register R_2, en 1 in register R_3.

Het programma is als volgt:

1 **B A R** 1
2 **C O N** 1
3 **B A R** 2
4 **B A R** 3
5 **B R A** 1
6 **V S P** 14
7 **C O N** 2
8 **V E R** 2
9 **B A R** 2
10 **B R A** 1
11 **A F T** 3
12 **B A R** 1
13 **S P R** 6
14 **B R A** 2
15 **S T O P**

Definitie 5.1.3. Een afbeelding

$$f: \mathbb{N}^k = \underbrace{\mathbb{N} \times \mathbb{N} \times \cdots \times \mathbb{N}}_{k \text{ keren}} \to \mathbb{N}$$

is berekenbaar door middel van een registermachine als er een registermachine M bestaat die bij iedere input $a_1, a_2, \cdots, a_k \in \mathbb{N}$ stopt na een eindig aantal stappen met output $f(a_1, \cdots, a_k)$.

Definitie 5.1.4. Een relatie R op $\mathbb{N}$ met k argumenten is *berekenbaar door middel van een registermachine* als de afbeelding

$$\begin{aligned} f_R : \mathbb{N}^k \to \mathbb{N} : (a_1, \cdots, a_k) &\mapsto 1 && \text{als } (a_1, \cdots, a_k) \in R \\ (a_1, \cdots, a_k) &\mapsto 0 && \text{als } (a_1, \cdots, a_k) \notin R \end{aligned}$$

berekenbaar is door middel van een registermachine.

Dat de relatie R berekenbaar is betekent dus dat er een registermachine bestaat die bij input $a_1, \cdots, a_k$ stopt na een eindig aantal stappen met output 1 als $(a_1, \cdots, a_k) \in R$ en output 0 in het andere geval.

Daar een deelverzameling van $\mathbb{N}$ een 1-voudige relatie is, kunnen we dus ook spreken over deelverzamelingen van $\mathbb{N}$ die berekenbaar zijn door middel van een registermachine.

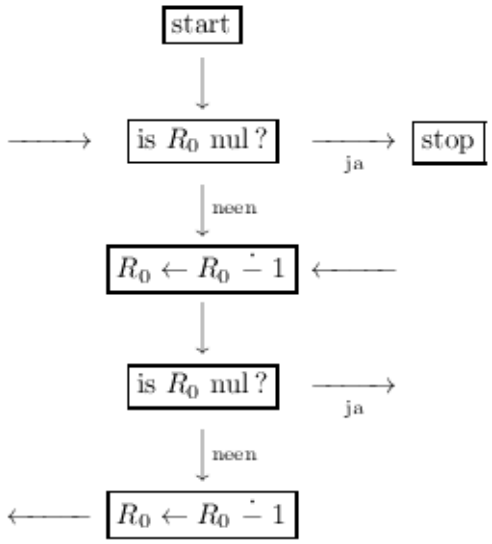
Wat hebben we tot nog toe gedaan? We hebben een soort van machinetaal voorgesteld waarin programma's kunnen beschreven worden om bepaalde relaties te berekenen. Maar er zijn zoveel machinetalen. Kunnen daarmee andere relaties berekend worden? Bv. kun je nieuwe nuttige en practisch realiseerbare operaties bedenken die als je ze zou toevoegen aan de taal van registermachines zouden toelaten om méér relaties te berekenen? Men gaat ervan uit dat dit onmogelijk is.

Hypothese 5.1.1 (van Church)**.** *Elke afbeelding $f : \mathbb{N}^k \to \mathbb{N}$ die berekenbaar is door middel van een "algoritme" ($\stackrel{def}{=}$ een rekenwijze die op een computer kan geprogrammeerd worden) is berekenbaar door middel van een registermachine.*

De bovenstaande hypothese kan niet wiskundig bewezen worden. Omdat de definitie van "algoritme" zeer vaag is. Als men met "algoritme" een programma van een registermachine bedoelt, dan valt er niets te bewijzen. Er zijn een hele reeks definities bedacht voor "algoritme". Voor elke redelijke definitie van "algoritme" die men ooit heeft voorgesteld, heeft men de hypothese van Church bewezen. Deze hypothese wordt daarom algemeen aanvaard. Voor een uitvoerige bespreking verwijzen we naar hoofdstuk V in het boek van Minsky : Computation, finite and infinite machines, Prentice-Hall (1972).

5.2 Onbeslisbaarheid van het stop-probleem

Voorbeeld 5.2.1. Beschouw de registermachine met het volgende programma (met registers R_0, R_1, R_2):



1	**B A R** 2	$\%R_2 := R_0$
2	**C O N** 1	
3	**B A R** 1	$\%R_1 := 1$
4	**B R A** 2	$\%R_0 := R_2$
5	**V S P** 10	$\%R_0 = 0?GOTO\ 10$
6	**A F T** 1	$\%R_0 := R_0 \dot{-} 1$
7	**V S P** 6	$\%R_0 = 0?GOTO\ 6$
8	**A F T** 1	$\%R_0 := R_0 \dot{-} 1$
9	**S P R** 5	
10	**S T O P**	

Met $R_0 \dot{-} 1$ bedoelen we:

- $R_0 - 1$ als $R_0 - 1 \geq 0$
- 0 als $R_0 < 0$.

Wat doet dit programma? Als $R_0 > 1$ dan zal de buitenste lus 2 aftrekken van R_0. Als $R_0 = 1$ dan gaat het programma in de binnenste oneindige lus. En als $R_0 = 0$, dan stopt het programma. Dit programma blijft dus 2 aftrekken tot het ofwel 1 berekend en in een oneindige lus gaat, ofwel 0 berekend en stopt. Het stopt als de beginwaarde in R_0 even is, en gaat in een oneindige lus als R_0 oneven is.

Definitie 5.2.1. Het stop-probleem is het probleem om te bepalen of een gegeven registermachine bij een gegeven input al dan niet na een eindig aantal stappen zal stoppen.

De vraag die we ons hier stellen is: is het mogelijk om een algoritme te schrijven dat het stop-probleem oplost? Zo'n algoritme zou als invoer een voorstelling van een registermiachine M en een invoer m krijgen, en zou dan na een eindig aantal rekenstappen stoppen met het antwoord 1 indien M stopt met invoer m, en anders met 0. Door de hypothese van Church zou er dan ook een registermachine M_{stop} moeten bestaan die dit berekent! Er moet daarvoor wel eerst een practisch probleem opgelost worden: een registermachine neemt enkel getallen als invoer, geen andere registermachines. Dus ontwikkelen we eerst een methode om willekeurige registermachines M te "coderen" als een natuurlijk getal.

We gaan aan elke registermachine M een natuurlijk getal associëren, het *codenummer M* genoemd. Eerst associëren we aan elke instructie een codenummer, volgens de onderstaande tabel:

Instructie	Codenummer
B R A n	$1+10n$
B A R n	$2+10n$
C O N n	$3+10n$
O P T n	$4+10n$
A F T n	$5+10n$
V E R n	$6+10n$
D E L n	$7+10n$
S P R n	$8+10n$
V S P n	$9+10n$
S T O P	10

Uit elk getal kunnen we berekenen of het met een instructie overeenkomt, en zo ja, welke instructie. Bv. 20 komt niet overeen met een instructie. Het getal 34 komt overeen met **O P T** 3.

Zij nu M een registermachine met programma P. Het *codenummer* van M is per definitie

$$\prod_i p_i^{(\text{codenummer van de } i^{\text{de}} \text{ instructie van } P)}$$

waarbij p_i het i^{de} priemgetal is, (bijvoorbeeld $p_1 = 2, p_2 = 3, p_3 = 5, p_4 = 7, p_5 = 11, \dots$).

Hier stelt $\prod_i$ het product voor over alle $i = 1, 2, \cdots$, (aantal instructies in P). Omdat elk getal van $\mathbb{N}_0$ slechts op één wijze in priemfactoren kan ontbonden worden, kan men uit het codenummer van M het programma P van M volledig reconstrueren. Anderzijds is niet elk getal in $\mathbb{N}_0$ een codenummer van een registermachine.

Notatie 5.2.1. De registermachine met codenummer n stellen we voor door M_n.

Indien $n \in \mathbb{N}$ geen codenummer van een registermachine is, dan spreken we af dat M_n de machine is met programma enkel de instructie **S T O P**. Merk op dat de code van het programma met enkel de instructie **S T O P** gelijk is aan 2^{10}. Het programma van de machine $M_{2^{10}}$ is dus

1 **S T O P**.

Dus, als n geen codenummer van een registermachine is, geldt $M_n = M_{2^{10}}$.

Nu keren we terug naar onze vraag: kan er een registermachine M_{stop} bestaan om het stop-probleem op te lossen? Deze machine zou twee inputs hebben: een code n (van de registermachine M_n) en een getal m. M_{stop} zou na eindig aantal stappen stoppen met 1 in de accumulator R_0 als M_n stopt met invoer m, en met 0 in R_0 als M_n niet stopt bij invoer m.

We zullen bewijzen dat zo'n registermachine niet kan bestaan, en dus, door de Hypothese van Church, dat er geen algoritme kan bestaan om het stop-probleem op te lossen.

Als M_{stop} zou bestaan, dan bestaat er natuurlijk ook een vereenvoudigde registermachine die input n heeft, en als output 1 als M_n stopt voor input n en 0 als M_n niet stopt voor input

n. (Namelijk: kopieer R_0 met n naar R_1 en voer vervolgens M_{stop} uit.) Deze vereenvoudigde registermachine berekent 0 of 1 voor invoer n en berekent dus volgende verzameling:

$$\{n \in \mathbb{N}_0 | M_n \text{ bij input } n \text{ stopt na een eindig aantal stappen"}\}$$

De volgende stelling zegt echter dat deze verzameling *niet* berekenbaar is!

Stelling 5.2.1. *(Onbeslisbaarheid van het stop-probleem) De verzameling*

$$\{n \in \mathbb{N}_0 | M_n \textit{ bij input } n \textit{ stopt na een eindig aantal stappen"}\}$$

is niet berekenbaar door middel van een registermachine.

Anders gezegd, er bestaat geen registermachine die voor elke input n stopt en 1 antwoordt indien M_n stopt bij invoer n, en 0 indien M_n niet stopt bij invoer n. Uit de hypothese van Church volgt dan dat er geen algoritme bestaat voor het stop-probleem.

Bew. Neem de afbeelding $h : \mathbb{N} \to \mathbb{N}$ die overeenkomt met de verzameling van de stelling (volgens Definitie 5.1.4). Deze is gedefiniëerd door

$$\begin{array}{ll} h(n) = 1 \ , & \text{als } M_n \text{ bij input } n \text{ stopt na een eindig aantal stappen} \\ h(n) = 0 \ , & \text{in het ander geval.} \end{array}$$

We dienen dus aan te tonen dat h niet berekenbaar is door middel van een registermachine.

Het bewijs is uit het ongerijmde. Veronderstel dat h *wel* berekenbaar is door middel van een registermachine. Dan bestaat er een registermachine M die bij elke input $n \in \mathbb{N}$ stopt na een eindig aantal stappen met output $h(n)$ in de accumulator R_0. We moeten nu tot een contradictie trachten te komen.

Zij P het programma van M met N het rangnummer van de laatste instructie **S T O P**, en zij P' het programma bekomen door in het programma P deze instructie **STOP** te vervangen door de volgende 3 bevelen toe te voegen:

$$\begin{array}{ll} (N) & \mathbf{V\ S\ P}\ N+2 \\ (N+1) & \mathbf{S\ P\ R}\ N \\ (N+2) & \mathbf{S\ T\ O\ P} \end{array}$$

Zij M' de registermachine met programma P'. Als er op het ogenblik dat gesprongen wordt naar $N+1$ een nul staat in de accumulator, dan stopt M' bij de volgende stap. In het andere geval gaat M' in een oneindige lus.

We hebben dus:

$$\begin{array}{c} M' \text{ bij input } n \text{ stopt na een eindig aantal stappen} \\ \Updownarrow \\ M \text{ bij input } n \text{ stopt na een eindig aantal stappen en heeft dan als output 0 in de accumulator} \\ \Updownarrow \\ h(n) = 0 \\ \Updownarrow \\ M_n \text{ bij input } n \text{ stopt } \textit{niet} \text{ na een eindig aantal stappen} \end{array}$$

Zij e het codenummer van de registermachine M'. Dus M' is M_e. Dan hebben we voor elke $n \in \mathbb{N}$:

$$M_e \text{ bij input } n \text{ stopt na een eindig aantal stappen}$$
$$\Updownarrow$$
$$M_n \text{ bij input } n \text{ stopt } \textit{niet} \text{ na een eindig aantal stappen}$$

Stellen we nu n gelijk aan e, dan bekomen we de volgende contradictie:

$$M_e \text{ bij input } e \text{ stopt na een eindig aantal stappen}$$
$$\Updownarrow$$
$$M_e \text{ bij input } e \text{ stopt } \textit{niet} \text{ na een eindig aantal stappen}$$

Dit is natuurlijk onmogelijk. Dus was onze assumptie verkeerd. M.a.w. er bestaat geen registermachine M die kan beslissen of M_n stopt bij input n. Deze contradictie beëindigt het bewijs van de stelling. Q.E.D.

Er zijn dus verzamelingen van natuurlijke getallen die onberekenbaar zijn.

5.3 Onbeslisbaarheid van $\mathbb{N}$

Zij $\Sigma_{+\times}$ het vocabularium bestaande uit:

- functiesymbolen $PLUS$ en $MAAL$, elk met twee argumenten,
- twee constanten $ZERO$ en ONE.

Met $\mathbb{N}$ bedoelen we de structuur van $\Sigma_{+\times}$

- domein $\mathbb{N}$,
- $PLUS$ wordt interpreteert als $+$,
- $MAAL$ als $\times$,
- $ZERO$ als 0 en ONE als 1.

We kunnen dus "spreken" over $\mathbb{N}$ in de predikatenlogicataal van $\Sigma_{+\times}$.

Stelling 5.3.1 (Onbeslisbaarheid van $\mathbb{N}$)**.** *Er bestaat geen algoritme om na te gaan of een zin van $\Sigma_{+\times}$ al dan niet waar is in $\mathbb{N}$.*

Bew. Het bewijs van deze stelling is te moeilijk om hier te geven, maar we leggen wel uit op welk idee het bewijs berust. Men bewijst eerst dat het stop-probleem "definieerbaar" is in $\Sigma_{+\times}$, daarmee bedoelt men dat er een formule $A[v_1]$ met één vrije variabele v_1 bestaat zodat voor alle $n \in \mathbb{N}$ geldt:

$$M_n \text{ bij input } n \text{ stopt na eindig aantal stappen}$$
$$\Updownarrow$$
$$\mathbb{N} \models A[\underbrace{1+1+\cdots+1}_{n \text{ keer}}]$$

Het bewijs van de "definieerbaarheid" van het stop-probleem is moeilijk – we geven het niet hier. Het is eenvoudig om in te zien dat dit de Stelling impliceert. Inderdaad, als er een algoritme zou bestaan om na te gaan of een willekeurige zin van $\Sigma_{+\times}$ waar is in $\mathbb{N}$, dan zouden we dat algoritme kunnen gebruiken om te testen of $\mathbb{N} \models A[1+1+\cdots+1]$. Dit zou een algoritme geven om na te gaan of M_n stopt bij input n, maar zo'n algoritme bestaat niet. Strikt genomen hebben we enkel bewezen dat er geen registermachine bestaat die dit kan, maar door de hypothese van Church volgt dat het ook niet kan in andere algoritmische talen. Q.E.D.

Opmerking 5.3.1. Het is gekend dat ook $\mathbb{Z}$ en $\mathbb{Q}$ onbeslisbaar zijn. Nochtans bewees Tarski dat $\mathbb{R}$ beslisbaar is, t.t.z. er bestaat wel een algoritme om na te gaan of een zin van $\Sigma_{+\times}$ al dan niet waar is in $\mathbb{R}$.

Tenslotte willen we nog het resultaat van Matijasevič (1970) vermelden:
Er bestaat geen algoritme om na te gaan of een vergelijking

$$Q_1(x_1, \cdots, x_n) = Q_2(x_1, \cdots, x_n) \tag{5.1}$$

met Q_1, Q_2 veeltermen in meerdere variabelen $x_1, \cdots, x_n$ met coëfficiënten in $\mathbb{N}$, al dan niet een oplossing heeft in natuurlijke getallen $x_1, \cdots, x_n$. (Opgepast: voor $n = 1$ bestaat er wel een algoritme want dan is elke oplossing $x_1 \in \mathbb{N}$ een deler van de constante term van $Q_1 - Q_2$). Merk op dat het resultaat van Matijasevič de onbeslisbaarheid van $\mathbb{N}$ impliceert. (Een goede referentie voor het bewijs van Matijasevič is het artikel van M. Davis : Hilbert's tenth problem is unsolvable, Amer. Math. Monthly 80 (1973), p. 233-269.) Het is niet gekend of er al dan niet een algoritme bestaat om na te gaan of een vergelijking van de vorm (5.1) een oplossing heeft in rationale getallen $x_1, \cdots, x_n$.

5.4 De Onvolledigheidsstelling van Gödel

De structuur $\mathbb{N}$ is één der meest fundamentele objecten in de wiskunde. De studie van de eigenschappen van de natuurlijke getallen (getaltheorie, getallenleer) is moeilijk en veel problemen zijn nog steeds niet opgelost. Zo vermoedt men bijvoorbeeld dat elk even natuurlijk getal (groter dan twee) kan geschreven worden als de som van twee priemgetallen. Dit is het vermoeden van Goldbach (1742) dat ondertussen al bijna 300 jaar oud is en nog steeds onbewezen. Ondertussen werd voor het bewijzen van de waarheid of onwaarheid ervan een prijs van één miljoen dollar uitgeloofd door het Clay Mathemical Institute.

Vele eigenschappen van de natuurlijke getallen zijn verre van evident. Bijvoorbeeld: een priemgetal p kan geschreven worden als de som van twee kwadraten van natuurlijke getallen asa p is een veelvoud van 4 plus 1 of $p = 2$. Zo'n eigenschappen die niet evident zijn wil men kunnen bewijzen.

Om eigenschappen van de natuurlijke getallen te bewijzen gaat men als volgt te werk. Men stelt verzamelingen van axioma's op die "evident waar" zijn in $\mathbb{N}$. Men noemt zo'n verzameling een *axiomasysteem*. Men probeert daaruit andere niet evidente eigenschappen te bewijzen.

Zo'n axiomasysteem kan oneindig zijn. Maar dan eisen we dat er een algoritme bestaat om na te gaan of een uitspraak behoort tot het axiomasysteem T (Deze voorwaarde is altijd voldaan indien T eindig is.) Anders zou ons systeem T onbruikbaar zijn!

De Peano axioma's PA vormen duidelijk zo een axiomasysteem. Het is oneindig door het inductieschema:

$$A[0] \wedge (\forall x)(A[x] \Rightarrow A[x+1]) \Rightarrow (\forall x)A[x]$$

Er bestaat duidelijk een eindigend algoritme om na te gaan of een zin al dan niet een instantiatie is van dit schema.

Vóór 1931 dacht men dat elke ware uitspraak over $\mathbb{N}$ (geformuleerd als een zin in $\Sigma_{+\times}$) bewijsbaar zou zijn uit PA. Het was dan ook een dramatische verrassing toen Kurt Gödel in 1931 bewees dat dit niet waar is. Op zich hoeft dat niet erg te zijn. Misschien bestaat er een ander axiomasysteem T zodat elke ware uitspraak over $\mathbb{N}$ bewijsbaar is uit dat systeem T? Helaas, dit is niet mogelijk. Gödel bewees dat zo een axiomasysteem T niet kan bestaan! Dit is de beroemde *Onvolledigheidsstelling van Gödel*.

Definitie 5.4.1. Een theorie T in $\Sigma_{+\times}$ (t.t.z. een verzameling zinnen van $\Sigma_{+\times}$) heet *berekenbaar* er een algoritme bestaat om na te gaan of een zin al dan niet tot T behoort.

Voorbeeld 5.4.1. De theorie PA bestaande uit de Peano axioma's, is berekenbaar. Inderdaad, het is duidelijk dat men op algoritmishe wijze kan nagaan of een zin al dan niet een Peano axioma is (omdat de (oneindige) lijst der Peano axioma's heel doorzichtig is). Uit de hypothese van Church volgt dan dat dit kan gedaan worden door middel van een registermachine. (Het gebruik van de hypothese van Church kan zoals altijd vermeden worden en vervangen worden door een saaie en omslachtige oefening in het programmeren van registermachines.)

Stelling 5.4.1 (De onvolledigheidsstelling van Gödel (1931))**.** *Zij T een verzameling van zinnen van $\Sigma_{+\times}$. Veronderstel dat*

1. *$\mathbb{N}$ is een model voor T (d.w.z. elke zin van T is waar in $\mathbb{N}$), en*
2. *T is berekenbaar (d.w.z. er bestaat een algoritme om na te gaan of een zin al dan niet tot T behoort).*

Dan bestaat er een zin A in $\Sigma_{+\times}$ zodat A waar is in $\mathbb{N}$ maar zodat A toch geen logisch gevolg is van T.

Opmerking 5.4.1. Die zin A is dan zeker niet bewijsbaar uit T, want als A bewijsbaar zou zijn door alleen maar te steunen op (de waarheid in $\mathbb{N}$ van) de zinnen in T, dan moet A waar zijn in elk model van T en dan zou A een logisch gevolg zijn van T wat niet het geval is.

Opmerking 5.4.2. Indien we voor T de verzameling PA der Peano axioma's nemen, dan bekomen we dat er een zin in $\Sigma_{+\times}$ bestaat die waar is in $\mathbb{N}$ maar niet bewijsbaar uit PA. (Inderdaad PA is berekenbaar.)

Opmerking 5.4.3. Indien T een ander axiomasysteem voor $\mathbb{N}$ is, zoals besproken in het begin van deze sectie, dan zijn de voorwaarden (1) en (2) in Stelling 5.4.1 voldaan. Merk op dat we niet moeten eisen dat de axioma's "evident"zijn. Het begrip "evident"kan trouwens niet wiskundig gedefinieerd worden!

Opmerking 5.4.4. Zonder voorwaarde (2) zou Stelling 5.4.1 niet waar zijn, want dan zouden we voor T de verzameling van alle ware zinnen (in $\mathbb{N}$) kunnen nemen. Deze verzameling is echter niet berekenbaar, want $\mathbb{N}$ is onbeslisbaar!

Bew. Bewijs van de onvolledigheidsstelling.

Uit het ongerijmde. Veronderstel dat voor elke zin A geldt dat

$$\text{als } A \text{ waar is in } \mathbb{N}, \text{ dan is } A \text{ een logisch gevolg van } T \tag{5.2}$$

Zij A een willekeurige zin van $\Sigma_{+\times}$. Dan is A ofwel waar, ofwel onwaar in $\mathbb{N}$. Als A waar is, dan hebben we volgens (5.2) dat A een logisch gevolg is van T. Als A onwaar is, dan hebben we dat $\neg A$ waar is, en dus volgens (5.2) dat $\neg A$ een logisch gevolg is van T. We besluiten dus

$$\text{voor elke zin } A \text{ geldt dat ofwel } A, \text{ ofwel } \neg A, \text{ een logisch gevolg is van } T \tag{5.3}$$

Indien T eindig is, dan weten we op grond van de semi-beslisbaarheid van de predikatenlogica (zie Hoofdstuk 3) dat er een computerprogramma P_1 bestaat dat bij input A stopt na een eindig aantal stappen met output "ja", als en slechts als A een logisch gevolg is van T. Deze bewering blijft ook geldig wanneer T oneindig is, op voorwaarde dat T berekenbaar is (dit moet je zelf kunnen inzien!).
Omwille van dezelfde reden bestaat er ook een computerprogramma P_2 dat bij input A stopt na een eindig aantal stappen met output "nee", als en slechts als $\neg A$ een logisch gevolg is van T.
Bij input A laten we tijdens de dag het programma P_1 werken, en tijdens de nacht het programma P_2. Omwille van (5.3) komt er dan steeds na een eindige tijd een antwoord "ja" of een antwoord "nee", en op dat ogenblik breken we de werking van beide programma's af. Het gegeven antwoord is "ja" indien A waar is in $\mathbb{N}$ (want dan is, volgens (5.2), de zin A een logisch gevolg van T), en het gegeven antwoord is "nee" in het andere geval. Maar dit geeft ons dus (steunende op de onware veronderstelling (5.2) van ons bewijs uit het ongerijmde) een algoritme (dat steeds na een eindig aantal stappen stopt) om na te gaan of een zin A al dan niet waar is in $\mathbb{N}$. Dit is in strijd met de onbeslisbaarheid van $\mathbb{N}$. Deze contradictie beëindigt het bewijs van de onvolledigheidsstelling. Q.E.D.

De *Onvolledigheidsstelling van Gödel* heeft drastisch onze "filosofischeöpvattingen over de wiskunde veranderd. Voordien geloofde men een axiomasysteem te kunnen opstellen voor de ganse wiskunde, zodat elke ware uitspraak in de wiskunde uit dit axiomasysteem zou kunnen bewezen worden. Dit was de ambitie van de *axiomatische methode*. Eén van de grote voorvechters in de jaren 20 van vorige eeuw daarvan was de Duitse wiskundige Hilbert. Maar dat bleek dus onmogelijk.

Nochtans blijft de axiomatische methode nuttig en onontbeerlijk. Uit de praktijk blijkt bijvoorbeeld dat *bijna* alle eigenschappen van de natuurlijke getallen, die men ooit heeft kunnen bewijzen, reeds bewijsbaar zijn uit PA.