Universiteit Utrecht

Departement Wiskunde

Bachelorscriptie

Berekenbaarheid en Realiseerbaarheid

Auteur:
Sophie Huiberts

Begeleider:
Dr. Jaap van Oosten

September 2015-Januari 2016

Inhoudsopgave

0.1 Inleiding

De theorie van berekenbaarheid heeft een nauwe band met wiskundige logica, en in het bijzonder met bewijstheorie. De eerste motivatie voor de grondleggers van de berekenbaarheidstheorie, Alonzo Church en Alan Turing, was zelfs een vraagstuk uit de logica: het *Entscheidungsproblem*. Het Entscheidungsproblem is een vraagstuk van de duitse wiskundige David Hilbert, en vraagt of er een algoritme bestaat dat, gegeven een logische zin en eventueel een verzameling axioma's, kan bepalen of deze zin bewezen kan worden uit de axioma's. Church en Turing publiceerden in 1936 onafhankelijk hun papers [1] en [12] waarin ze deze vraag negatief beantwoorden: er bestaat geen algoritme dat voor iedere logische zin kan bepalen of deze wel of niet bewijsbaar is.

De meest beroemde resultaten op dit gebied zijn echter zonder twijfel de onvolledigheidsstellingen van Gödel: een voldoende sterke theorie kan niet zowel volledig als consistent zijn, en deze theorieën zijn ook niet in staat om hun eigen consistentie te bewijzen. De samenwerking tussen de berekenbaarheidstheorie en de logica heeft nog veel meer resultaten geleverd, meer dan dat we hier kunnen opnoemen. In deze scriptie bespreken we een bewijsmethode genaamd realiseerbaarheid, dat een verband vormt tussen berekenbaarheid en bewijsbaarheid in intuïtionistische logica.

Eerst zullen we een korte introductie in berekenbaarheidstheorie geven. Daarna geven we een korte inleiding in intuïtionistische logica, om vervolgens door middel van onze kennis over berekenbaarheid een aantal conclusies te kunnen trekken over wat wel en niet bewijsbaar is.

Wie bekend is met het haltingprobleem, de Smn-stelling en het Kleene T-predicaat kan het eerste hoofdstuk overslaan zonder voorkennis te missen. Vanaf hoofdstuk 2 wordt de lezer ook verondersteld bekend te zijn met een aantal concepten uit eerste-orde logica en modeltheorie. Begrippen zoals taal, model, natuurlijke deductie en bewijsbomen zullen niet uitgelegd worden. Wie hier niet mee bekend is kan hoofdstuk 2 van [7] te lezen om de nodige voorkennis op te doen.

0.2 Bronvermelding

In het eerste hoofdstuk wordt berekenbaarheidstheorie behandeld, waar Jaap Van Oosten's dictaten [6] en [7] als bronnen dienden. De bron voor het tweede hoofdstuk was het boek van Troelstra en van Dalen [10]. De voornaamste bronnen voor het laatste hoofdstuk waren de papers [3] en [5] en het boek [10], andere bronnen zijn in de tekst vermeld. In hoofdstuk 1 volgen we ruwweg de behandeling van [6]. De rest van de hoofdstukken volgt een eigen behandeling. De definitie van de projectiefuncties j_1, j_2 in definitie 2 is eigen werk en heb ik niet in de literatuur kunnen vinden.

0.3 Notatie

Wanneer we een functie aanduiden doen we dat vaak zonder de functie een naam te geven. In plaats daarvan gebruiken we lambda-notatie om aan te geven wat de parameters van een functie zijn. De functie $\lambda x.x^2$ is de functie die x naar x^2 stuurt. We schrijven

een enkele lambda gevolgd door meerdere variabelen voor functies die meerdere parameters nodig hebben. Zo is $\lambda xy.x^y$ de functie die, gegeven getallen x en y, het getal x^y teruggeeft.We zullen ook functies nodig hebben die een getal naar een functie sturen. Dit noteren we met met meerdere lambdas achter elkaar. Als we de functie f definiëren als $\lambda x \lambda y.x^y$, dan is $f(5) = \lambda y.5^y$, en is dus $(f(5))(2) = 25$. We zullen $\lambda x_1 \cdots x_n F(x_1, \ldots, x_n)$ ook wel afkorten tot $\lambda \overline{x} F(\overline{x})$.

Hoofdstuk 1

Berekenbaarheidstheorie

We noemen een functie *effectief berekenbaar* als deze functie door een mens met een onbeperkte levensduur en voorraad pennen en papier kan worden berekend. Hierbij houden we geen rekening met hoe lang het zou duren om tot een antwoord te komen. Church en Turing hebben hun definities van berekenbaarheid geformuleerd in een poging om de berekenbare functies precies de effectief berekenbare functies te laten zijn. De aanname dat dit inderdaad de juiste formalisatie van berekenbaarheid is, wordt de *Church-Turing hypothese* genoemd.

1.1 Registermachines

In deze scriptie gebruiken we de registermachine als model van berekenbaarheid. Intuïtief lijkt de registermachine erg op een normale computer. De registermachine bestaat uit een geheugen voor variabelen, een plek waar een programma staat opgeslagen en een verwijzing naar de huidige instructie, die we de *instructiepointer* noemen. De registermachine heeft de mogelijkheid om het gegeven programma uit te voeren. Het geheugen van de registermachine bestaat uit aftelbaar oneindig veel registers $R_1, R_2, \ldots$, en in ieder register kan een natuurlijk getal worden opgeslagen. Wanneer de registermachine begint met het uitvoeren van een programma met n argumenten, bevatten alle registers waarde 0, behalve de eerste n registers, die de invoer van het programma bevatten. Wanneer het programma stopt, staat in het eerste register de uitvoer van het programma.

De registermachine kan een aantal verschillende instructies uitvoeren, en programma's worden gecodeerd door een lijst instructies.

- Het uitvoeren van een instructie $(+, i, n)$ bestaat uit het optellen van 1 bij de waarde van register R_i, en door te gaan met de n-de instructie
- Het uitvoeren van een instructie $(-, i, n, m)$ gaat als volgt:
  - Als register R_i waarde 0 bevat, gaat de registermachine door met de n-de instructie.
  - Anders wordt de waarde van register R_i met 1 verlaagd, en gaat de registermachine door met de m-de instructie.

De registermachine stopt wanneer deze de n-de instructie wil uitvoeren, maar het programma minder dan n instructies bevat.

Voorbeeld 1. *Maak de waarde van register R_1 gelijk aan* 1.

1. $(-, 1, 2, 1)$
2. $(+, 1, 3)$

Voorbeeld 2. *Tel de waarden van R_2 bij de waarde van R_1 op.*

1. $(-, 2, 3, 2)$
2. $(+, 1, 1)$

Voorbeeld 3. *Vermenigvuldig de waarden van R_1 en R_2. Het tussenresultaat berekenen we in R_4. Dit doen we door, zo lang de waarde van R_1 groter is dan* 0*, de waarde van R_2 op te tellen bij R_3 en R_4, vervolgens de waarde van R_3 terug te zetten in R_2, de waarde van R_1 met* 1 *te verlagen en opnieuw te beginnen.*

1. $(-, 1, 7, 2)$
2. $(-, 2, 5, 3)$
3. $(+, 3, 4)$
4. $(+, 4, 2)$
5. $(-, 3, 1, 6)$
6. $(+, 2, 5)$
7. $(-, 4, 9, 8)$
8. $(+, 1, 7)$

Voorbeeld 4. *Ga naar instructie n als $R_1 \geqslant R_2$ en anders naar instructie m.*

1. $(-, 2, n, 2)$
2. $(-, 1, m, 1)$

Definitie 1. *De* compositie *PQ van twee programma's P en Q krijgen we door eerst P uit te voeren, en als P stopt, daarna Q uit te voeren op het geheugen zoals P het achter heeft gelaten. De compositie van P en Q stopt alleen als P stopt op zijn invoer en Q stopt op het geheugen zoals het was toen P stopte. We vinden PQ als volgt. Stel n gelijk aan het aantal instructies in P. Definieer nu P' door iedere instructie die verwijst naar een instructie met een index groter dan n, te laten verwijzen naar $n + 1$. Definieer Q' door in Q iedere verwijzing naar instructies met n te verhogen. We vinden PQ nu als P' gevolgd door Q'.*

Voorbeeld 5. *Stel P wordt gegeven door*

1. $(-, 1, 2, 1)$
2. $(+, 1, 3)$
3. $(-, 2, 4, 3)$
4. $(+, 2, 9)$

en Q wordt gegeven door

1. $(-, 1, 3, 2)$
2. $(+, 2, 1)$

Dan wordt PQ gegeven door

1. $(-, 1, 2, 1)$
2. $(+, 1, 3)$
3. $(-, 2, 4, 3)$
4. $(+, 2, 5)$
5. $(-, 1, 7, 6)$
6. $(+, 2, 5)$

Merk op dat niet ieder registermachineprogramma ooit hoeft te stoppen met uitvoeren.

Voorbeeld 6. *Dit registermachineprogramma stopt op geen enkele invoer.*

1. $(+, 1, 1)$

Het stoppen van programma's speelt binnen de berekenbaarheidstheorie een grote rol, dus we zullen hier later nog op terug komen. Vanaf nu, wanneer we het over een berekenbare functie f hebben, bedoelen we dat er een registermachineprogramma F bestaat zo dat voor iedere $\vec{k} \in \mathbb{N}^n$ geldt dat, als $f(\vec{k})$ gedefinieerd is, dat P stopt op invoer $\vec{k}$ met als uitvoer $f(\vec{k})$, en dat als $f(\vec{k})$ niet gedefinieerd is, dat P niet stopt op invoer $\vec{k}$.

1.2 Recursieve functies

Een andere klasse functies waarvan aannemelijk is dat ze effectief berekenbaar zijn, zijn de *recursieve functies*. In de paragraaf hierna zullen we zien dat de berekenbare functies allemaal recursief zijn, en dat alle recursieve functies berekenbaar zijn. Recursieve functies zijn echter overzichtelijker te specificeren.

Voor het opbouwen van recursieve functies beginnen we met een kleinere verzameling functies, de primitief recursieve functies. Dit is de kleinste verzameling die voldoet aan

- De nulfunctie Z waar $Z(n) = 0$ voor iedere $n \in \mathbb{N}$, is primitief recursief.

- De opvolgersfunctie S waar $S(n) = n + 1$ voor iedere $n \in \mathbb{N}$, is primitief recursief.
- De projectiefuncties Π_i^j, waar $\Pi_i^j(x_1, \ldots, x_j) = x_i$ voor iedere $x_1, \ldots, x_i \in \mathbb{N}$, waar $1 \leqslant i \leqslant j$, zijn primitief recursief.
- **Compositie:** Als f een primitief recursieve functie in n argumenten is, en $g_1, \ldots, g_n$ primitief recursieve functies in m argumenten zijn, dan is

  $$f(g_1(x_1, \ldots, x_m), \ldots, g_n(x_1, \ldots, x_m))$$

  een primitief recursieve functie in m argumenten.
- **Primitieve recursie:** Als f en g primitief recursief zijn, in respectievelijk n en $n + 2$ argumenten, dan is de functie G gegeven door

  $$\begin{aligned} G(0, x_1, \ldots, x_n) &= f(x_1, \ldots, x_n) \\ G(k + 1, x_1, \ldots, x_n) &= g(k, G(k, x_1, \ldots, x_n), x_1, \ldots, x_n) \end{aligned}$$

  primitief recursief in $n + 1$ argumenten. Voor $n = 0$ stellen we $G(0) = k$ voor constante $k \in \mathbb{N}$.

Om deze nogal abstracte definities wat tastbaarder te maken, volgen nu een aantal voorbeelden van definities van primitief recursieve functies.

Voorbeeld 7. *Optellen*

$$\begin{aligned} Plus(0, m) &= \Pi_1^1(m) \\ Plus(n + 1, m) &= S(\Pi_2^3(n, Plus(n, m), m)) \end{aligned}$$

Voorbeeld 8. *Vermenigvuldigen*

$$\begin{aligned} Keer(0, m) &= 0 \\ Keer(n + 1, m) &= Plus(\Pi_3^3(n, Keer(n, m), m), \Pi_2^3(n, Keer(n, m), m)) \end{aligned}$$

Voorbeeld 9. *We kunnen ook aftrekken. Omdat we alleen met gehele getallen werken, stellen we $n \dot{-} m = n - m$ als $n > m$ en $n \dot{-} m = 0$ als $n \leqslant m$.*

$$\begin{aligned} MinEen(0) &= 0 \\ MinEen(n + 1) &= \Pi_1^2(n, MinEen(n)) \\ OmgekeerdMin(0, n) &= n \\ OmgekeerdMin(m + 1, n) &= MinEen(\Pi_2^3(m, OmgekeerdMin(m, n), n)) \\ n \dot{-} m &= OmgekeerdMin(\Pi_2^2(n, m), \Pi_1^2(n, m)) \end{aligned}$$

Het symmetrisch verschil $|n - m|$ kunnen we nu vinden met $(n \dot{-} m) + (m \dot{-} n)$

Voorbeeld 10. *Gegeven x, y, z, geef y terug als $x = 0$ en z als $x > 0$.*

$$\begin{aligned} Alsdan(0, y, z) &= \Pi_1^2(y, z) \\ Alsdan(x + 1, y, z) &= \Pi_4^4(x, Alsdan(x, y, z), y, z) \end{aligned}$$

We zien dat we in een primitief recursieve functie ook gevalsonderscheidingen kunnen doen. Vanaf nu zullen we iets losser omgaan met het definiëren van functies om de leesbaarheid te verhogen.

Soms is een functie makkelijk te definiëren door zijn inverse. Gegeven getallen a, b waar b ongelijk is aan 0 is $\lfloor a/b \rfloor$ primitief recursief te bepalen door eerst te overschatten, en vervolgens omlaag te gaan totdat we het juiste getal tegenkomen.

Voorbeeld 11.

$$\begin{aligned} helper(0, a, b) &= 0 \\ helper(n+1, a, b) &= \begin{cases} helper(n, a, b) & als\ (n+1)b \dot{-} a \neq 0 \\ n+1 & anders \end{cases} \\ Deling(a, b) &= helper(a, a, b) \end{aligned}$$

Definitie 2. *De* paringsfunctie j *is een functie die we nog vaker zullen gebruiken. Deze functie gebruiken we om een tweetal getallen te coderen in één getal. We definiëren ook de projectiefuncties* j_1, j_2*, die we gebruiken om de oorspronkelijke getallen terug te vinden. Dus* $j_1(j(a, b)) = a$ *en* $j_2(j(a, b)) = b$*.*

$$\begin{aligned} j(a, b) &= \frac{(a+b)^2 + 3a + b}{2} \\ helper(0, m) &= 0 \\ helper(n+1, m) &= \begin{cases} n & als\ \frac{n^2+n}{2} \dot{-} m = 0 \\ helper(n, m) & anders \end{cases} \\ j_1(n) &= n \dot{-} \frac{helper(n, n)^2 + helper(n, n)}{2} \\ j_2(n) &= helper(n, n) \dot{-} j_2(n) \end{aligned}$$

De functie j zoals we deze hier hebben gegeven vormt een bijectie tussen $\mathbb{N} \times \mathbb{N}$ en $\mathbb{N}$, en is volgens de Fueter-Pólya stelling, naast zijn "omgekeerde" functie $\lambda baj(a, b)$, de enige kwadratische functie met deze eigenschap. De helperfunctie die wordt gebruikt in de definitie van j_1 en j_2 zoekt de grootste n zodat $(n^2 + n)/2$ kleiner of gelijk is aan m. Dit is mogelijk doordat we een bovengrens hebben voor de waarde die we zoeken, zodat we vanaf boven kunnen zoeken totdat n voldoet. De helperfunctie vindt zo de waarde van $a + b$.

Het zal blijken dat niet iedere berekenbare functie primitief recursief is. De berekening van een primitief recursieve functie stopt bijvoorbeeld altijd. Een voorbeeld van een totale functie die wel berekenbaar is, maar niet primitief recursief, is de Ackermann-functie. Hieronder geven we een definitie van Rosza Péter.

Voorbeeld 12.

$$\begin{aligned} A(0, m) &= m + 1 \\ A(n+1, 0) &= A(n, 1) \\ A(n+1, m+1) &= A(n, A(n+1, m)) \end{aligned}$$

Kenmerkend aan deze functie is dat de waardes heel snel groeien. De eerste waardes zijn $A(0,0) = 1, A(1,1) = 3, A(2,2) = 7$, maar de waarde van $A(4,4)$ is al $2^{2^{2^{65536}}} - 3$. De Ackermann-functie groeit sneller dan elke primitief recursieve functie. Zonder een volledig bewijs te geven, merken we op dat door structurele inductie op de opbouw van primitief recursieve functies te bewijzen is dat voor iedere primitief recursieve functie F een getal n bestaat zodat $F(x_1, \ldots, x_k) < A(n, x_1 + \cdots + x_k)$. De Ackermann-functie is dus niet primitief recursief.

Deze recursie loopt daarentegen wel gegarandeerd altijd af. Voor $A(0, m)$ is dit duidelijk te zien. Stel nu dat $A(n, m)$ gedefinieerd is voor iedere m. Uit de tweede regel van de definitie volgt dat $A(n+1, 0)$ is gedefinieerd. Als $A(n+1, k)$ is gedefinieerd voor alle $k < l$, weten we dat $A(n+1, l)$ is gedefinieerd, en omdat $A(n, m)$ is gedefinieerd voor iedere m, volgt dat ook $A(n, A(n+1, l)) = A(n+1, l+1)$ is gedefinieerd. De functie is dus totaal, maar niet primitief recursief.

Om toch alle functies te kunnen definiëren, vullen we de mogelijke operaties aan met onbegrensd zoeken. De resulterende verzameling functies noemen we de partieel recursieve of μ-recursieve functies, en is de kleinste verzameling die voldoet aan:

- Alle primitief recursieve functies zijn μ-recursief.
- De μ-recursieve functies zijn gesloten onder compositie.
- De μ-recursieve functies zijn gesloten onder primitieve recursie voor partiële functies.
- De μ-recursieve functies zijn gesloten onder minimalisatie. Als $f(x, \overline{y})$ mu-recursief is, dan is de functie die voor gegeven $\overline{y}$ de kleinste x vindt zodat $f(x, \overline{y}) = 0$, ook mu-recursief. We noteren deze functie als $\lambda \overline{y}.\mu x f(x, \overline{y}) = 0$. Er geldt dat de berekening voor $\mu x f(x)$ stopt precies als er een n is zodat de berekening voor $f(n)$ stop, $f(n) = 0$ en voor iedere $m < n$ geldt dat de berekening voor $f(m)$ stopt en $f(m) \neq 0$.

De uitkomst van een μ-recursieve functie is alleen gedefinieerd als alle invoer gedefinieerd is. Als F op geen enkele invoer stopt, dan stopt $\Pi^2_1(0, F(x))$ voor geen enkele x. Hetzelfde geldt voor primitieve recursie: $F(n, \overline{x})$ is alleen gedefinieerd als de berekening voor $F(i)$ stopt voor $0 \leqslant i \leqslant n$.

We zullen ook wel $F(\overline{x})\downarrow$ schrijven om aan te geven dat de berekening van F op invoer $\overline{x}$ stopt.

1.3 Equivalentie van recursieve functies en registermachinefuncties

We zullen nu zien dat iedere recursieve functie berekend kan worden door een registermachine, en dat iedere functie die een registermachine kan berekenen recursief is. Eerst laten we zien dat de recursieve functies berekend kunnen worden door een registermachine.

- De nulfunctie Z kan berekend worden door een registermachine:

1. $(-, 1, 2, 1)$

- De opvolgersfunctie S kan berekend worden door een registermachine:

  1. $(+, 1, 2)$

- De projectiefuncties Π_i^j kunnen berekend worden door een registermachine:

  1. $(-, 1, 2, 1)$
  2. $(-, i, 4, 3)$
  3. $(+, 1, 2)$

- De compositie van recursieve functies f en $g_1, \ldots, g_n$, met registermachineprogramma's $F, G_1, \ldots, G_n$, kan worden berekend door een registermachine. Neem m het aantal argumenten van de G_i, en neem k zo dat R_k het hoogste register is dat wordt gebruikt door de programma's $F, G_1, \ldots, G_n$. We berekenen de compositie van de functies $f, g_1, \ldots, g_n$ door de invoer en tussenresultaten te bewaren in registers die niet gebruikt worden door $F, G_1, \ldots, G_n$.

  1. Kopieer de waarden van $R_1, \ldots, R_m$ naar $R_{k+1}, \ldots, R_{k+m}$.
  2. Voer G_1 uit.
  3. Verplaats de waarde van R_1 naar R_{k+m+1}.
  4. Stel de waarde van alle registers tot en met R_k op 0.
  5. Kopieer de waarden van $R_{k+1}, \ldots, R_{k+m}$ naar $R_1, \ldots, R_m$.
  6. Voer G_2 uit.

     ...
  7. Voer G_n uit.
  8. Verplaats de waarde van R_1 naar R_{k+m+n}.
  9. Stel de waarde van alle registers tot en met R_k op 0.
  10. Verplaats de waarden van $R_{k+m+1}, \ldots, R_{k+m+n}$ naar $R_1, \ldots, R_n$.
  11. Voer F uit.

- Stel h wordt gegeven door primitieve recursie uit f en g, en f en g worden berekend met programma's F en G. Neem n zo dat f precies n argumenten heeft en g precies $n + 2$ argumenten. Neem k zo dat R_k het hoogste register is dat wordt gebruikt door de programma's F, G en stel $N = k + n + 2$. In R_N onthouden we hoe vaak G is uitgevoerd.

  1. Verplaats de waarden van $R_1, \ldots, R_{n+1}$ naar $R_{k+1}, \ldots, R_{k+n+1}$.
  2. Kopieer de waarden van $R_{k+2}, \ldots, R_{k+n+1}$ naar $R_1, \ldots, R_{n-1}$.
  3. Voer F uit.
  4. Stel de waarde van registers $R_2, \ldots, R_k$ op 0.
  5. Verplaats de waarde van R_1 naar R_2.

6. Tel 1 op bij de waarde van R_N en kopieer naar R_1.
7. Kopieer de waarden van $R_{k+2}, \ldots, R_{k+n+1}$ naar $R_3, \ldots, R_{n+1}$.
8. Voer G uit.
9. Stop als R_{k+1} dezelfde waarde als R_N bevat, ga anders door bij stap 4.

- Stel g wordt gegeven door $\lambda x_1 \cdots x_n.\mu y f(y, x_1, \ldots, x_n) = 0$, en f wordt berekend door F. Neem k zo dat R_k het hoogste register is dat wordt gebruikt door F en stel $N = k + n + 2$. In R_N zullen we de geschikte y proberen te vinden.

  1. Verplaats de waarden van $R_1, \ldots, R_n$ naar $R_{k+1}, \ldots, R_{k+n+1}$.
  2. Kopieer de waarden van $R_{k+1}, \ldots, R_{k+n+1}$ naar $R_2, \ldots, R_{n+1}$.
  3. Kopieer de waarde van R_N naar R_1.
  4. Voer F uit.
  5. Als R_1 waarde 0 bevat, ga door bij stap 6, ga anders door bij stap 7.
  6. Verplaats de waarde van R_N naar R_1 en stop.
  7. Verhoog de waarde van R_N met 1.
  8. Stel de waarde van registers $R_1, \ldots, R_k$ op 0.
  9. Ga door bij stap 2

De andere kant op zullen we op een iets sterkere manier bewijzen, om direct een belangrijke stelling te bewijzen: er bestaat een berekenbare functie die iedere andere berekenbare functie uit kan rekenen. Hiervoor zullen we eerst wat voorwerk moeten doen.

Om eigenschappen van de recursieve functies te bewijzen, zullen we recursieve functies coderen in getallen. Daarvoor zullen we eerst een lijst getallen moeten coderen in een enkel getal. Dit doen we door de getallen $c_1, \ldots, c_n$ te coderen als $j(n, j(c_1, j(c_2, \ldots j(c_n, 0))) \ldots))$. Een registermachineprogramma kunnen we nu coderen in een getal door $(+, a, b)$ te coderen als $j(1, j(a, j(b, 0)))$, een instructie $(-, a, b, c)$ te coderen als $j(2, j(a, j(b, j(c, 0))))$, en een programma te coderen als de lijst van codes van de instructies. Als een getal een recursieve functie codeert, noemen we dat getal ook wel een *index* van die functie.

De manier van het coderen van lijsten van getallen zoals we hier doen stelt ons in staat om primitief recursief een aantal operaties op lijsten uit te voeren, zoals het vinden van de lengte van een lijst of een element op een gegeven positie binnen een lijst:

$$\begin{aligned} \mathit{lengte}(l) &= j_1(l) \\ \mathit{element}(0, l) &= j_1(l) \\ \mathit{element}(i+1, l) &= j_2(\mathit{element}(i, l)) \end{aligned}$$

We zullen $\mathit{element}(i, l)$ ook wel schrijven als $(l)_i$. Let op dat we het eerste element van de lijst vinden met $(l)_1$.

Nu definiëren we de primitief recursieve functies T, U. De functie T wordt het *Kleene T-predicaat* genoemd, en de functie U staat bekend als de *uitkomstfunctie*. Deze vormen samen een *universele recursieve functie*, en kunnen gebruikt worden om iedere recursieve

functie te berekenen, net zoals hoe een moderne computer allerlei verschillende computerprogramma's kan uitvoeren. Het T-predicaat $T(n, e, i, u)$ heeft 4 argumenten. Als eerst het aantal argumenten dat het uit te voeren programma nodig heeft, dan een index van het programma om uit te voeren, dan een lijst met invoer en als laatst een volledige geschiedenis van het uitvoeren van het gegeven programma op de gegeven invoer. Een $u \in \mathbb{N}$ die voldoet aan $T(n, e, i, u)$ codeert een lijst $x_1, \ldots, x_t$ zo dat het programma gecodeerd door e met invoer gecodeerd door i na t stappen stopt, en iedere x_k van de vorm $j(p, g)$ is, waar p naar de huidige instructie wijst, en g een lijst is met alle geheugenwaardes na k stappen. Omdat iedere stap maar één registerwaarde veranderd, en de rest waarde 0 heeft, hoeft g maar een eindig lange lijst te coderen. Het getal u codeert de geschiedenis van de machine die het programma uitvoert. De functie $U(u)$ vindt vervolgens de uitkomst die bij deze geschiedenis hoort.

Om te bevestigen of u inderdaad een geldige geschiedenis is voor het programma met index e op invoer $(i)_1, \ldots, (i)_n$ moet een aantal voorwaarden worden gecontroleerd:

- Het geheugen op het eerste tijdstip is gelijk aan de invoer.
- Op het eerste tijdstip wordt de eerste instructie uitgevoerd.
- Op het laatste tijdstip wordt gepoogd een niet-bestaande instructie uit te voeren.
- Tussen opeenvolgende tijdstippen verandert maximaal 1 geheugenwaarde.
- Alle instructies worden correct uitgevoerd.

$$
\begin{aligned}
Pointer(u, s) &= ((u)_s)_1 \\
Geheugen(u, s) &= ((u)_s)_2 \\
Vergelijk(n, i, u) &= \begin{cases} 0 \text{ als } |(Geheugen(u, n))_n - (i)_n| = 0 \\ 1 \text{ anders} \end{cases} \\
Vergelijkrij(0, i, u) &= 0 \\
Vergelijkrij(n+1, i, u) &= Vergelijk(n+1, i, u) + Vergelijkrij(n, i, u) \\
Geheugenklopt(e, u, k) &= \begin{cases} 1 \text{ als } Vergelijkrij(u, Geheugen(u, k), Geheugen(u, k \dot{-} 1)) \dot{-} \\ \qquad Vergelijk(j_1(j_2((e)_{Pointer(u,k)})), Geheugen(u, k), u) \neq 0 \\ 0 \text{ anders} \end{cases} \\
Plusgoed(e, u, k, h) &= \begin{cases} 1 \text{ als } (Geheugen(u, k))_{j_2(j_1(h))} \neq (Geheugen(u, k \dot{-} 1))_{j_2(j_1(h))} + 1 \\ 1 \text{ als } j_1(j_2(j_2((e)_{Pointer(u,k \dot{-} 1)}))) \neq Pointer(k) \\ 0 \text{ anders} \end{cases} \\
Mingoed(e, u, k, h) &= \begin{cases} 1 \text{ als } (Geheugen(u, k))_{j_2(j_1(h))} \neq (Geheugen(u, k \dot{-} 1))_{j_2(j_1(h))} \dot{-} 1 \\ 1 \text{ als } \text{ en } j_1(j_2(j_2(h)) \neq Pointer(k) \\ 1 \text{ als } \text{ en } j_1(j_2(j_2(j_2(h)) \neq Pointer(k) \\ 0 \text{ anders} \end{cases}
\end{aligned}
$$

$$Stap(e,u,k) = \begin{cases} 1 \text{ als } Geheugenklopt(e,u,k) \neq 0 \\ 1 \text{ als } j_1((e)_{Pointer(u,k)}) \neq 1,2 \\ 1 \text{ als } j_1((e)_{Pointer(u,k)}) = 1 \text{ en } Plusgoed(e,u,k,(e)_{Pointer(u,k \dot{-} 1)}) \\ 1 \text{ als } j_1((e)_{Pointer(u,k)}) = 2 \text{ en } Mingoed(e,u,k,(e)_{Pointer(u,k \dot{-} 1)}) \\ 0 \text{ anders} \end{cases}$$

$$Iederestap(e,u,0) = 0$$

$$Iederestap(e,u,k+1) = Stap(e,u,k+1) + Iederestap(e,u,k)$$

$$T(n,e,i,u) = \begin{cases} 1 \text{ als } Pointer(u,1) \neq 1 \\ 1 \text{ als } Geheugen(u,0) \neq i \\ 1 \text{ als } Iederestap(n,e,i,u,lengte(u)) \neq 0 \\ 1 \text{ als } Pointer(u,lengte(u)) \dot{-} lengte(e) \neq 0 \\ 0 \text{ anders} \end{cases}$$

$$U(u) = (Geheugen((u)_{lengte(u)}))_1$$

Omdat 0 staat voor waar en > 0 staat voor onwaar, is in de code hierboven "en" te schrijven als het optellen van de uitkomsten, "of" als het vermenigvuldigen van de uitkomsten, en "niet" als $1 \dot{-} n$.

Met $U(\mu u T(n,e,i,u))$ voeren we dus het programma gecodeerd door e uit, en krijgen we de uitkomst als dit programma op de gegeven invoer stopt. We zullen ook wel $\phi_e(n)$ schrijven voor $\lambda n.U(\mu u T(1,e,j(n,0),u))$, en analoog voor functies met meer argumenten. Verder zullen we de afkorting $T(e,n,u)$ voor $T(1,e,j(n,0),u)$ gebruiken. We schrijven $\phi_e(n)\downarrow$ voor $\exists u T(e,n,u)$, en $\phi_e(n)\uparrow$ voor $\neg\exists u T(e,n,u)$.

Iedere recursieve functie is dus berekenbaar, en iedere berekenbare functie is recursief.

1.4 De Smn stelling, het stopprobleem en de recursiestelling

Twee indices of twee verschillend opgebouwde recursieve functies kunnen dezelfde functie berekenen. Als f en g dezelfde functie berekenen, schrijven we $f \simeq g$. We noemen deze gelijkheid ook wel *Kleene-gelijkheid*. Er geldt dat $f \simeq g$ dan en slechts dan als $f(\overline{x})\downarrow$ precies als $g(\overline{x})\downarrow$, en wanneer ze stoppen, dat $f(\overline{x}) = g(\overline{x})$. Nu we over genoeg basiskennis over berekenbare functies beschikken, kunnen we beginnen met een aantal belangrijke eigenschappen te bewijzen.

De Smn-stelling heeft betrekking tot het gedeeltelijk invullen van argumenten voor recursieve functies, en is een belangrijke stelling binnen de berekenbaarheidstheorie. De Smn-stelling zegt dat er voor iedere n, m een (primitief) recursieve functie S_n^m bestaat, die gegeven een index van een recursieve functie met $n+m$ argumenten, de eerste n invult met gegeven waarden.

Stelling 1. *Er bestaan primitief recursieve functies S_n^m zo dat voor iedere $e, x_1, \ldots, x_m$ geldt dat*

$$\phi_{S_n^m(e,x_1\ldots,x_m)} \simeq \lambda y_1, \ldots, y_n.\phi_e(x_1, \ldots, x_m, y_1, \ldots, y_n)$$

Bewijs. De expliciete definitie van de geschikte S_n^m-functie is helaas tamelijk lang. Eerst definiëren we een primitief recurieve functie om, gegeven twee codes voor registermachineprogramma's, een code voor de compositie van de twee programma's terug te geven. Daarna definiëren we een functie om gegeven getallen i, k met $i < k$ een code te maken voor een programma om de waarden van registers $R_i, \ldots, R_{i+l}$ te verplaatsen naar de registers $R_k, \ldots, R_{k+l}$, en vervolgens een primitief recursieve functie om, gegeven getallen i, x een code te geven voor een programma dat register R_i waarde x geeft. De functie S_n^m stellen we op als

$$
\begin{aligned}
Plak'(l, 0, n, k) &= k \\
Plak'(l, i+1, n, k) &= j((l)_{n \dot{-} i}, Plak'(l, i, n, k)) \\
Plak(x, y) &= Plak'(x, lengte(x), lengte(x), y) \\
Verschuif'(n, s) &= j(j_1(n), j(j_1(j_2(n)), j(j_1(j_2(j_2(n))) + s, j(j_1(j_2(j_2(j_2(n)))) + s, 0)))) \\
Verschuif(l, s, 0) &= 0 \\
Verschuif(l, s, i+1) &= j(Verschuif'((l)_{i+1}, s), Verschuif(l, s, i)) \\
Compositie(f, g) &= Plak(f, Verschuif(g, lengte(f), lengte(g))) \\
Verplaats(i, k) &= j(j(2, j(i, j(3, j(2, 0)))), j(j(1, j(k, j(1, 0))), 0)) \\
Verplaatsrij(i, k, 0) &= 0 \\
Verplaatsrij(i, k, l+1) &= Compositie(Verplaats(i+l+1, k+l+1), Verplaatsrij(i, k, l)) \\
Zet(i, 0) &= 0 \\
Zet(i, k+1) &= Compositie(j(1, j(i, j(2, 0))), Zet(i, k)) \\
S_n^m(e, x_1, \ldots, x_m) &= Compositie(Verplaatsrij(1, m+1, n), \\
&\quad Compositie(Zet(1, x_1), \\
&\quad \cdots \\
&\quad Compositie(Zet(m, x_m), \\
&\quad e)) \cdots)
\end{aligned}
$$

□

De meest bekende stelling van de berekenbaarheidstheorie is waarschijnlijk wel het *stopprobleem* of *halting problem*. De vraag is of het door middel van een berekenbare functie mogelijk is om voor iedere index e en invoer n te bepalen of $\phi_e(n)$ stopt. We zullen zien dat dit niet kan.

Stelling 2. *Er bestaat geen recursieve functie F zo dat $F(e, n) = 0$ als $\phi_e(n)\downarrow$, en $F(e, n) = 1$ als $\phi_e(n)\uparrow$.*

Bewijs. We bewijzen dit door middel van diagonalisatie, op eenzelfde manier als in het bewijs van Cantor dat de kardinaliteit van de reële getallen groter is dan de kardinaliteit van de natuurlijke getallen. Stel er bestaat een recursieve functie $F(e, n)$ zoals hierboven beschreven. Dan is er ook een recursieve functie $D(e) = F(e, e)$, die de waarden op de diagonaal berekent. We vinden nu de functie $H(e) = \mu k(D(e, e) = 1)$ die geen index kan

hebben. Stel dat h een index is van H. Als $H(h)$ zou stoppen, dan is $1 = D(h) \simeq F(h,h)$, en stopt $H(h)$ dus niet, en we vinden een tegenspraak. Als $H(h)$ niet zou stoppen, dan is $0 = D(h) = F(h,h)$, en stopt $H(h)$ dus wel. Ook hier vinden we een tegenspraak, en we concluderen dat $F(e,n)$ geen index heeft, en dus niet recursief is. □

De volgende stelling zal in de rest van deze scriptie niet verder terugkomen, maar het is een mooie en nuttige stelling die we de lezer niet willen onthouden. Het bewijs is ontleent aan [9](11.2).

Stelling 3 (Recursiestelling)**.** *Voor iedere partieel recursieve functie F bestaat er een index e zo dat $\phi_e \simeq \phi_{F(e)}$*

Bewijs. Definieer $G(x,y) = \phi_{\phi_x(x)}(y)$, en vind met de Smn stelling een functie $H(x)$ zo dat $\phi_{H(x)} \simeq \lambda y.G(x,y) \simeq \lambda y.\phi_{\phi_x(x)}(y) \simeq \phi_{\phi_x(x)}$. Nu geldt voor iedere berekenbare functie F dat als e een index is van $\lambda x F(H(x))$, dat $\phi_{H(e)} \simeq \phi_{\phi_e(e)}$, en $\phi_e(e) = F(H(x))$ door keuze van e, dus $\phi_{H(e)} \simeq \phi_{F(H(e))}$, zoals gewenst. □

Hoofdstuk 2

Intuïtionistische logica

Beschouw de bewering "Er bestaan irrationale getallen a en b zo dat a^b rationaal is". Deze uitspraak kunnen we eenvoudig bewijzen door op te merken dat $\sqrt{2}^{\sqrt{2}}$ ofwel rationaal ofwel irrationaal is. Als het rationaal is, is onze uitspraak bewezen, en als het irrationaal is, dan is $\left(\sqrt{2}^{\sqrt{2}}\right)^{\sqrt{2}} = 2$ rationaal. De bewering is dus waar, maar helaas weten we niet of $a = \sqrt{2}, b = \sqrt{2}$ of $a = \sqrt{2}^{\sqrt{2}}, b = \sqrt{2}$ voldoet. Dit bewijs is daarom *niet constructief*. Een *constructief bewijs* is een bewijs dat niet alleen vertelt dat een zin waar is, maar dit soort extra informatie geeft.

Sommige wiskundigen zijn van mening dat niet-constructieve bewijzen geen echte bewijzen zijn, en accepteren dus alleen constructieve bewijzen. Een bekende wiskundige uit deze groep was L.E.J. Brouwer. Zijn filosofie over wanneer een bewijs constructief is, wordt het *intuïtionisme* genoemd. Binnen het intuïtionisme wordt een grote rol gespeeld door de BHK-interpretatie van Brouwer, Heyting en Kolmogorov over wat een constructief bewijs is:

- Een bewijs van $A \wedge B$ is een paar (a, b) waar a een bewijs is van A en b een bewijs is van B.
- Een bewijs van $A \vee B$ is een paar $(0, a)$ waar a een bewijs is van A, of een paar $(1, b)$ waar b een bewijs is van B.
- Een bewijs van $A \to B$ is een functie f die een bewijs van A omzet in een bewijs van B.
- Een bewijs van $\exists x A(x)$ is een paar (a, b) waar b een bewijs is van $A(a)$.
- Een bewijs van $\forall x A(x)$ is een functie f die iedere t omzet in een bewijs van $A(t)$.
- Een formule $\neg A$ interpreteren we als $A \to \bot$.

De BHK-interpretatie werkt goed samen met de meeste logische regels. Alleen de wet van de uitgesloten derde, $A \vee \neg A$, werkt hier niet goed mee samen. De wet van de uitgesloten derde is equivalent met de bewering $\neg\neg A \to A$, die we normaal altijd aannemen. Intuïtionistisch geldt deze wet niet, omdat we niet weten welke van A of $\neg A$ waar is. In het eerste voorbeeld dat we zagen ging het mis, doordat we er van uit gingen

dat als $\sqrt{2}^{\sqrt{2}}$ niet irrationaal is, dat het dan rationaal is.
De eisen die de BHK-interpretatie aan een bewijs stelt kunnen verschillen voor twee equivalente zinnen. Een bewijs van de uitspraak $x = y \vee x \neq y$ vereist ook kennis over of x en y gelijk zijn of niet, terwijl $\neg(x \neq y \wedge x = y)$ hier (klassiek) equivalent mee is, maar geen extra kennis nodig heeft volgens de BHK-interpretatie. Zo is er voor iedere zin A een equivalente zin zonder $\vee$ en $\exists$, en deze zinnen zijn wel intuïtionistisch bewijsbaar. Deze toekenning van equivalente zinnen heet de *Godel-Gentzen vertaling*. In deze scriptie zullen we hier niet verder op in gaan. De geïnteresseerde lezer raden we [10] van Troelstra en Van Dalen aan.

In dit werk zullen we gebruik maken van Heyting rekenkunde, een constructief systeem van axioma's voor de natuurlijke getallen. De axioma's zijn gelijk aan de axioma's van Peano rekenkunde, een klassieke axiomatisatie voor de natuurlijke getallen.

De taal van HA, die we aanduiden met $\mathcal{L}_{HA}$, bestaat uit de constante 0, een unair functiesymbool S voor de opvolgersfunctie en binaire functiesymbolen $+$ en $\cdot$ voor optellen en vermenigvuldigen. De axioma's van HA zijn

- $\forall n \neg(S(n) = 0)$
- $\forall nm(S(n) = S(m) \to n = m)$
- $\forall x(x + 0 = x)$
- $\forall xy(x + S(y) = S(x + y))$
- $\forall x(x \cdot 0 = 0)$
- $\forall xy(x \cdot S(y) = (x \cdot y) + y)$

En verder voor iedere formule A met vrije variabele x een inductie-axioma

$$(A(0) \wedge \forall n(A(n) \to A(S(n))) \to \forall n A(n)$$

Ondanks dat $A \vee \neg A$ niet voor iedere formule A bewijsbaar is, zijn er wel veel formules waarvoor $HA \vdash A \vee \neg A$ geldt.

Stelling 4. *Voor formules zonder onbegrensde kwantoren geldt $HA \vdash A \vee \neg A$.*

Bewijs. We bewijzen dit met inductie op formules. We behandelen eerst de inductiestappen.
Voor formules van de vorm $A \vee B$:

$$\dfrac{\dfrac{\dfrac{A^\dagger}{A \vee B}}{(A \vee B) \vee \neg(A \vee B)} \quad \dfrac{\dfrac{\dfrac{\dfrac{A^\dagger \quad \neg A^\dagger}{\bot} \quad \dfrac{B^\dagger \quad \neg B^\dagger}{\bot}}{\bot}}{\neg(A \vee B)}}{(A \vee B) \vee \neg(A \vee B)} \quad A \vee \neg A}{(A \vee B) \vee \neg(A \vee B)} \quad \dfrac{\dfrac{B^\dagger}{A \vee B}}{(A \vee B) \vee \neg(A \vee B)} \quad B \vee \neg B}{(A \vee B) \vee \neg(A \vee B)}$$

Voor formules van de vorm $A \wedge B$:

$$\dfrac{\dfrac{\dfrac{A^\dagger \quad B^\dagger}{A \wedge B}}{(A \wedge B) \vee \neg(A \wedge B)} \quad \dfrac{\dfrac{\dfrac{A \wedge B^\dagger}{A} \quad \neg A^\dagger}{\bot}}{\neg(A \wedge B)}}{(A \wedge B) \vee \neg(A \wedge B)} \; A \vee \neg A \qquad \dfrac{\dfrac{\dfrac{A \wedge B^\dagger}{B} \quad \neg B^\dagger}{\bot}}{\neg(A \wedge B)}}{(A \wedge B) \vee \neg(A \wedge B)} \; B \vee \neg B$$

Voor formules van de vorm $A \to B$:

$$\dfrac{\dfrac{\dfrac{A \to B^\dagger \quad A^\dagger}{B} \quad \neg B^\dagger}{\bot}}{\neg(A \to B)} \quad \dfrac{\dfrac{B^\dagger}{A \to B}}{(A \to B) \vee \neg(A \to B)} \; B \vee \neg B \qquad \dfrac{\dfrac{\dfrac{\neg A^\dagger \quad A^\dagger}{\bot}}{B}}{A \to B}}{(A \to B) \vee \neg(A \to B)} \; A \vee \neg A$$

Voor formules van de vorm $\exists x < t A(x)$, waar t niet in A voorkomt, geven we het bewijs zonder bewijsboom, omdat de formules allemaal erg veel ruimte innemen. We bewijzen dit met inductie op t. Duidelijk geldt $\neg\exists x < 0 A(x)$. Als $A(n)$ dan geldt $\exists x < n+1 A(x)$, en als $\neg A(n)$ en $\neg\exists x < n A(x)$ dan volgt $\neg\exists x < n+1 A(x)$. Met de inductiehypothese $A(n) \vee \neg A(n)$ volgt $\exists x < n+1 A(x) \vee \neg\exists x < n+1 A(x)$, en met het inductieaxiomaschema concluderen we $\exists x < t A(x) \vee \neg\exists x < t A(x)$ voor alle termen t.

Voor formules van de vorm $\forall x < t A(x)$, waar t niet in A voorkomt, gebruiken we inductie naar t. Ook hier zou een bewijsboom helaas te veel ruimte innemen. Het bewijs gaat analoog met het geval $\exists x < t A(x)$.

Basisstap: Iedere atomaire formule in Heyting rekenkunde is van de vorm $a = b$. Om $a = b \vee \neg a = b$ te bewijzen maken we gebruik van dubbele inductie. Eerst bewijzen $a = 0 \vee \neg a = 0$ met inductie naar a, vervolgens bewijzen we $a = S(b) \vee \neg a = S(b)$ met inductie naar a, door gebruik te maken van het axioma $S(x) = S(y) \to x = y$. Nu volgt met inductie naar b dat $a = b \vee \neg a = b$. □

De taal en axioma's van HA worden soms ook anders gekozen. Zo is het mogelijk om symbolen voor (alle) primitief recursieve functies en/of predicaten toe te voegen, samen met axioma's om deze symbolen te definiëren. Dit kunnen we doen zonder echt eigenschappen van HA te veranderen, omdat alle primitief-recursieve functies en predicaten te vervangen zijn door termen en formules zonder onbegrensde kwantoren. Wanneer we in deze scriptie een primitief-recursieve functie of primitief-recursief predicaat gebruiken

in een HA-formule mag dit worden beschouwd als afkorting van de bijbehorende term of formule in enkel de eerder gegeven taal van HA.

Een eigenschap die we volgens de BHK-interpretatie zouden verwachten van een constructieve theorie T, is dat als $T \vdash A \vee B$, dat dan $T \vdash A$ of $T \vdash B$, en dat als $T \vdash \exists x A(x)$, dat er dan een x is zo dat $T \vdash A(x)$. Dit noemen we de disjunctie-eigenschap en de existentie-eigenschap. Deze eigenschappen zullen we in het volgende hoofdstuk voor HA bewijzen.

Hoofdstuk 3

Realiseerbaarheid

De BHK-interpretatie van de intuïtionistische logica verwijst naar functies die een bewijs van een formule A omzetten in een bewijs van een formule B, en functies die een getal n kunnen omzetten in een bewijs van een formule $A(n)$. In de BHK-interpretatie wordt niet gespecificeerd wat voor functies dit zijn, maar omdat constructieve bewijzen bedoelt zijn om door mensen te worden geïnterpreteerd, ligt het voor de hand om te eisen dat deze functies voor mensen te berekenen moeten zijn. Uit de Church-Turing hypothese volgt dat deze functies dus recursieve functies moeten zijn. Door middel van de eerder gedefinieerde indices voor berekenbare functies en de paringsfunctie wordt het mogelijk om de extra informatie die een intuïtionistisch bewijs moet leveren te coderen in getallen. Deze techniek vormt de grondslag van realiseerbaarheid. Realiseerbaarheid werd als eerste geformuleerd door Stephen Kleene in [3] in de vorm die we hier als eerst omschrijven, en is sinds toen uitgegroeid tot een veelzijdige verzameling technieken die bruikbaar zijn in meerdere gebieden binnen de logica. We zullen eerst Kleene's originele realiseerbaarheid bespreken, en daarna twee variaties hier op.

Wanneer een getal n de extra informatie voor een bewijs van een zin A codeert, zeggen we dat A wordt *gerealiseerd* door n. We geven dit ook wel aan met $n \Vdash A$. We schrijven ook wel $\Vdash A$ als A wordt gerealiseerd maar de realisator er niet toe doet.

- Een atomaire zin heeft geen extra informatie nodig, dus voor atomaire A stellen we $n \Vdash A$ precies wanneer A waar is.
- Een getal n realiseert $A \wedge B$ precies wanneer $n = j(a, b)$ zodat $a \Vdash A$ en $b \Vdash B$.
- Een getal n realiseert $A \vee B$ precies wanneer $n = j(0, a)$ en $a \Vdash A$, of $n = j(1, b)$ en $b \Vdash B$.
- Een getal e realiseert $A \to B$ precies wanneer als $a \Vdash A$, dan $\phi_e(a)\downarrow$ en $\phi_e(a) \Vdash B$.
- Een getal n realiseert $\exists x A(x)$ precies wanneer $n = j(a, b)$ en $b \Vdash A[x/a]$.
- Een getal e realiseert $\forall x A(x)$ precies wanneer voor iedere n geldt dat $\phi_e(n)\downarrow$ en $\phi_e(n) \Vdash A[x/n]$.

Ook hier interpreteren we $\neg A$ als $A \to \bot$.

Voorbeeld 13. *$\bot \to A$ wordt gerealiseerd door ieder getal. $\bot$ is atomair en onwaar, en wordt dus niet gerealiseerd. Ieder algoritme geeft dus, wanneer het als invoer een realisator van $\bot$ krijgt, een realisator van A terug.*

Voorbeeld 14. *Klassiek gezien is iedere zin A realiseerbaar of niet realiseerbaar. Als A niet realiseerbaar is, wordt $\neg A$ door elk getal gerealiseerd. Klassiek gezien wordt $A \vee \neg A$ dus gerealiseerd, ondanks dat we niet noodzakelijk weten welke van A en $\neg A$ gerealiseerd wordt.*

Voorbeeld 15. *Als A wordt gerealiseerd, wordt $\neg\neg A$ ook gerealiseerd. Immers is $\neg\neg A$ een afkorting voor $(A \to \bot) \to \bot$, en als we een realisator n van A hebben, kunnen we iedere realisator e van $A \to \bot$ naar een realisator van $\bot$ sturen door $\phi_e(n)$ te berekenen.*

Omdat de realisator geen bewijs levert voor atomaire zinnen, is het vinden van een realisator voor een bekende ware zin vaak makkelijk, zoals we zien in dit voorbeeld.

Voorbeeld 16. *De laatste stelling van Fermat, $\forall n \forall a \forall b \forall c \neg(n > 2 \wedge a^n + b^n = c^n)$, wordt gerealiseerd door iedere index van $\lambda n \lambda a \lambda b \lambda c.j(0,0)$. Voorbeeld ontleend aan [4].*

We kunnen realiseerbaarheid gebruiken om uitspraken te bewijzen over wat Heyting rekenkunde wel en niet kan bewijzen. Daarvoor zullen we een aantal verbanden leggen tussen realiseerbaarheid en bewijsbaarheid in HA. Het volgende bewijs is oorspronkelijk van Nelson[5], maar is hier aangepast voor inductie op bewijsbomen.

Lemma 1. *De axioma's van HA worden gerealiseerd.*

Bewijs. Alle axioma's behalve het inductie-axiomaschema zijn van de vorm $\forall x A(x)$ en waar op de natuurlijke getallen, en worden dus onder andere gerealiseerd door indices van de functie $\lambda x.0$. We hoeven dus alleen nog aan te tonen dat de inductie-axioma's worden gerealiseerd. Deze zijn van de vorm $A(0) \wedge \forall n(A(n) \to A(n+1)) \to \forall n A(n)$. Stel $k = j(a,b)$ realiseert $A(0) \wedge \forall n(A(n) \to A(n+1))$, dan wordt $\forall n A(n)$ gerealiseerd met de functie F gegeven door primitieve recursie:

$$F(a,b,0) = a$$
$$F(a,b,n+1) = \phi_{\phi_b(n)}(F(a,b,n))$$

Immers kunnen we voor iedere n een realisator vinden van $A(n)$ door te beginnen met een realisator van $A(0)$ en telkens van een realisator van $A(k)$ een realisator van $A(k+1)$ te maken, net zo lang tot we een realisator van $A(n)$ hebben gevonden. Als f een index is voor F vinden we met de Smn-stelling een index e voor de functie $\lambda k S_1^2(f, j_1(k), j_2(k))$, en e realiseert de inductie-axioma's. □

Om nu te bewijzen dat we vanuit een bewijs in HA een realisator kunnen maken, hebben we een lemma nodig met een erg technische inductiehypothese. Dit is onder andere doordat in een bewijsboom verschillende vrije variabelen kunnen voorkomen, maar we realiseerbaarheid alleen gedefinieerd hebben voor zinnen zonder vrije variabelen.

Lemma 2. *Voor iedere bewijsboom B met conclusie A bestaat een (primitief) recursieve functie F_B zo dat, als in de open aannamen en conclusie van B de vrije variabelen*

$v_1, \ldots, v_m$ worden gebruikt, dat $F_B(n_1, \ldots, n_k)$ een index is zo dat, als $p_1, \ldots, p_m$ realisators zijn van de ongemarkeerde aannamen in B met de vrije variabelen ingevuld met $n_1, \ldots, n_k$, dat dan $\phi_{F_B(n_1,\ldots,n_k)}(p_1, \ldots, p_m)\downarrow$ en $\phi_{F_B(n_1,\ldots,n_k)}(p_1, \ldots, p_m)$ een realisator van $A[n_1/v_1, \cdots, n_k/v_k]$.

Bewijs. We construeren F_B met inductie op bewijsbomen. We schrijven $F \Vdash A$ om aan te duiden dat F een functie is die voldoet aan de hierboven gegeven voorwaarde voor een bewijsboom met A als conclusie. Om de juiste argumenten aan de juiste functies te geven zullen we indexsets $I, J \subset \mathbb{N}$ voor realisators gebruiken, en indexsets $K, L \subset \mathbb{N}$ voor vrije variabelen.

Ass Voor de assumptiebomen zoeken we een primitief recursieve functie die, gegeven de invulling van de vrije variabelen, een functie geeft die, gegeven een realisator van de aanname, een realisator van de conclusie teruggeeft. Bij assumptiebomen is de aanname gelijk aan de conclusie, dus als e een index is van $\lambda x.x$, dan is $\lambda v_1 \cdots v_k.e$ een geschikte functie.

$\wedge$I Gegeven $F_A \Vdash A$, $F_B \Vdash B$. Kies een index e van

$$\lambda a b p_1 \cdots p_N.j(\phi_a(p_{I_1}, \ldots, p_{I_i}), \phi_b(p_{J_1}, \ldots, p_{J_j}))$$

We vinden nu

$$S^2_N(e, F_A(v_1, \ldots, v_m), F_B(v_{K_1}, \ldots, v_{K_k})) \Vdash A \wedge B$$

$\wedge$E Gegeven $F \Vdash A \wedge B$. Kies een index e van

$$\lambda e p_1 \cdots p_N.j_1(\phi_e(p_1 \ldots, p_N))$$

We vinden nu

$$S^1_N(e, F(v_1 \ldots v_m)) \Vdash A$$

Analoog vinden we eenzelfde functie voor de andere kant van de conjunctie door j_1 met j_2 te vervangen.

$\vee$I Gegeven $F \Vdash A$. Kies een index e van

$$\lambda a p_1 \cdots p_N.j(0, \phi_a(p_1, \ldots, p_N))$$

We vinden nu

$$S^1_N(e, F(v_1, \ldots, v_l)) \Vdash A \vee B$$

Analoog vinden we eenzelfde functie voor de andere kant van de disjunctie door 0 met 1 te vervangen.

$\vee$E Gegeven $F_{A\vee B} \Vdash A \vee B$, $F_A \Vdash C$, $F_B \Vdash C$, waar de bewijsboom van F_A als eerste ongemarkeerde aanname A heeft, en de bewijsboom van F_B als eerste ongemarkeerde aanname B heeft. Kies een index e van

$$\lambda d a b p_1 \cdots p_N. \begin{cases} \phi_a(j_2(\phi_d(p_1, \ldots, p_n)), p_{I_1}, \ldots, p_{I_i}) & \text{als } j_1(\phi_d(p_1, \ldots, p_n)) = 0 \\ \phi_b(j_2(\phi_d(p_1, \ldots, p_n), p_{J_1}, \ldots, p_{J_j}) & \text{anders} \end{cases}$$

Hier definiëren we conditionele keuze anders dan eerst. Voorheen moesten alle argumenten gedefinieerd zijn voordat door kon worden gegaan met de berekening. In dit geval definiëren we de conditionele keuze met behulp van de functie $\lambda kef\overline{x}.U(\mu y.T(e,\overline{x},k\cdot y)\vee T(f,\overline{x},(1-k)\cdot y))$, zodat alleen de relevante berekening hoeft te eindigen. Nu geldt

$$S_N^3(e,F_{A\vee B}(v_1,\ldots,v_m),F_A(v_{K_1},\ldots,v_{K_k}),F_B(v_{L_1},\ldots,v_{L_l}))\Vdash C$$

$\to$I Gegeven $F_A\Vdash A$, waar de bewijsboom van F_A als laatste ongemarkeerde aanname B heeft. Kies een index e van $\lambda fp_1\cdots p_N.S_1^N(f,p_1,\ldots,p_N)$. We vinden nu

$$S_N^1(e,F_A(v_1,\ldots,v_l))\Vdash B\to A$$

$\to$E Gegeven $F_A\Vdash A$, $F_{A\to B}\Vdash A\to B$. Kies een index e van

$$\lambda abp_1\cdots p_N\phi_{\phi_b(p_{J_1},\ldots,p_{J_j})}(\phi_a(p_{I_1},\ldots,p_{I_i}))$$

We vinden nu

$$S_N^2(e,F_A(v_1,\ldots,v_m),F_{A\to B}(v_{K_1},\ldots,v_{K_k}))\Vdash B$$

$\forall$I Gegeven $F_A\Vdash A(x)$. We vinden

$$\lambda v_1\cdots v_k\lambda p_1\cdots p_N\lambda x.\phi_{F_A(x,v_1,\ldots,v_m)}(p_1,\ldots,p_N)\Vdash\forall xA(x)$$

$\forall$E Gegeven $F_A\Vdash\forall xA(x)$. Als x correspondeert met v_i, kies een index e van

$$\lambda fcp_1\cdots p_N\phi_{\phi_f(p_1,\ldots,p_N)}(c)$$

We vinden nu

$$S_n^2(e,F_A(v_1,\ldots,v_m),v_{m+1})\Vdash A(x)$$

$\exists$I Gegeven $F_A\Vdash A[x/t]$. Omdat iedere term in HA is opgebouwd door middel van constantes, variabelen, optellen en aftrekken, is het mogelijk een primitief recursieve functie $G(v_1,\ldots,v_k)$ te vinden zo dat $G(n_1,\ldots,n_k)=t[v_1/n_1,\ldots,v_m/n_m]$ voor iedere $n_1,\ldots,n_k\in\mathbb{N}$. Kies e een index van $\lambda tep_1\cdots p_n.j(t,\phi_e(p_1,\ldots,p_n))$. We vinden nu

$$S_n^2(e,G(v_1,\ldots,v_m),F_A(v_1,\ldots,v_m))\Vdash\exists xA(X)$$

$\exists$E Gegeven $F_A\Vdash\exists xA(x)$, $F_B\Vdash B$, waar de bewijsboom van F_B ongemarkeerde aanname $A(n)$ heeft, en n niet vrij is in B.

$$\lambda\overline{v}\lambda\overline{p}.\phi_{F_B(\overline{v},j_2(\phi_{F_A(\overline{v})}(\overline{p})))}(\overline{p},j_1(\phi_{F_A(\overline{v})}(\overline{p})))\Vdash B$$

□

Stelling 5. *Als $\Gamma\vdash A$ en alle zinnen in Γ realiseerbaar zijn, is A ook realiseerbaar.*

Bewijs. Laat B een bewijsboom zijn voor een bewijs van A met alle open aannamen in Γ. Vind F_B zoals hierboven. Omdat alleen de open aannamen nog realisators nodig hebben, en we weten dat alle open aannamen realiseerbaar zijn, kunnen we voor iedere $n_1, \ldots, n_k \in \mathbb{N}$ realisators $p_1, \ldots, p_n \in \mathbb{N}$ van de open aannamen vinden. We vinden $\phi_{F_B(n_1,\ldots,n_l)}(p_1, \ldots, p_n)$ als realisator van A, en de bewering is bewezen. □

Dit verband gaat niet de andere kant op: er zijn zinnen die wel realiseerbaar zijn, maar niet bewijsbaar in HA, zoals de Gödelzin uit het bewijs van de onvolledigheidsstellingen. Er zijn ook realiseerbare zinnen die niet bewijsbaar zijn in HA omdat ze klassiek onwaar zijn, zoals de volgende:

Stelling 6. *De zin* $\neg\forall e(\exists n.T(e,e,n) \vee \neg\exists n.T(e,e,n))$ *wordt gerealiseerd.*

Bewijs. Stel dat $\forall e(\exists n.T(e,e,n) \vee \neg\exists n.T(e,e,n))$ zou worden gerealiseerd door een getal f. Het volgt dat de totale functie $G(e) = j_1(\phi_f(e))$ recursief is, maar dat $G(e) = 0$ precies als $\phi_e(e)\downarrow$ en $G(e) = 1$ als $\phi_e(e)\uparrow$. Dit is in tegenspraak met het stopprobleem, dus $\forall e(\exists n.T(e,e,n) \vee \neg\exists n.T(e,e,n))$ wordt niet gerealiseerd. Omdat het geen realisators heeft, stuurt elke berekenbare functie iedere realisator van $\forall e(\exists n.T(e,e,n) \vee \neg\exists n.T(e,e,n))$ naar een realisator van $\bot$, en we concluderen dat $\neg\forall e(\exists n.T(e,e,n) \vee \neg\exists n.T(e,e,n))$ wordt gerealiseerd. □

Omdat $\forall e(\exists n.T(e,e,n) \vee \neg\exists n.T(e,e,n))$ niet wordt gerealiseerd, wordt het dus ook niet bewezen door HA. Zinnen van de vorm $\forall xA \vee \neg A$ zijn dus niet altijd bewijsbaar in HA, en zien we dat HA de wet van uitgesloten derde niet bewijst.

Als laatste beschouwen we de *rekenkundige vorm van Church's thesis*, een axiomaschema dat voor iedere HA-formule A zegt

$$\forall n\exists mA(n,m) \to \exists e\forall n\exists u(T(e,n,u) \wedge A(n,U(u))$$

We korten dit ook wel af met CT_0. Church's thesis krijgt zijn naam van de Church-Turing hypothese, maar de Church's thesis uit de constructieve logica is veel sterker: het zegt dat iedere definieerbare functie berekenbaar is. Dit is in tegenspraak met de wet van uitgesloten derde. Stel immers dat we de axioma's van HA, samen met CT_0 en de wet van uitgesloten derde aannemen. Dan kunnen we door middel van de wet van uitgesloten derde de zin $\exists x(x = 0 \to \exists yT(n,n,y)) \wedge (x = 1 \to \neg\exists yT(n,n,y)) \wedge \neg x > 1$ bewijzen, maar uit CT_0 zou volgen dat er een recursieve functie bestaat die het haltingprobleem oplost. We zullen daarentegen zien dat CT_0 aannemen samen met de axioma's van HA wel een consistente theorie oplevert. Dit bewijzen we door aan te tonen dat CT_0 realiseerbaar is. Realiseerbaarheid in HA vormt dan een model van $HA + CT_0$, en we vinden een consistente theorie voor natuurlijke getallen die afwijkt van de Peano axioma's voor klassieke rekenkunde.

Als i een index is van j_1, en k een index is van

$$\lambda en.j(\mu u.T(e,n,u), j(0, j_2(\phi_e(n))))$$

dan is iedere index van

$$\lambda e.j(Compositie(e,i), S_1^1(k,e))$$

een realisator van CT_0. Omdat CT_0 realiseerbaar is, weten we dat $HA + CT_0$ consistent is. Desondanks wordt CT_0 niet bewezen door HA, want $HA + CT_0 \vdash \neg\forall x(\exists y T(x,x,y) \vee \neg\exists y T(x,x,y))$, wat een conflict geeft met de wet van uitgesloten derde, terwijl de klassieke variant van HA, Peano rekenkunde, ook consistent is. Op deze manier is het dus mogelijk om HA op een interessante manier uit te breiden, op een manier die in klassieke logica niet mogelijk is.

3.1 Geformaliseerde realiseerbaarheid

Realiseerbaarheid is een erg flexibele techniek, en is op verschillende manieren te variëren. We bespreken nu *geformaliseerde realiseerbaarheid*. Dat is realiseerbaarheid, uitgedrukt in formele taal. We zullen in deze en de volgende paragraaf een interessante toepassing van geformaliseerde realiseerbaarheid zien.

In dit geval formaliseren we realiseerbaarheid dus in Heyting rekenkunde. Dit doen we door iedere HA-formule A een andere HA-formule $x\,\mathbf{r}\,A$ toe te kennen, waar x niet vrij is in A. We bouwen de formule $x\,\mathbf{r}\,A$ op aan de hand van de structuur van A:

- Voor atomaire formules definiëren we $x\,\mathbf{r}\,(t = s) :=\ x = 0 \wedge t = s$.
- Voor conjuncties definiëren we $x\,\mathbf{r}\,A \wedge B := j_1(x)\,\mathbf{r}\,A \wedge j_2(x)\,\mathbf{r}\,B$.
- Voor disjuncties definiëren we $x\mathbf{r}\,A \vee B := (j_1(x){=}0 \wedge j_2(x)\mathbf{r}\,A) \vee (j_1(x){=}1 \wedge j_2(x)\mathbf{r}\,B)$.
- Voor implicaties definiëren we $x\,\mathbf{r}\,A \to B := \forall y(y\,\mathbf{r}\,A \to \exists u(T(x,y,u) \wedge U(u)\,\mathbf{r}\,B))$. Hier mag u niet vrij zijn in A.
- Bij universele kwantoren definiëren we $x\,\mathbf{r}\,\forall y A := \forall y \exists u(T(x,y,u) \wedge U(u)\,\mathbf{r}\,A)$. Ook hier mag u niet vrij zijn in A.
- Bij existentiële kwantoren definiëren we $x\,\mathbf{r}\,\exists y A := j_2(x)\,\mathbf{r}\,A[y/j_1(x)]$.

Let op dat de notatie hier misschien een verkeerde suggestie wekt: behalve voor atomaire formules komt A nergens letterlijk voor in $x\,\mathbf{r}\,A$. Er wordt geen functie op A toegepast. We gebruiken $x\,\mathbf{r}\,A$ enkel als afkorting voor een veel langere logische formule.

Net zoals iedere HA-zin A met $HA \vdash A$ realiseerbaar is, is er ook voor iedere HA-zin A met $HA \vdash A$ een getal n zodat $HA \vdash n\,\mathbf{r}\,A$. Dit bewijs lijkt erg op het bewijs voor gewone realiseerbaarheid, en zullen we dus niet opnieuw behandelen. Er zijn een aantal belangrijke verschillen tussen realiseerbaarheid en geformaliseerde realiseerbaarheid, zoals dat we over gewone realiseerbaarheid klassiek konden redeneren, terwijl dit bij geformaliseerde realiseerbaarheid niet zomaar kan.

3.2 Existentie- en disjunctie-eigenschap van HA

Zoals al eerder uitgelegd, verwachtten we dat HA de disjunctie-eigenschap en existentie-eigenschap heeft. Dat wil zeggen: als $HA \vdash A \vee B$, dan $HA \vdash A$ of $HA \vdash B$, en als $HA \vdash \exists x A(x)$ dan is er een n zodat $HA \vdash A[x/n]$.

De existentie-eigenschap voor HA werd voor het eerst bewezen door Ronald Harrop in [2]. We geven hier een variant van het bewijs uit [10](p.243). Hiervoor maken we

ook weer een kleine verandering aan de realiseerbaarheid die we gebruiken. Deze variant op geformaliseerde realiseerbaarheid heet *q-realiseerbaarheid*. We nemen bijna de hele definitie van geformaliseerde realiseerbaarheid over voor q-realiseerbaarheid, behalve de definitie voor implicaties. Voor implicaties definiëren we

$$x \,\mathbf{q}\, A \to B := \forall y(y \,\mathbf{r}\, A \to \exists u(T(x,y,u) \wedge U(u) \,\mathbf{r}\, B)) \wedge A \to B$$

Ook voor q-realiseerbaarheid geldt nog steeds dat als $HA \vdash A$ er een n bestaat zo dat $HA \vdash n \,\mathbf{q}\, A$. We zullen zien dat deze kleine aanpassing genoeg is om te kunnen bewijzen dat de q-realiseerbare formules precies de ware formules zijn:

Stelling 7. *Voor iedere HA-formule A geldt $HA \vdash \exists x(x \,\boldsymbol{q}\, A) \to A$.*

Bewijs. Dit bewijzen we met inductie naar de complexiteit van A. We behandelen slechts een deel van de inductiestappen.

Voor de axioma's van HA is de uitspraak $\exists x(x \,\mathbf{r}\, A) \to A$ triviaal waar.

Voor atomaire formules concluderen we het gewenste als volgt:

$$\dfrac{\dfrac{\dfrac{\exists x(x \,\mathbf{q}\, t = s)^\dagger}{\exists x(x = 0 \wedge t = s)} \quad \dfrac{x = 0 \wedge t = s^\dagger}{t = s}}{t = s}}{\exists x(x \,\mathbf{q}\, t = s) \to t = s}$$

Voor formules van de vorm $\exists y A$ merken we op dat $\exists x(x \,\mathbf{q}\, \exists y A)$ per definitie gelijk is aan $\exists x(j_2(x) \,\mathbf{q}\, A[y/j_1(x)])$. We vinden nu het volgende bewijs:

$$\dfrac{\dfrac{\exists x(j_2(x) \,\mathbf{q}\, A[y/j_1(x)])^\dagger \qquad \dfrac{\dfrac{j_2(x) \,\mathbf{q}\, A[y/j_1(x)]^\dagger}{\exists t(t \,\mathbf{q}\, A[y/j_1(x)])} \qquad \exists t(t \,\mathbf{q}\, A[y/j_1(x)]) \to A[y/j_1(x)]}{A[y/j_1(x)]}}{\exists y A}}{\exists x(x \,\mathbf{q}\, \exists y A) \to \exists y A}$$

Voor formules van de vorm $\forall y A$ merken we op dat $\exists x(x \,\mathbf{q}\, \forall y A)^\dagger$ per definitie gelijk is aan $\exists x(\forall y(\exists u(T(x,y,u) \wedge U(u) \,\mathbf{q}\, A)))$. We vinden nu:

$$\dfrac{\dfrac{\exists x(\forall y(\exists u(T(x,y,u) \wedge U(u) \,\mathbf{q}\, A)))^\dagger \qquad \dfrac{\dfrac{\dfrac{T(x,y,u) \wedge U(u) \,\mathbf{q}\, A^\dagger}{U(u) \,\mathbf{q}\, A}}{\exists t(t \,\mathbf{q}\, A)} \qquad \dfrac{\forall y(\exists u(T(x,y,u) \wedge U(u) \,\mathbf{q}\, A))^\dagger}{\exists u(T(x,y,u) \wedge U(u) \,\mathbf{q}\, A)}}{\exists t(t \,\mathbf{q}\, A)}}{\exists t(t \,\mathbf{q}\, A)} \qquad \exists t(t \,\mathbf{q}\, A) \to A}{\dfrac{A}{\forall y A}}}{\exists x(x \,\mathbf{q}\, \forall y A) \to \forall y A}$$

□

De existentie- en disjunctie-eigenschap volgen nu vanzelf:

Stelling 8. *HA heeft de existentie-eigenschap: Als* $HA \vdash \exists x A$*, dan is er een* $n \in \mathbb{N}$ *zo dat* $HA \vdash A[x/n]$

Bewijs. Uit $HA \vdash \exists y A$ volgt dat er een n bestaat zodat $HA \vdash n \,\mathbf{q}\, \exists x A(x)$, en per definitie $HA \vdash j_1(n) \,\mathbf{q}\, A[j_2(n)/x]$. Omdat $A[j_2(n)/x]$ wordt ge-q-realiseerd, volgt dat $HA \vdash A[j_2(n)/x]$. □

Stelling 9. *HA heeft de disjunctie-eigenschap: Als* $HA \vdash A \vee B$*, dan* $HA \vdash A$ *of* $HA \vdash B$*.*

Bewijs. Uit $HA \vdash A \vee B$ volgt dat er een n bestaat zodat $HA \vdash n \,\mathbf{q}\, A \vee B$, oftewel $HA \vdash (j_1(n) = 0 \wedge j_2(n) \,\mathbf{q}\, A) \vee (j_1(n) = 1 \wedge j_2(n) \,\mathbf{q}\, B)$. Uit de waarde van n concluderen we $HA \vdash (j_1(n) = 0 \wedge j_2(n) \,\mathbf{q}\, A) \vee \bot$ of $HA \vdash \bot \vee (j_1(n) = 1 \wedge j_2(n) \,\mathbf{q}\, B)$. We vinden nu dat $HA \vdash \exists x x \,\mathbf{q}\, A$ of $HA \vdash \exists x x \,\mathbf{q}\, B$, en dus $HA \vdash A$ of $HA \vdash B$ zoals gewenst. □

Bibliografie

[1] A. Church. *An unsolvable problem of elementary number theory.* American journal of mathematics (1936): 345-363.

[2] R. Harrop. *On disjunctions and existential statements in intuitionistic systems of logic.* Mathematische Annalen, 132:347-361, 1956

[3] S.C. Kleene. *On the interpretation of intuitionistic number theory.* The Journal of Symbolic Logic 10.04 (1945): 109-124.

[4] A. Miquel. 2014. *Computational interpretation of proofs: An introduction to realizability.* Geraadpleegd 9 oktober 2015. `https://www.fing.edu.uy/~amiquel/cirm14-1.pdf`

[5] D. Nelson. *Recursive functions and intuitionistic number theory.* Transactions of the American Mathematical Society (1947): 307-368.

[6] J. van Oosten. *Basic Computability Theory*, 2013.

[7] J. van Oosten. *Gödel's Incompleteness Theorems*, 2015.

[8] J. van Oosten. 2000. *Realizability: An Historical Essay.* Geraadpleegd 20 december 2015. `www.staff.science.uu.nl/~ooste110/realizability/history.ps.gz`

[9] H. Rogers. *Theory of recursive functions and effective computability.* Vol. 126. New York: McGraw-Hill, 1967.

[10] A.S. Troelstra, D. van Dalen. *Constructivism in Mathematics*, Elsevier Science Publishers B.V., 1988.

[11] A.S. Troelstra. *Notions of realizability for intuitionistic arithmetic and intuitionistic arithmetic in all finite types*, in: Fenstad (1971) 369-405

[12] A.M. Turing. *On computable numbers, with an application to the Entscheidungsproblem.* J. of Math 58.345-363 (1936).