SJÄLVSTÄNDIGA ARBETEN I MATEMATIK

MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET

Vad skulle vi göra utan Turingmaskiner

av

Helena Collert

2026 - No K15

Vad skulle vi göra utan Turingmaskiner

Helena Collert

Självständigt arbete i matematik 15 högskolepoäng, grundnivå

Handledare: Peter LeFanu Lumsdaine

2026

Sammanfattning

Trots att vi använt oss av beräkningar i tusentals år är det en ung disciplin inom matematiken att förstå vad det egentligen innebär att något är beräkningsbart. Är $2 \cdot 4 = 8$? Självklart! skulle nog alla utom de allra minsta svara. Men det är inte självklart. Faktum är att det finns få självklara beräkningar. En beräkning som faktiskt alla har kunnat enas om att den är självklar är att efter 0 kommer 1 och efter 1 kommer 2.. Eller egentligen inte det heller. Det man enats om är att efter ett tal vilket som helst kommer ett tal som är 1 mer än just det talet. Den beräkningen är en av tre s.k. basfunktioner som ligger till grund för alla beräkningar i hela världen.

Det här arbetet ska försöka redogöra för hur *Church-Turing*s hypotes, själva definitionen av vad som är beräkningsbart, växte fram. Mitt huvudsakliga uppdrag är att presentera EN form av beräkningsbarhet, teorin om *Partiella Rekursiva funktioner*. Men vi ska också titta på λ-*kalkylen* som ligger till grund för flera, fortfarande aktuella, dataspråk. Och för den mer handfasta begripligheten av hur beräkningsbarhet är uppbyggd ska vi studera *URM*, en förenklad variant av den moderna datorns föregångare, *Turingmaskinen*. Jag kommer också redogöra för två av bevisen för att det s.k. *Avgörbarhetsproblemet* saknar lösning.

Abstract

Even though we have been using calculations for thousands of years, it is a young discipline in mathematics to understand what it actually means to be computable. Is $2 \cdot 4 = 8$? Of course! everyone except the very youngest would probably answer. But it is not obvious. The fact is that there are few selfevident calculations. One calculation that everyone has been able to agree on is that after 0 comes 1 and after 1 comes $2 \ldots$ Or not really that one either. What is agreed upon is that after a number comes a number that is 1 more than that particular number. That definition of a calculation is one of three so called Base functions forming the basis of all calculations in the entire world.. This work will try to explain how *Church-Turing* thesis, the very definition of what is computable emerged. My main task is to present ONE form of computability, the theory of *Partial Recursive functions*. But we will also look at the λ -calculus, which is the basis for several, still current, computer languages. And for the more tangible understanding of how

computability is built up we will study the *URM*, a simplified version of the modern computer's predecessor, the *Turing machine*. I will also present two of the proofs showing that the so called *Decision problem* has no solution.

Tack till

Jag vill rikta ett varmt tack till min handledare Peter LeFanu Lumsdaine och Annemarie Luger, huvudlärare på Matematiska institutionen för oändligt tålamod, vänlighet och varsamhet vilket, förutom fantastisk matematisk vägledning, möjliggjort för mig att slutföra detta arbete.

Innehåll

1 Introduktion

David Hilbert (1862–1943) var en tysk matematiker och logiker, som ägnade en stor del av sitt liv åt att försöka bevisa att matematiken är konsekvent och att det går att bygga ett fundament av axiom som gör motsägelser inom det matematiska systemet omöjliga, rent logiskt. Han skriver själv 1917 i sin skrift *Axiomatisches Denken s. 411*: ”Det grundläggande kravet på Axiomteori måste gå längre [än att bara undvika kända logiska paradoxer när de dyker upp] nämligen att visa att motsättningar, inom varje kunskapsområde, är helt omöjliga baserat på ett etablerat axiomsystem.” (Hilbert 1917)

Hans tankar nedtecknades runt 1920 i den omfattande skriften *Hilberts Program* författad utifrån de 23 *Hilbertproblemen* rörande olösta matematiska problem han själv presenterat år 1900. Utifrån fråga nr 2 formulerade han ytterligare en tes tillsammans med en av sina elever och sedermera kollega, den tyska matematikern *Wilhelm Ackermann* (1896–1962). Tesen går ut på att det går att formulera en algoritm för att avgöra om ett givet påstående i första ordningens logik är giltigt eller inte. Idag kallat *Avgörbarhetsproblemet*. (Ursprungligen ty. *Das Entscheidungsproblem*). Trots Hilberts enorma meritlista och omfattande analyser visade sig båda dessa idéer dessvärre sakna verifierbarhet. Mitt arbete ska handla om hur hans påståenden ledde fram till en av vår tids största upptäckter; att det går att definiera vad som överhuvudtaget är beräkningsbart. Materialet är omfattande och jag ska försöka ge en form av helhetsbild för läsaren. Mitt huvudsakliga uppdrag är dock att noggrant presentera EN form av beräkningsbarhet; teorin om *Partiella Rekursiva funktioner*. Jag ska också redogöra för varför avgörbarhetsproblemet saknar lösning. På vägen kommer vi bland många självklart att möta den amerikanske matematikern *Alonzo Church*(1903–1995) och *Alan Turing* (1912–1954) en engelsk matematiker, logiker och kryptoanalytiker vilka har betytt allra mest för diskussionen om beräkningsbarhet. Deras berömda tes; *Church-Turings hypotes* kommer också att avsluta detta arbete.

2 Diskussionen

2.1 Första spiken i kistan för David Hilbert

Kurt Gödel (1906–1978) en österrikisk matematiker och en av tidernas främsta logiker gav David Hilbert rätt såtillvida att den första ordningens predikatlogik är fullständig. Alla sanna satser kan bevisas sanna inom systemet. Detta var välkänt redan innan och inget revolutionerande. Men det systemet är unikt, för generellt sett stämmer inte detta.

Genom ett bevis, publicerat 1931, där Gödel numrerade alla satser enligt ett bestämt mönster s.k. *Gödelnumrering*, kunde han bevisa att det inom varje konsistent formellt system som innehåller aritmetiska operationer finns sanna påståenden som kan bevisas vara både sanna och falska samtidigt. Att ett påstående kan bevisas sant och falskt på samma gång är förstås oacceptabelt. Det finns ingenting som säger att påståendet inte kan bevisas vara sant eller falskt i ett annat system men det räcker ju inte om Hilberts teori skulle gälla! Detta bevis kallas för *Gödels första ofullständighetsteorem*.

Gödels andra ofullständighetsteorem är en följdsats och lyder: ”Konsistensen hos ett formellt system, tillräckligt för aritmetiken, kan inte bevisas inom systemet”. (Gödel 1931) I sig var denna upptäckt naturligtvis revolutionerande men en annan sak som fick uppmärksamhet av den amerikanske matematikern *Stephen Cole Kleene* (1909–1994) var att Gödel i sina bevis använde en särskild sorts aritmetiska funktioner som han kallade rekursiva funktioner.

Kleene utvecklade dessa, som han kom att kalla primitiva rekursiva funktioner till Generella eller Partiella rekursiva funktioner, vilket han skrev om i sitt arbete 1936 publicerat i Matematische annalen samma år (Kleene 1936) Och dessa skulle visa sig omistliga när det kom just till definitionen av beräkningsbarhet.

2.2 Alan Turing och diagonalproblemet eller andra spiken i kistan för David Hilbert

Alan Turing var bara 22 år när han på en föreläsning av den ungersk-amerikanske matematikern *Johan von Neumann* (1903–1957) hörde talas om Hilberts idé om en algoritm som skulle kunna avgöra bevisbarheten i godtyckliga påståenden i matematisk logik. En algoritm som i förlängningen då skulle kunna

avgöra om en matematisk funktion är beräkningsbar eller inte, utan att behöva testa själva funktionen.

Neumann själv höll inte för osannolikt att det skulle kunna finnas någon form av mekanisk process som skulle kunna tillämpas på ett matematiskt påstående och svara på om det gick att bevisa utan att pröva påståendet först.

Andra forskare slog ifrån sig och en del raljerade t.o.m. över idén. [1] Men Turing föll just för uttrycket ”En mekanisk process” och började fundera över, tro det eller ej, skrivmaskinens uppbyggnad. Ett verktyg han alltid fascinerats av. Skrivmaskinen med sina typarmar som skriver en bokstav i taget på ett exakt avstånd från varandra, där bokstäverna slutligen blir till ord och komplicerade sammanhang.

Turings dröm var dock en matematisk skrivmaskin som inte behövde en människas intervention. Han föreställde sig en självständig maskin som utförde alla matematiska beräkningar som kunde tänkas existera enligt en exakt programmering och där processen utfördes av ett typhuvud som rörde sig över ett endimensionellt papper i form av en oändlig, rutindelad remsa. Vid varje steg skulle typhuvudet röra sig ett steg åt höger eller vänster eller stanna och i varje ruta skulle det skrivas exakt en symbol, raderas en symbol eller accepteras det som redan stod skrivet. När programmet var klart skulle maskinen stanna och ge ett resultat i ruta nummer ett.

Turing hade vid denna tid (1935) ett stort stipendium på Cambridge där han delade sin tid mellan att studera och undervisa och den myckna fritiden tillbringade han med långdistanslöpning. Det lär ha varit när han låg och vilade på en äng i Grantchester under en av dessa löprundor som han kom på hur Hilberts tredje fråga skulle besvaras. Inspirerad av det s.k. *Diagonalbeviset* som den tyske matematikern *Georg Cantor* (1845–1918) hade presenterat, där han på ett elegant sätt hade bevisat att alla irrationella tal var oupräkneligt många och att de därigenom skiljde sig från de rationella talen som var uppräkningsbara.

Turing var helt överens med Cantor i den grundläggande analysen men det han upptäckte var att det inte gick att omfatta ens det uppräkningsbara. Det

[1] Matematikern G.H. Hardy (1877–1947) kommenterade syrligt att en sådan mekanisk process såklart var omöjlig och att det var tur eftersom det skulle göra alla matematiker överflödiga! Citerat ur *A mathematical proof* (s.16) publicerat i tidskriften *Mind* 1929 efter en föreläsning i Cambridge året innan

uppräkningsbara gav helt enkelt upphov till det ouppräkningsbara. Svaret på Hilbets fråga måste bli nej. Och det skulle, via Cantors tankar, bevisas just med en mekanisk process, en matematisk skrivmaskin. Eller som det till slut kom att kallas; en *Turingmaskin*! 1936 gavs Alan Turings *On computable numbers, with an application to the Entscheidungsproblem* (Turing 1936) ut och efter det var inget sig likt i den beräkningsbara världen. För visst kunde man använda mekaniska processer för att lösa matematiska problem. Kanske att det t.o.m. var det allra bästa sättet med tanke på den kommande utvecklingen som vi inte ens sett slutet på idag. (Hodges 1983)

I kommande två kapitel ska vi titta närmare på detta. Definitioner och beräkningar är hämtade ur eller inspirerade av kapitel 1-2 i Nigel Cutlands bok *Computability, An introduction to recursive function theory* (Cutland 1980).

3 Vad är en funktion och vilka funktioner är beräkningsbara?

Låt oss föreställa oss en magisk låda som vi döper till multiplikation.

$$\text{Vi skickar in två tal } (2,4) \rightarrow \left[\text{Multiplikation}\right] \rightarrow \text{Och ut kommer produkten } 8$$

Vi säger att multiplikation är en funktion som varje gång, på samma sätt utför samma matematiska procedur som vi kan lita på oavsett vilka tal vi skickar in. Den klarar även att räkna ut en luring som två negativa tal utan att missa att det ger ett positivt svar. Lätt för oss att förstå som räknat multiplikation hela livet att förstå. Vad är problemet?

Problemet är att förstå hur multiplikation fungerar på ett exakt plan. Ved händer egentligen i lådan? Kan vi vara säkra på att svaret alltid stämmer? Och på att det alltid kommer ett svar? Och finns det funktioner (= magiska lådor) som faktiskt inte är beräkningsbara (levererar ett svar) trots att de verkar utföra en beräkning? Se på följande exempel:

$$\pi \longrightarrow \left[g(n)\right] \longrightarrow \overbrace{7,7,7,\cdots,7}^{n}$$

$$g(n) = \begin{cases} 1 & \text{om det går att hitta n 7:or i rad i } \pi\text{:s decimalutveckling,} \\ 0 & \text{annars} \end{cases}$$

Helt rimlig tanke. Vi matar in π i vår nyss uppkomna låda som vi kan döpa till *Decimalkontroll* och slår oss till ro för att se om den hittar n st. 7:or och då skickar ut en 1:a efter en stund. Men om den inte hittar n st. 7:or kommer den fortsätta söka och oturligt nog har π oändligt många decimaler, så vi kommer kanske aldrig att få veta om *decimalkontroll* spottar ur sig en 1:a. En 0:a kommer iallafall inte att komma, det går ju alltid att söka lite till..

Det som behövs är alltså att bestämma exakt hur alla beräkningsbara algoritmer, som vi instinktivt vet fungerar, är uppbyggda för att lita på att vi tänkt rätt. Men vi behöver också kunna definiera vilka funktioner som inte är beräkningsbara och det svåraste av allt; hur visar vi hur algoritmer som endast fungerar för vissa ingångsvärden är uppbyggda? Vad finns *egentligen* i den magiska lådan?

4 Den obegränsade registreringsmaskinen

4.1 Maskinens uppbyggnad

Vi fortsätter på fantasins väg och placerar en sorts imaginär dator framför oss, inte olik den vi använder varje dag, men mycket enklare och med ett obegränsat minne och med en obegränsad yta att utföra beräkningar på. Vi kallar denna fantasidator för *Den obegränsade registreringsmaskinen* (fr. eng. *The universal register machine, URM*). Utan att gå in på detaljer kan vi påstå att den är en variant av den maskin som Alan Turing utvecklade. Maskinen består av en lång remsa som börjar med ruta 1 till vänster och åt höger fortsätter med ruta 2, ruta 3,..., ruta i.. (för varje $i \in \mathbb{N}$)

	R_1	R_2	R_3	R_4	R_5	R_6	
$\cdots$	r_1	r_2	r_3	r_4	r_5	r_6	$\cdots$

Varje ruta innehåller ett naturligt tal som vi benämner r_1 i R_1, r_2 i R_2 etc.

Innehållet i rutorna kan ändras av instruktioner som URM känner igen. Instruktionerna svarar mot mycket enkla performativa matematiska procedurer. En komplett lista med instruktioner utgör ett program P.

Vi börjar alltså med att förse URM med ett program P och en första konfiguration, d.v.s. en sekvens $a_1, a_2, a_3, \ldots$ av naturliga tal i rutorna $R_1, R_2, R_3, \ldots$ följt av ett oändligt antal nollor i det oändliga antal rutor som följer.

Antag att P består av s instruktioner $I_1, I_2, I_3, \ldots, I_s$.

URM börjar beräkningen med att lyda I_1, sedan $I_2, I_3 \ldots$ etc. tills uträkningen är klar. Svaret anges i R_1, om inget annat anges. Vi ska längre fram i texten presentera ett exempel på additionsberäkningen

$$x + y = 5 + 3,$$

men först ska vi gå igenom de fyra typerna av instruktioner som URMs program består av. De lyder:

1. Nollinstruktionen

   För varje $n = 1, 2, 3 \ldots$ finns en nollinstruktion $Z(n)$ (fr. eng. *zero*). Svaret från URM på $Z(n)$ är att ändra innehållet i ruta $n, R(n)$ till 0 medan alla andra rutor förblir oförändrade. Om URM är i följande konfiguration:

R_1	R_2	R_3	R_4	R_5	R_6	
9	6	5	23	7	4	$\cdots$

   ... och lyder Nollinstruktionen $Z(3)$ får vi alltså resultatet:

R_1	R_2	R_3	R_4	R_5	R_6	
9	6	0	23	7	4	$\cdots$

   då svaret på $Z(n)$ är noterat som $0 \to R_n$ eller ännu hellre

   $r_n := 0$ (r_n blir 0 oavsett tidigare värde.)

2. Efterföljandeinstruktionen

   För varje $n = 1, 2, 3 \ldots$ finns det en Efterföljandeinstruktion $S(n)$ (fr.eng. *successor*). Svaret från URM är denna gång att ändra rutans nummer uppåt med 1. Resultatet av Efterföljandekonstruktionen $S(n)$ är noterat som $r_n + 1 \to R_n$ eller $r_n := r_n + 1$ (r_n "blir" $r_n + 1$)

3. Överföringsinstruktionen

   För varje $m = 1, 2, 3 \ldots$ och $n = 1, 2, 3 \ldots$ finns en Överföringsinstruktion $T(m, n)$(fr. eng. *transfer*). URM:s svar på instruktionen är att byta ut

siffran r_n i R_n mot siffran r_m i R_m. Alla andra rutor är precis som innan, oförändrade. Om URM är i följande konfiguration:

R_1	R_2	R_3	R_4	R_5	R_6	
9	6	5	23	7	4	$\cdots$

och lyder Överföringsinstruktionen $T(3,1)$ får vi såklart resultatet:

R_1	R_2	R_3	R_4	R_5	R_6	
5	6	5	23	7	4	$\cdots$

Svaret på $T(m,n)$ noteras som $r_m \to R_n$ eller $r_n := r_m$ (r_n ”blir” r_m).

4. Hoppinstruktionen

   Hoppinstruktionen används för att kunna hoppa i instruktionslistan om du vill ha alternativ till påföljande instruktion. Ett exempel kan se ut som följer:

   P befinner sig på instruktion 5; I_5 och skulle alltså egentligen gå omedelbart till I_6, men I_5 bjuder på två alternativ; ”Om $r_2 = r_3$ **hoppa** till I_{10} i programmet, om $r_2 \neq r_3$ gå till nästkommande instruktion i programmet, i detta fall I_6”.

   Instruktionen ser ut som följer: $J(m,n,q)$ (fr.eng. *jump*) där $m = 1,2,3\ldots$, $n = 1,2,3\ldots$ och $q = 1,2,3\ldots$

   Antag att Hoppinstruktionen är införlivad i ett program P och att innehållet i R_m och R_n jämförs utan någon ytterligare förändring. Då gäller;

   Om $r_m = r_n$ går URM till den q:te instruktionen i P. Om $r_m \neq r_n$, går URM till nästa instruktion i P.

   Ex.

$$x + y = 5 + 3,$$

R_1	R_2	R_3	R_4	R_5	R_6	
5	3	0	0	0	0	$\cdots$

$I_1:$ $J(3,2,5)$
$I_2:$ $S(1)$
$I_3:$ $S(3)$
$I_4:$ $J(1,1,1)$
$I_5:$

I ord:

Om $r_2 = r_3$ gå till I_5 och stanna. Om $r_2 \neq r_3$ addera 1 till r_1 och addera 1 till r_3. Se om $r_1 = r_1$ (vilket naturligtvis är fallet:-), börja då om från början. Om $r_2 = r_3 \ldots$ Fortsätt denna beräkningsloop tills $r_2 = r_3$ och gå till I_5 och stanna. Svaret (8) kommer att läsas av i R_1 och URM kommer att ha stoppat.

4.2 Den formella definitionen av URM-beräkningsbarhet

Beräkningen i förra avsnittet utfördes i ett begränsat antal steg och kom till ett avslut. Vi säger att beräkningen konvergerar. Skulle beräkningen ej gå att genomföra och alltså aldrig komma fram till ett avslut säger vi att beräkningen divergerar. Så vad är en URM-beräkningsbar funktion formellt?

Om vi har en funktion från $\mathbb{N}^n \to \mathbb{N}$ $(n \geq 1)$ och vi kör den i URM, då är den spontana känslan att funktionen har en lösning i form av ett tal OM den är beräkningsbar. Stannar beräkningen aldrig och vi lämnas utan svar är det rimligt att tänka sig att det kan bero på något av följande: 1) Att funktionen ej är beräkningsbar 2) att funktionen f från $\mathbb{N}^n$ till $\mathbb{N}$ inte fått de ingångsvärden som är definierade för f.

Vi ger följande definitioner:

Låt f vara en partiell funktion från $\mathbb{N}^n$ till $\mathbb{N}$.

a) Antag att P är ett program och låt $a_1, a_2, a_3, .., a_n, b \in \mathbb{N}$

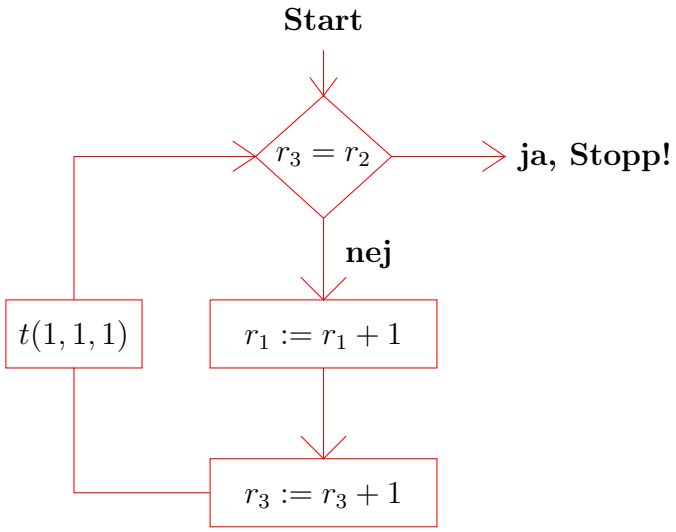
(i) Beräkningen $P(a_1, a_2, .., a_3)$ konverterar till b om $P(a_1, a_2, .., a_n) \downarrow$ och lämnar svaret b i R_1 då beräkningen slutar. Vi skriver $P(a_1, a_2, .., a_n) \downarrow b$;

(ii) P URM-beräknar f om, för alla $a_1, a_2, .., a_n, b$ $P(a_1, a_2, .., a_n) \downarrow b$
OMM $(a_1, a_2, .., a_n) \in Dom(f)$ och $f(a_1, a_2, .., a_n) = b$ (Särskilt gäller att $P(a_1, a_2, .., a_n) \downarrow OMM$ $(a_1, a_2, .., a_n) \in Dom(f)$).

b) Funktionen f är URM-beräkningsbar om det finns ett program som URM-beräknar f.

Klassen av URM-beräkningsbara funktioner betecknas $\mathscr{C}$ och n-ära URM-beräkningsbara funktioner betecknas $\mathscr{C}_n$. Under detta kapitel säger vi att något är beräkningsbart om det kan beräknas av en URM.

Vi tittar nu på flödesschemat av vårt exempel $x + y = 5 + 3$,



Vi ser att funktionen $f(5,3)$ går att beräkna på en URM. Alltså är funktionen beräkningsbar!

Notera att det kan finnas flera program som beräknar en och samma funktion. Det är ingen skillnad på funktionernas beräkningsbarhet på grund av detta faktum. Däremot räknar varje program endast en funktion av en given aritet.

4.3 När ingångsvärden påverkar beräkningsbarheten

Vi ska nu titta närmare på ett program som stoppar och ger ett svar för vissa ingångsvärden men inte för andra, d.v.s. där endast vissa ingångsvärden, $a_n \in dom(f)$. Det beror helt enkelt på hur vår initiala konfiguration ser ut.

Vi tar programmet P som räknar följande funktion;

$$f(x) = \begin{cases} \frac{1}{3}x, & \text{Om } x \text{ är delbart med 3,} \\ \text{Odefinierat,} & \text{annars} \end{cases}$$

Programmet ser ut som följer;

$$\begin{aligned} I_1 &: \ S(2) \\ I_2 &: \ S(2) \\ I_3 &: \ S(2) \\ I_4 &: \ S(3) \\ I_5 &: \ J(1,2,7) \\ I_6 &: \ J(1,1,1) \\ I_7 &: \ T(3,1) \end{aligned}$$

Och som vi förstår av följande bild gäller ju endast detta program för ingångsvärden som är jämnt delbara med tre.

R_1	R_2	R_3	R_4	R_5	R_6	
6	0	0	0	0	0	$\cdots$

Stannar efter 2 loopar.

R_1	R_2	R_3	R_4	R_5	R_6	
5	0	0	0	0	0	$\cdots$

Loopar för evigt..

Vi ser nedan den allmänna formen för att beskriva just dessa beräkningar:

$$g(n) = \begin{cases} \text{Det unika } b, & \text{så att } P(a_1, .., an) \downarrow b \text{ om } P(a_1, .., a_n) \downarrow, \\ \text{Odefinierat,} & \text{om } P(a_1, .., a_n) \uparrow \end{cases}$$

URM fungerar även för att bestämma egenskaper hos tal och relationer mellan tal. Om vi exempelvis frågar huruvida y har egenskapen av att vara en multipel av x och vi har svaren *ja* eller *nej* att tillgå kan svaren utan problem representeras av 1 och 0 eller vice versa. Även booleska operationer går att utföra eftersom vi kan ersätta sant med 1 och falskt med 0.

URM fungerar endast i domänen av de naturliga talen $\mathbb{N}$. Vi ska se att det gäller även andra former av beräkningsbarhet. Men med URM är det lätt gjort att koda om exempelvis funktioner på heltalen $\mathbb{Z}$ till funktioner på de naturliga talen, beräkna uppgiften och sedan koda tillbaka svaret till heltalsdomänen.

4.4 Exempel på mer komplexa funktioner, beräknade av URM

Målet är såklart att vi kan visa att vi i slutändan kan använda URM till alla beräkningsbara funktioner, tal, relationer etc. Vi kan då helt enkelt kombinera enkla program för att lyckas lösa mer invecklade problem. Vi noterar först att vi har tre basfunktioner som är beräkningsbara:

Lemma 4.1. *Följande basfunktioner går att beräkna:*

a) Nollfunktionen $\mathbf{0}(\mathbf{0}(x) = 0$ *för alla* $x)$*,*

b) Efterföljandefunktionen $x + 1$*,*

c) För varje $n \geq 1$ *och* $1 \leq i \leq n$*, Projektionsfunktionen* U_i^n *som skrives* $U_i^n(x_1, x_2, \ldots, x_n) = x_i$*.*

Bevis. Enligt:

a) $\mathbf{0}$: Program $Z(1)$,

b) $x + 1$: Program $S(1)$,

c) U_i^n: Program $T(i, 1)$.

□

En bra metod för att skapa nya funktioner är att projicera in resultatet från en funktion i en annan funktion. Detta benämns även *Kompositionsfunktion.* I följande sats visar vi att när denna process används på beräkningsbara funktioner, så är resultatfunktionen också beräkningsbar. Vi säger kortfattat att $\mathscr{C}$ är låst el. begränsad under substitution.

Sats 4.2. *Antag att* $f(y_1, y_2 \ldots, y_k)$ *och* $g_1(\mathbf{x}), g_2(\mathbf{x}), \ldots g_k(\mathbf{x})$ *är beräkningsbara funktioner där* $\mathbf{x} = (x_1, x_2, \ldots, x_n)$*, då är funktionen* $h(\mathbf{x})$ *definierad som* $h(\mathbf{x}) \simeq f(g_1(\mathbf{x}), g_2(\mathbf{x}), \ldots g_k(\mathbf{x}))$ *beräkningsbar.*

Bevis. Antag att $F, G_1, \ldots, G_k$ är standardprogram som beräknar $f, g_1, \ldots, g_k$, då skapar vi ett program H som omfattar följande tydliga procedur för att beräkna h:

Givet $\mathbf{x}$, använd programmen $G_1, \ldots, G_k$ för att i tur och ordning beräkna $(g_1(\mathbf{x}), g_2(\mathbf{x}), \ldots g_k(\mathbf{x}))$ och spara resultaten på bestämd plats. Använd sedan program F för att beräkna $f(g_1(\mathbf{x}), g_2(\mathbf{x}), \ldots g_k(\mathbf{x}))$. □

Vi får inte förlora information på vägen om erhållna värden som ska användas i kommande beräkningar! Vi bestämmer att:

$m = max(n, k, p(F), p(G_1), \ldots, p(G_k))$ och vi börjar med att lagra $\mathbf{x}$ i rutorna $R_{m+1}, \ldots, R_{m+n}$; rutorna $R_{m+n+1}, \ldots, R_{m+n+k}$ kommer att användas för att dubblera värdena av $g_i(\mathbf{x})$ när de beräknats för $i = 1, 2, \ldots, k$. De rutorna är helt ignorerade vid beräkningarna av funktionen.

$R_1 \ldots R_m$	$R_{m+1} \ldots R_{m+n}$	R_{m+n+1}	R_{m+n+2}	$\ldots$	R_{m+n+i}		
$\cdots$	$\mathbf{x}$	$g_1(\mathbf{x})$	$g_2(\mathbf{x})$	$\cdots$	$g_i(\mathbf{x})$	0	$0 \cdots$

Vi ska nu titta på ett program för att beräkna funktionen

$$h(x_1, x_2, x_3) = x_1 + x_2 + x_3$$

där vi byter ut $x_1 + x_2$ mot x och x_3 mot y i $x + y$. Vi beräknar exemplet $5 + 2 + 4$, vilket ger oss ett startfält som ser ut som följer;

R_1	R_2	R_3	R_4	R_5	R_6	
5	2	4	0	0	0	$\cdots$

Om vi uttrycker beräkningen som:

$h =$ add $\bullet$ {add $\bullet$ (x_1, x_2), proj $\bullet$ (x_3)}

kan den beskrivas med nedanstående program på en URM:

$I_1:$	$T(1,4)$	vi lagrar x_1, x_2 och x_3 i rutorna $4, 5$ och 6
$I_2:$	$t(2,5)$	
$I_3:$	$T(3,6)$	
$I_4:$	$T(3,1)$	$\rightarrow g_1(\boldsymbol{x})$ skickas till ruta 1
$I_5:$	$T(1,7)$	$\rightarrow$ vi lagrar $g_1(\boldsymbol{x})$ i ruta 7
$I_6:$	$T(4,1)$	$\rightarrow$ vi hämtar tillbaka x_1
$I_7:$	$Z(3)$	
$I_8:$	$J(3,2,12)$	vi beräknar $g_2(\boldsymbol{x})$
$I_9:$	$S(1)$	
$I_{10}:$	$S(3)$	
$I_{11}:$	$J(1,1,8)$	
$I_{12}:$	$T(1,8)$	$\rightarrow$ vi lagrar $g_2(\boldsymbol{x})$ i ruta 8
$I_{13}:$	$T(7,2)$	$\rightarrow$ vi hämtar tillbaka $g_1(\boldsymbol{x})$
$I_{14}:$	$Z(3)$	
$I_{15}:$	$T(3,2,19)$	vi beräknar $f(g(\mathbf{x}))$
$I_{16}:$	$S(1)$	
$I_{17}:$	$S(3)$	
$I_{18}:$	$J(1,1,15)$	
$I_{19}:$	STOPP	$\rightarrow$ Vi läser av svaret i R_1

Såhär ser det ut när vi är klara. Det korrekta svaret är tydligt avläsbart i ruta R_1:

R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
11	4	4	5	2	4	4	7	0	$0\cdots$
$h \simeq f(g(\mathbf{x}))$				$\mathbf{x}$		$g_1(\mathbf{x})$	$g_2(\mathbf{x})$		

4.5 Redo för nästa kapitel i beräkningsbarhetens tjänst

Trots att jag bara givit ett fåtal exempel hoppas jag att läsaren fått en någotsånär fullödig bild av hur URM fungerar. Jag tänker att vi nu är redo för mitt huvudsakliga tema för denna text; teorin om *Partiella Rekursiva funktioner*.

Där inget annat anges har jag i kapitel 5 hämtat fakta från *Mathematical Logic, A course with exercises Part II* av *René Cori* och *Daniel Lascar* i översättning av *Donald H. Pelletier*, sid. 7–22 (Cori och Lascar 2000)

5 Partiella Rekursiva funktioner

5.1 En första översikt

Partiella Rekursiva funktioner är uppbyggda av:

- De tre *Basfunktionerna* vilka är beräkningsbara enligt Lemma 4.1 och Bevis 4.1 i föregående avsnitt

och de därifrån härledda

- *Kompositionsfunktionerna*
- *Primitiv rekursion*

och

- Begränsad och obegränsad *Minimalisation*

Dessa funktioner arbetar endast på de naturliga talen $\mathbb{N}$. Vi säger att deras *definitionsmängd* och *värdemängd* alltid består av naturliga tal $\mathbb{N}$.

Innan vi ger definitionerna kommer här några förtydliganden gällande notation och förklaringar av uttryck:

- Låt P vara ett heltal. Vi skriver

  $$\mathcal{F}_p : \mathbb{N}^p \to \mathbb{N}$$

  Om $p = 0$ kommer vi enligt konvention att hävda att det enda elementet i $\mathbb{N}^p$ är den tomma mängden. $\mathcal{F}_0$ är alltså funktionen av 0 element. Mängden av $\bigcup_{p\in\mathbb{N}} \mathcal{F}_p$ betecknas $\mathcal{F}$.

- om i är ett heltal $1 \leq i \leq p$ är den i:te projektionen P_i^p i funktionen $\mathcal{F}_p$ definierad som

  $$P_i^p(x_1, x_2, x_3, \ldots, x_p) = x_i$$

  Vi kallar den *Projektionsfunktionen* P.

- Per definition är *Efterföljandefunktionen* S:

$$S(x) = x + 1$$

vars värde av ett heltal n är $n + 1$.

- Om $f_1, f_2, \ldots, f_n$ hör till $\mathcal{F}_p$ och g hör till $\mathcal{F}_n$ då är den *Sammansatta funktionen* h definierad som

$$h = g(f_1, f_2, \ldots, f_n)(x_1, x_2, \ldots, x_p)$$

Definition 5.1. Definition via Rekursion *Rekursionsfunktionen*

• Detta är proceduren för att definiera funktionen som följer av följande uppenbara faktum.

Om $f \in \mathcal{F}_p$ och $g \in \mathcal{F}_{p+2}$, då har vi en och endast en funktion $h \in \mathcal{F}_{p+1}$ som uppfyller ovanstående villkor:

för alla $x_1, x_2, \ldots, x_p$ och y i $\mathbb{N}$

(i) $h(x_1, x_2, \ldots, x_p, 0) = f(x)$,

(ii) $h(x_1, x_2, \ldots, x_p, y + 1) = g(x_1, x_2, \ldots, x_p, y, h(x_1, x_2, \ldots, x_p, y))$.

Vi säger att h är funktionen som är *definierad via rekursion* från f (basvillkoret) och g (det rekursiva steget).

Definition 5.2. Mängden av primitiva rekursiva funktioner är den minsta delmängden E av $\mathcal{F}$ som uppfyller följande villkor

1) För varje heltal p innehåller E Nollfunktionen $\mathbf{0} : \mathbb{N} \to \mathbb{N}$ och alla därifrån härledda konstantfunktioner från $\mathbb{N}^p$ till $\mathbb{N}$,

2) För alla heltal p och för alla heltal i där $1 \leq i \leq n$, innehåller E alla projektioner P_i^p,

3) E innehåller efterföljandefunktionen S,

4) E är låst under sammansättning av funktioner. Det betyder att om n och p är heltal, om $f_1, f_2, \ldots, f_n$ är element i $\mathcal{F}_p$ som hör till E och om g är ett element i $\mathcal{F}_n$ som också hör till E, då hör den sammansatta funktionen $g(f_1, f_2, \ldots, f_n)$ till E.

5) E är låst under rekursion, det betyder att om p är ett heltal och om funktionerna f i $\mathcal{F}_p$ och g i $\mathcal{F}_{p+2}$ båda hör till E, då är funktionen h definierad via rekursion från f och g, även den tillhörande E.

Vi sätter:

$$\mathcal{R}_0 = \{\mathbf{0} : \mathbb{N} \to \mathbb{N}\} \cup \{\mathcal{P}_i^p : \mathbb{N}^p \to \mathbb{N}, 1 \leq i \leq p\} \cup \{\mathcal{S} : \mathbb{N}^2 \to \mathbb{N}\}$$

$$\begin{aligned}\mathcal{R}_{n+1} = \mathcal{R}_n \ &\cup \{h : h \text{ beräknas via rekursion från två funktioner i } \mathcal{R}_n\} \\ &\cup \{h : h \text{ beräknas via sammansättning från två funktioner i } \mathcal{R}_n\}\end{aligned}$$

Mängden av primitiva rekursiva funktioner är då lika med $\bigcup_{n\in\mathbb{N}} \mathcal{R}_n$

För att bevisa att en funktion är primitivt rekursiv räcker det med att visa hur man erhåller den genom att använda klausulerna 4) och 5) via funktionerna beskrivna i klausulerna 1), 2) och 3). Å andra sidan, för att bevisa att alla primitiva funktioner besitter egenskapen $\mathscr{E}$, räcker det att visa att funktionerna i klausulerna 1), 2) och 3) besitter den och att klassen av funktioner som har denna egenskap är sluten vid sammansättning och rekursion.. Vi kan också se att det, för varje primitiv rekursiv funktion f, finns en algoritm som beräknar den. Det är sant för funktionerna i $\mathcal{R}_0$ och om det är sant för funktionerna i $\mathcal{R}_n$ är det också sant för dem i $\mathcal{R}_{n+1}$.

Definition 5.3. En delmängd $\mathcal{A} \subseteq \mathbb{N}^p$ kallas för primitivt rekursiv om dess karaktäristiska funktion är primitivt rekursiv. Den karaktäristiska funktionen $(\mathcal{X}^{\mathcal{A}} =)\mathcal{X}_{\mathcal{A}}$ för mängden $\mathcal{A}$ definieras av

$$\mathcal{X}_{\mathcal{A}} = \begin{cases} 1 & \text{om } \ (n_1, n_2, \ldots, n_p) \in \mathcal{A} \\ 0 & \text{annars} \end{cases}$$

Om $\mathscr{E}(x_1, x_2, \ldots, x_p)$ är en egenskap applicerbar på heltalen $n_1, n_2, \ldots, n_p$ (vi säger också predikatet med n argument) är $\mathscr{E}$ primitivt rekursiv om mängden

$$\{(x_1, x_2, \ldots, x_p) : (x_1, x_2, \ldots, x_p) \ \text{ uppfyller } \mathscr{E}\}$$

är primitivt rekursiv.

Definition 5.3 är ett exempel på att det går att utvidga beräkningsbara funktioner till beräkningsbara tal, egenskaper, talserier och relationer. Naturliga tal, heltal, rationella tal och exakt definierade reella tal såsom ex. $\sqrt{2}, e$ och π.

En för mig intuitiv men långtifrån matematisk exakt definition av vad ett icke beräkningsbart tal är skulle kunna uttryckas som följer. Det är hämtat från internet under en icke verifierar källa men jag nämner det ändå:

”Oändliga reella tal som i sin egen beskrivning är i sin kortaste form är ej beräkningsbara. Ingen algoritm kan i ett ändligt antal steg beräkna dem”.

5.2 Primitiva rekursioners definitioner och mängder

Nu följer här ett antal exempel på vardagliga beräkningar beskrivna i rekursiv form.

5.2.1 Addition

Addition $f(x, y) = x + y$ är en primitiv rekursiv funktion som definieras enligt följande

$$\begin{aligned} x + 0 &= x \\ x + (y + 1) &= (x + y) + 1 \end{aligned}$$

Om vi påminner oss om att

$$\begin{aligned} f &: \mathbb{N}^p \to \mathbb{N}, \\ g &: \mathbb{N}^{p+2} \to \mathbb{N} \\ h &: \mathbb{N}^{p+1} \to \mathbb{N} \end{aligned}$$

kan vi skriva

$$\begin{aligned} h(x, 0) &= f(x) = P_1^1 \\ h(x, y + 1) &= g(x, y, h(x, y)) = S(h(x, y)) \end{aligned}$$

och vi nu sätter h= addition (add) och ρ = rekursion så får vi slutligen

$$add = \rho^2[P_1^1, S \bullet P_3^3]$$

I det föregående och hädanefter använder vi oss av regeln att om vi permuterar, identifierar eller lägger till ”stumma” variabler hos en primitivt rekursiv funktion, så är även resultatet primitivt rekursivt.

Exempelvis från funktionen $f(y_1, y_2)$ kan vi erhålla:

$$\begin{aligned} h_1(x_1, x_2) &\simeq f(x_2, x_1) \\ h_2(x) &\simeq f(x, x) \\ h_3(x_1, x_2, x_3) &\simeq f(x_2, x_3) \end{aligned}$$

Denna regel användes även vid vårt exempel med URM:

$$h(x_1, x_2, x_3) = x_1 + x_2 + x_3$$

5.2.2 Multiplikation

Multiplikation $f(x, y) = x \cdot y$ är en primitiv rekursiv funktion som definieras enligt följande;

$$\begin{aligned} x \cdot 0 &= 0 \\ x \cdot (y + 1) &= x \cdot y + x \end{aligned}$$

$$\begin{aligned} h(x, 0) &= f(0) \\ h(x, y + 1) &= g(x, y, h(x, y)) \end{aligned}$$

Om vi denna gång skriver h = multiplikation (mult.) och ρ = rekursion, får vi (tillsammans med vår redan definierade addition)

$$mult = \rho^2[P_2^2, add \bullet [P_1^3, P_3^3]]$$

Denna, lite kryptiska, definition går att härleda via det banala exemplet:

$$3 \cdot 3 = 2 \cdot 3 + 3 = 1 \cdot 3 + 3 + 3 = 3 + 3 + 3$$

5.2.3 Mindre än eller lika med

Lite längre fram i min text kommer jag att använda mig av den rekursiva funktionen *"MindreänEllerLikamed"* och det definieras i sin tur via de rekursiva funktionerna *"BlirNoll"* (fr.eng *IsZero*) och subtraktion via *Företrädarefunktionen, "pred".* (fr. eng. *Predecessor function*). Jag vill därför redan nu göra en översikt även av dessa. I detta lilla avsnitt samt i avsnitten 5.2.4 – 5.3 återvänder jag till *Computability* kap. 1–2 (Cutland 1980)

Företrädarefunktionen*** $x \dot{-} 1$ definieras via rekursion och bevisas enligt följande:

$$\begin{aligned} 0 \dot{-} 1 &= 0 \\ (x+1) \dot{-} 1 &= x \end{aligned}$$

Och om vi sätter $h = pred$

$$\begin{aligned} h(0) &= f() = 0 = noll^0 \\ h(y+1) &= g(y, h(y)) = y \end{aligned}$$

så får vi:

$pred = \rho^0[noll^0, proj_1^1]$

Avbruten subtraktion $x \dot{-} y$ kan skrivas som:

$$x \dot{-} y = \begin{cases} x - y & (\text{om } \ x \geq y) \\ 0 & \text{annars} \end{cases}$$

och bevisas enligt följande:

$$\begin{aligned} x \dot{-} 0 &= x \\ x \dot{-} (y+1) &= (x \dot{-} y) \dot{-} 1 \text{ (med hjälp av ***)} \end{aligned}$$

där, om $h = sub$

$$\begin{aligned} h(x, 0) &= f(x) = P_1^1 \\ h(x, y+1) &= g(x, y, h(x, y)) \end{aligned}$$

har vi definitionen:

sub $= \rho^2[P_1^1, \text{pred} \bullet P_3^3]$

Vi definierar:

$BlirNoll = sub \bullet [\ S \bullet \mathbf{0}, P_1^1]$

och ”Mindre än eller lika med” kan nu definieras som

$MindreänEllerLikamed = BlirNoll \bullet sub.$

5.2.4 Definition genom falluppdelning

En användbar variant av funktioner är de som är beroende av omständigheter. För ett visst ingångsvärde används sålunda en funktion och för ett annat används en annan. Dessa funktioner är även de, primitivt rekursiva. Det mest klassiska exemplet är

$$|x| = \begin{cases} x & \text{om } \ x \geq 0, \\ -x & \text{om } \ x < 0 \end{cases}$$

Ett lite mer vardagligt exempel skulle kunna vara $g(x)$ där g är beräkningen av genomsnittlig tillväxttakt i cm/månad för ett barn 0–12 månader

$$g(\boldsymbol{x}) = \begin{cases} f_1(\boldsymbol{x}) & 0-3 \text{ mån}, \\ f_2(\boldsymbol{x}) & 4-6 \text{ mån}, \\ f_3(\boldsymbol{x}) & 7-9 \text{ mån}, \\ f_4(\boldsymbol{x}) & 10-12 \text{ mån}. \end{cases}$$

Definition 5.4. Anta att $f_1(\boldsymbol{x}), \ldots, f_k(\boldsymbol{x})$ är totala beräkningsbara funktioner och att

$M_1(\boldsymbol{x}), \ldots M_k(\boldsymbol{x})$ är bestämda egenskaper som säkrar att motsvarande funktion $g(\boldsymbol{x})$ gäller. Då är funktionen $g(\boldsymbol{x})$ giltigt beskriven som

$$g(\boldsymbol{x}) = \begin{cases} f_1(\boldsymbol{x}) & \text{om } M_1(\boldsymbol{x}) \text{ gäller}, \\ f_2(\boldsymbol{x}) & \text{om } M_2(\boldsymbol{x}) \text{ gäller}, \\ \vdots & \\ \vdots & \\ f_k(\boldsymbol{x}) & \text{om } M_k(\boldsymbol{x}) \text{ gäller}, \end{cases}$$

beräkningsbar.

Bevis. $g(\boldsymbol{x}) = C_1 \cdot M_1(\boldsymbol{x}) f_1(\boldsymbol{x}) + \ldots + C_k \cdot M_k(\boldsymbol{x}) f_k(\boldsymbol{x})$

är beräkningsbar via substitution av addition och multiplikation. C är här de Booleska sant och falskt. □

5.2.5 Exempel på en funktion inom en annan domän än de naturliga talen, $\mathbb{N}$

Som tidigare nämnts arbetar både URM och beräkningar/definitioner av primitiva rekursiva funktioner endast på de naturliga talen, $\mathbb{N}$. Men det går att koda om tal från exempelvis domänen heltal, $\mathbb{Z}$. När vi gör det hamnar vi just i definitioner/beräkningar genom falluppdelning. Här är ett exempel.

Funktionen $f(x) = x - 1$ är för alla värden på x endast beräkningsbar i heltalsdomänen $\mathbb{Z}$. Vi kodar om den till $f^*(x)$ på heltalsdomänen $\mathbb{N}$ enligt:

$f^* = a \circ f \circ a^{-1}$ där;

$$a(n) = \begin{cases} 2n & \text{om } n \geq 0, \\ -2n-1 & \text{om } < 0 \end{cases}$$

$$a^{-1}(m) = \begin{cases} \frac{1}{2}m & \text{om } m \text{ är jämn}, \\ -\frac{1}{2}(m+1) & \text{om } m \text{ är udda} \end{cases}$$

Vi applicerar beräkningarna:

$$\begin{aligned} 2(\frac{1}{2}x - 1) &= x - 2 \\ -2(\frac{1}{2}x - 1) - 1 &= -x + 1 \\ 2(-\frac{1}{2}x - \frac{1}{2} - 1) &= -x - 3 \\ -2(-\frac{1}{2}x - \frac{1}{2} - 1) - 1 &= x + 2 \end{aligned}$$

och resultatet blir:

$$f^*(x) = \begin{cases} x-2 & \text{om } x \geq 2, \ x \text{ är jämn}, \\ x+2 & \text{om } x \geq -1, \ x \text{ är udda} \\ -x+1 & \text{om } x < 2. \ x \text{ är jämn} \\ -x-3 & \text{om } x < -1, \ x \text{ är udda} \end{cases}$$

På detta sätt går funktionen att beräkna för alla värden på x.

5.3 Begränsad minimalisation eller μ-operatorn

Även om vi har bevisat att en viss funktion eller att en viss kombination av funktioner är rekursiva kan det vara av intresse att definiera funktionen med hjälp av begränsad minimalisation, den s.k. μ-operatorn. Jag illustrerar med ett utförligt exempel.

Anta att $f(\boldsymbol{x}, z)$ är en funktion, vilken som helst. Den begränsade summan, $\sum_{z<y} \ f(\boldsymbol{x}, z)$ och den begränsade produkten $\prod_{z<y} \ f(\boldsymbol{x}, z)$ är funktionerna av $\boldsymbol{x}, y$ beskrivna enligt följande:

i)

$$\begin{cases} \sum\limits_{z<0} f(\boldsymbol{x}, 0) = & 0 \\ \sum\limits_{z<y+1} f(\boldsymbol{x}, z) = & \sum\limits_{z<y} f(\boldsymbol{x}, z) + f(\boldsymbol{x}, y), \end{cases}$$

ii)

$$\begin{cases} \prod\limits_{z<0} f(\boldsymbol{x}, z) = & 1 \\ \prod\limits_{z<y+1} f(\boldsymbol{x}, z) = & \left(\prod\limits_{z<y} f(\boldsymbol{x}, z) \right) \cdot f(\boldsymbol{x}, z). \end{cases}$$

tes 5.5. *Anta att $f(\boldsymbol{x}, z)$ är en total beräkningsbar funktion, då är funktionerna $\sum_{z<y} f(\boldsymbol{x}, z)$ och $\prod_{z<y} f(\boldsymbol{x}, z)$ beräkningsbara.*

Bevis. Ekvationerna i och ii är definierade via rekursion från beräkningsbara funktioner. □

Även följande slutsats gäller:

Följdsats 5.6. *Anta att $f(\boldsymbol{x}, z)$ och $k(\boldsymbol{x}, \boldsymbol{w})$ är totala beräkningsbara funktioner, då är även funktionerna*

$$\sum_{z<k(\boldsymbol{x},\boldsymbol{w})} f(\boldsymbol{x}, z) \tag{1}$$

och

$$\prod_{z<k(\boldsymbol{x},\boldsymbol{w})} f(\boldsymbol{x}, z) \tag{2}$$

beräkningsbara.

Bevis. Vi substituerar y mot $k(\boldsymbol{x}, \boldsymbol{w})$ i den bundna summan $\sum_{z<k(\boldsymbol{x},\boldsymbol{w})} f(\boldsymbol{x}, z)$ och i den bundna produkten $\prod_{z<k(\boldsymbol{x},\boldsymbol{w})} f(\boldsymbol{x}, z)$ □

Om nu den bundna produkten blir noll vill vi kanske veta när det sker i beräkningen. Då kan vi använda oss av μ-operatorn. Vi skriver $\mu z < y(...)$ d.v.s. det minsta z, mindre än y så att...

För att denna funktion ska vara total bestämmer vi att $z = y$ om inget sådant värde på z existerar. Då kan vi t.ex. från en given funktion $f(\boldsymbol{x}, z)$ definiera en ny funktion g som

$$g(\boldsymbol{x}, y) = \mu z < y(f(\boldsymbol{x}, z)) = 0 \begin{cases} \text{det minsta } \ z < y, \text{så att } \ f(\boldsymbol{x}, z) = 0 \\ \text{(om ett sådant } \ z \text{ finns)}, \\ y \text{ om inget sådant } \ z \text{ finns.} \end{cases}$$

$\mu z < y$ kallas den begränsade minimalisationsoperatorn eller den begränsade μ-operatorn.

Sats 5.7. *Antag att $f(\boldsymbol{x}, y)$ är en total beräkningsbar funktion, då är även $\mu z < y(f(\boldsymbol{x}, z) = 0)$ det.*

Bevis. Studera funktionen

$$h(\boldsymbol{x}, v) = \prod_{u \leq v} sg(f(\boldsymbol{x}, u))$$

som är beräkningsbar enligt följdsats 5.6

Antag att följande gäller för ett givet $\boldsymbol{x}, y$, $z_0 = \mu z < y(f(\boldsymbol{x}, z) = 0)$.

Då ser vi att:

- om $v < z_0$ är $h(\boldsymbol{x}, v) = 1$
- om $z_0 \leq v < y$ är $h(\boldsymbol{x}, v) = 0$

och följaktligen:

- $z_0 =$ antalet v mindre än y så att $h(\boldsymbol{x}, v) = 1$, $= \sum_{v<y} h(\boldsymbol{x}, v)$

Därför är:

$$\mu z < y(f(\boldsymbol{x}, z) = 0) = \sum_{v<y} \Big(\prod_{u \leq v} sg(f(\boldsymbol{x}, u)) \Big)$$

och beräkningsbart enligt tes 5.5!

□

Vi återkommer lite längre fram till den s.k. obegränsade μ-operatorn som har ett något annorlunda användningsområde. Men först några ytterligare exempel på primitiva rekursiva funktioner som definieras med eller utan μ-operatorns hjälp!

5.3.1 Primtalsserien

Mängden $\{x : x$ är ett primtal$\}$ är primitivt rekursiv. x är ett primtal *omm* $x > 1$ och $\forall y < x(y \leq 1$ eller $y = x$ eller y delar inte $x)$

Även π- funktionen är primitivt rekursiv via vår bundna μ-generator på följande sätt:

$$\pi(0) = 2$$

$$\pi(n+1) = \mu z \leq (\pi(n)! + 1)[z > \pi(n) \text{ och } z \text{ är ett primtal}]$$

(Här använder vi oss av det välkända faktum att det alltid existerar ett primtal strikt mellan p och $p! + 2$)

5.3.2 Kodning av talföljder 1.

Begreppet beräkningsbarhet är även applicerbart på kodning av talföljder, d.v.s. det går att via funktioner göra om ändliga talföljder till tal eller uttrycka dem som andra talföljder. Här följer två olika system för denna uppgift. Observera att det inte bara är möjligt utan helt nödvändigt, både att kunna uttrycka talföljden som ett heltal i $\mathbb{N}$ och att ur samma heltal kunna återgå till talföljden det motsvarade!

Påstående 5.8. *För varje nollskilt heltal p, finns det primitiva rekursiva funktioner*

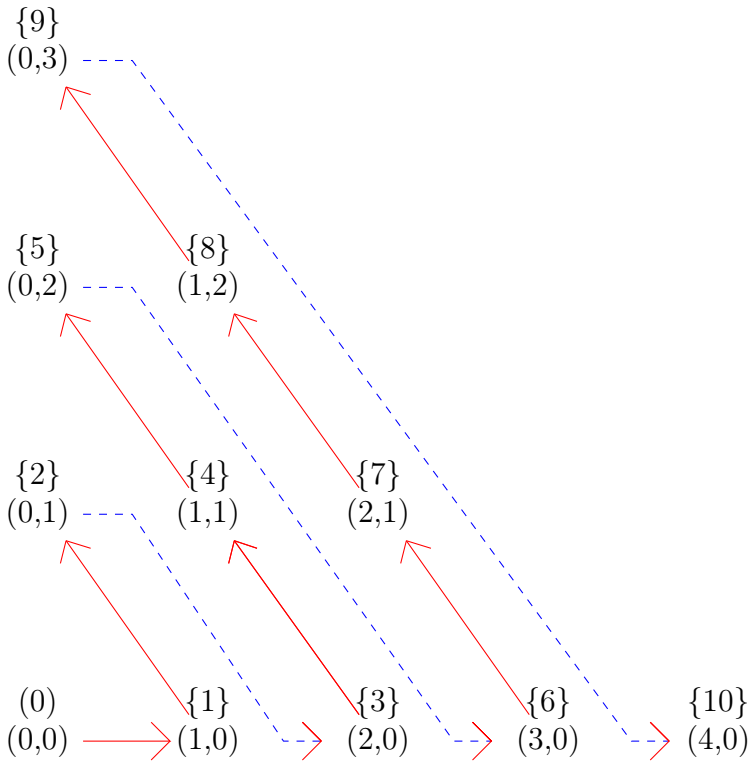
$$\alpha_p \in \mathcal{F}_p \text{ och } \beta_1^p, \beta_2^p, \ldots, \beta_p^p \in \mathcal{F}_1$$

som har följande egenskap:

α_p är en bijektion från $\mathbb{N}^p$ till $\mathbb{N}$ vars inverterade funktion uttrycks som

$$\alpha_p^{-1} = (\beta_1^p(x), \beta_2^p(x), \ldots, \beta_p^p(x)).$$

Bevis. Vi börjar med att konstruera α_2. för att göra det räknar vi upp talparen av heltal enligt diagrammet som följer:



Mer precist, vi numrerar paren (x, y) genom att följa diagonalerna så att summan av $x + y$ är konstant. Värdet av $\alpha_2(x, y)$ är exakt det nummer som representerar antalet föregångare av (x, y) i denna uppräkning. Ex. $\overset{\{7\}}{(2, 1)}$ är det åttonde talparet i uppräkningen! Alltså innan paret $(m + n, 0)$ finns det

$$1 + 2 + \ldots + (n + m) = \frac{1}{2}(n + m + 1)(n + m)$$

Talpar. Paret (m, n) ligger på samma diagonal och ligger exakt n steg över

det. Följaktligen:

$$\alpha(m,n) = \frac{1}{2}(n+m+1)(n+m)+n$$

□

Notera att α_2 är primitivt rekursiv och är större än eller lika stor som både n och m. Eftersom α_2 är bijektiv kan vi hitta tillbaka till n och m genom att använda någon av följande funktioner:

$$\beta_1^2(x) = \mu z \leq x[\exists t \leq x\ \alpha_2(z,t) = x]$$

och

$$\beta_2^2(x) = \mu z \leq x[\exists t \leq x\ \alpha_2(t,z) = x]$$

β_1^2 och β_2^2 är då också primitivt rekursiva.

Vi kan nu definiera

$$\alpha_3(x,y,z) = \alpha_2(x,\alpha_2(y,z))$$

och

$$\beta_1^3 = \beta_1^2, \beta_2^3 = \beta_1^2 \circ \beta_2^2 \text{ och } \beta_3^3 = \beta_2^2 \circ \beta_2^2$$

och mer generellt

$$\alpha_{p+1}(x_1,x_2,\ldots,x_p,x_{p+1}) = \alpha_p(x_1,x_2,\ldots,x_p,x_{p-1},\alpha_2(x_p,x_{p+1}));$$

$$\beta_1^{p+1} = \beta_1^p,\ \beta_2^{p+1} = \beta_2^p,\ldots,\ \beta_{p-1}^{p+1} = \beta_{p-1}^p,\ \beta_p^{p+1} = \beta_1^2 \circ \beta_p^p,\ \beta_{p+1}^{p+1} = \beta_2^2 \circ \beta_p^p,$$

Slutsatsen är att vi skriver

$$\alpha_1(x) = x$$

och

$$\beta_1^1(x) = x$$

och vi använder $\mathcal{S}$ för att benämna mängden av ändliga talföljder av heltal enligt

$$[\mathcal{S} = \mathcal{M}(\mathbb{N})]$$

5.3.3 Fibonacciserien

Leonardo Fibonacci levde i Pisa, Italien mellan ca 1170 och 1250. I skriften *Liber Abaci* introducerade han en talserie (vilken dock antagligen var känd betydligt tidigare), numera kallad *Fibonacciserien* som går ut på att ett visst tal i serien alltid är summan av de två föregående talen. Fibonacci utgick från genetik och framförallt i vilken takt kaniner förökade sig. (En liten svaghet var att han utgick ifrån att kaninerna hade evigt liv och det kan ju ifrågasättas :-) Senare har det dock visat sig att det finns många ställen i naturen där denna serie uppenbarar sig, som i spiralmönstren på kottar och ananas.

Forskning har visat att det antagligen inte är en slump att bladens placering runt en stjälk följer Fibonacciserien. Det är det effektivaste sättet att maximera solexponering, s.k. Phyllotaxis (grek. *phýllon* (blad) och *táxis* (ordning))! (Coxeter 1969) Seriens första tal är: $0, 1, 1, 2, 3, 5, 8, 13...$

Vi har alltså att:

$$\begin{cases} f(0) &= f(1) = 1 \\ f(n+2) &= f(n) + f(n+1), \end{cases}$$

Fibonacciserien är ett exempel på en primitivt rekursiv funktion där rekursionen sker simultant. Det är ju inte endast ett tal, utan två som ändras i varje steg. Om vi definierar *Fibonacci rekursivt = g(n)* kan vi skriva

$$g(n) = \alpha_2(f(n), f(n+1))$$

..och då ser vi att serien är primitivt rekursiv enligt:

$$\begin{cases} g(0) = & \alpha_2(1,1) \\ g(n+1) = & \alpha_2(\beta_2^2(g(n)), \beta_2^2(g(n)) + \beta_1^2(g(n))), \end{cases}$$

..och eftersom g är det är även f det då $f = \beta_1^2 \circ g$

Vi får anledning att återkomma till ett mer komplicerat exempel på detta i avsnitt 5.3.5.

Ytterligare ett intressant fenomen som följer med serien är att om vi dividerar ett tal med närmast föregående så konvergerar kvoten mot det gyllene snittet. Detta blir tydligt ju högre upp i talserien vi kommer. Det går att hitta flera

belagda exempel, men också märkliga sammanträffanden och serien har därför kallats magisk vilket jag dock med varm hand lämnar åt andra att utforska närmare..

5.3.4 Kodning av talföljder 2.

I presentationen av mitt arbete i kapitel 2 nämnde jag uttrycket *Gödelnumrering* som Kurt Gödel använde för att bevisa sina två ofullständighetssatser. För tydlighetens skull ska jag här i korthet beskriva vilken metod som skapades för just detta ändamål. Denna utgör det andra exemplet för kodning av talföljder som jag vill presentera. Definitionerna är hämtade ur professor *Dag Westerståhl*s (undervisningskompendium, Göteborgs universitet) *Rekursionsteori, En Introduktion* (1981) sid. 30–31. (Westerståhl 1981)

Gödelnumrering bygger på Aritmetikens fundamentalsats som säger att varje naturligt tal större än 1 kan skrivas som en produkt av en unik uppsättning primtal:

$$x = p_0^{x_0} \cdot p_1^{x_1} \ \cdot \ \ldots \ \cdot p_{k-1}^{x_{k-1}},$$

där $x_{k-1} > 0$ med nödvändighet.

För varje $k > 0$ definierar vi nu en k-ställig funktion $\langle x_0, \ldots, x_{k-1} \rangle^k$ enligt

1) $\langle x_0, \ldots, x_{k-1} \rangle^k = p_0^{x_0+1} \cdot p_1^{x_1+1} \ \cdot \ \ldots \ \cdot p_{k-1}^{x_{k-1}+1}$

   Vi sätter också

2) $\langle \rangle^0 = 1$

   Talet $\langle x_0, \ldots, x_{k-1} \rangle^k$ kallas sekvenstalet för k-tipeln $(x_0, \ldots, x_{k-1})$;

   Den tomma talföljden har alltså sekvenstalet 1.

3) Skälet till att vi ökar exponenterna med 1 är att vi vill att t.ex. $\langle 0, 0 \rangle^2$ ska vara skiljt från $\langle 0, 0, 0 \rangle^3$; i själva verket gäller nu:

4) $\langle x_0, \ldots, x_{k-1} \rangle^k = \langle y_0, \ldots, y_{m-1} \rangle^m \ \Leftrightarrow k = m$ och $\ x_i = y_i$ för $\ i < k$

   I fortsättningen utelämnar vi oftast det övre indexet och skriver helt enkelt $\langle x_0, \ldots, x_{k-1} \rangle$ - ställigheten framgår ändå.

   Mängden av talföljder kallas Sq, alltså:

5) $x \in Sq \Leftrightarrow$ det finns $k, x_0, \ldots, x_{k-1}$ sådana att $x = \langle x_0, \ldots, x_{k-1} \rangle$.

Trots att det finns oändligt många sekvenstal är det inte alla tal som kvalificerar sig. Talet 21 kan t.ex. inte vara ett sekvenstal!

$21 = 3^1 \cdot 7^1$ motsvarar ingen talföljd som uppfyller våra kriterier. Titta t.ex. på tidigare nämnda $\langle 0, 0, 0 \rangle$. Det kommer med nödvändighet att översättas i talet $2^{0+1} \cdot 3^{0+1} \cdot 5^{0+1} = 30$. D.v.s. Varje sekvenstal måste vara uppbyggt av minst ett primtal av varje storlek upp till det talets största primtal!

Vi definierar nu:

6) $(x)_i = \mu y_{y<x}(\sim p_i^{y+2} | x)$

och

$lg(x) = \mu y_{y<x}(\sim p_y | x)$ om $x > 0$ och $lg(0) = 1$ Då gäller att $lg(<>) = 0$

och

Om $x = \langle x_0, \ldots, x_{k-1} \rangle^k$ så $lg(x) = k$ och $(x)_i = x_i$ för $i < k$

Om x är ett sekvenstal är alltså lg(x) längden på följden som x är sekvenstalet för, och $(x)_i$ är det $(i+1)$:a elementet i den för $i < lg(x)$.

Märk att:

7) Om $sq(x)$ så $lg(x) < x$ och $(x)_i < x$ för $i < lg(x)$.

De tal vi får genom denna strategi är såklart mycket stora. Men metoden är vattentät. En talföljd kan på detta sätt översättas till ett unikt tal och det går att ur talet hitta tillbaka till talföljden!

Vi tittar på ett exempel.

8) $\langle 4, 0, 2 \rangle = 2^5 \cdot 3^1 \cdot 5^3 = 32 \cdot 3 \cdot 125 = 12\ \ 000$

$(x)_3 : 2$ för: $5^4 \nmid 12\ \ 000$

Längd : 3 för: $7 \nmid 12\ \ 000$

Slutsatsen är alltså: Primtalet 5 representeras av en $2 : a$ och $lg(x)$ är 3.

5.3.5 Ackermann-Péters funktion och den obegränsade μ-operatorn

Det finns funktioner som vi intuitivt ser är beräkningsbara men som vi anar inte har strukturen av en primitivt rekursiv funktion. Vi vill kunna utvidga begreppet beräkningsbarhet även till dem. Betrakta exempelvis följande funktion som definieras:

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

Funktionen kallas *Ackermann-Péters*[2] *funktion* (Dean och Naibo 2025) och är en variant av alla de funktioner som går under namnet *Ackermanns funktion*. Vi gör en tabell för några av de första resultaten så kan vi tydligt följa progressionen av hur svaren utvecklar sig:

[2]Rózsa Péter (1905–1977), ungersk matematiker, känd som de rekursiva funktionernas moder. Péter hade stort inflytande, men som både kvinna och judinna under andra världskriget blev antagligen hennes storslagna karriär ändå betydligt blygsammare än den kunde ha blivit.

m \ n	0	1	2	3	4	5	-------
0	1	2	3	4	5	6	$(n+1)\cdots$
1	2	3	4	5	6	7	$(n+2)\cdots$
2	3	5	7	9	11	13	$(2n+3)\cdots$
3	5	13	29	61	253	509	$(2^{n+3}-3)\cdots$
4	13	65533	$2^{65533}-3$	$*$	$*$	$*$	$(2^{\overbrace{2^{\cdot^{\cdot^{\cdot^{2}}}}}^{n+3}}-3)\cdots$
5	65533	$*$	$*$	$*$	$*$	$*$	-------
⋮	⋮	⋮	⋮	⋮	⋮	⋮	

Det är uppenbart att funktionen ger svar för samtliga ingångsvärden. För A(m,n) kommer vi att på andra raden hitta ett $A(x,y)$ med antingen $x < m$ någonstans i en tidigare rad, eller $x = m,\ y < n$ tidigare i samma rad. Via induktion på m:

$\mathcal{A}_m(-) := \mathcal{A}(m,-)$ är den primitiva funktionen:

Basfall: Via definition av $\mathcal{A}_0$

Efterföljande: Anta att $\mathcal{A}_m$ är primitivt rekursiv

$$\begin{cases} A_{m+1}(0) & = A(m+1,0)) = A(m,1) \\ A_{m+1}(n+1) & = \underbrace{A_m}_{\text{primitiv}}(A_{m+1}(n)) \\ & \text{(via induktionshypotesen)} \end{cases}$$

Att Ackermanns funktion inte är primitivt rekursiv i sin helhet ska vi dock snart bli varse!

5.3.6 Hur går vi vidare?

Vi har hittills förhållit oss till exempel på total och primitiv rekursion. Alla tal relationer, funktioner, talföljder och egenskaper har varit möjliga att härleda via de tre basfunktionerna samt komposition, rekursion och bunden minimalisation. Men med detta exempel blir det tydligt att vårt behov av definitioner inte är uttömt. Förutom totala rekursiva funktioner som inte är primitiva (liknande just Ackermann) har vi också exempel på funktioner som är primitivt rekursiva för vissa ingångsvärden men inte för andra, s.k. *Partiellt rekursiva funktioner*. Samlingsnamnet för alla är därför *Generellt rekursiva funktioner* eller rätt och slätt *Rekursiva funktioner*.

Totala rekursiva funktioner går från $\mathbb{N}^p \to \mathbb{N}$. Partiellt rekursiva funktioner går också från $\mathbb{N}^p$ till $\mathbb{N}$ men beteckningen blir istället $\mathbb{N}^P \rightharpoonup \mathbb{N}$.

Det är ofta beskrivet som att Ackermanns funktion kom till som ett försök till bevis för att det finns totala rekursiva funktioner som inte är primitiva men det är en grov förenkling. Historien är betydligt mer spännande än så.

David Hilbert höll ett omtalat föredrag 1924 som han kallade *"Om det oändliga"* (ty. *Über das unendliche*) där han diskuterade huruvida det var möjligt att via de ändliga matematiska verktygen beräkna det oändliga. I samma föredrag presenterade han dessutom sitt kända exempel *Hilberts hotell* (ty. *Hilberts hotel*) som är ett bevis på att Naturliga tal, Heltal, och Rationella tal, men inte reella tal har lika stora oändligheter. (Hilbert 1927)

Hilbert försökte förstå och skilja på "vanlig rekursion" (läs primitiv rekursion) och "transfinit rekursion" som leder inte bara till oändligheten utan till den oändlighet som inte är uppräkningsbar. Det han kom fram till utvecklades

sedan till just Ackermanns funktion som självklart skapades i samarbete med Wilhelm Ackermann!

För tydlighetens skull följer här återigen min tidigare definition och sedan exemplet $\mathcal{A}(2,1)$. Trots små ingångsvärden blir det tydligt att det dröjer innan det går att leverera ett svar. Och det är heller inte möjligt att med ett snabbt ögonkast komma fram till ens ett överslag om vad vad svaret ska bli!

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

$$\begin{aligned} A(2,1) &= A(1,A(2,0)) \\ &= A(1,A(1,1)) \\ &= A(1,A(0,A(1,0))) \\ &= A(1,A(0,A(0,1))) \\ &= A(1,A(0,2)) \\ &= A(1,3) \\ &= A(0,A(1,2)) \\ &= A(0,A(0,A(1,1))) \\ &= A(0,A(0,A(0,A(1,0)))) \\ &= A(0,A(0,A(0,A(0,1)))) \\ &= A(0,A(0,A(0,2))) \\ &= A(0,A(0,3)) \\ &= A(0,4) \\ &= 5 \end{aligned}$$

Funktionen fungerar för alla ingångsvärden men redan för $\mathcal{A}(4,2)$ blir svaret hela $2^{65536}-3$, ett tal med 19729 siffror. Varje steg går som jag visade lätt att följa men anledningen till att resultaten snabbt blir astronomiska (tyvärr mer än så..) är att det sker simultan rekursion vilket också kallas *inkapslad* (eng.

nested) funktion. Fibonacciserien som vi mötte i avsnitt 5.3.3 var ju också ett exempel på detta men skillnaden är att den i sin helhet är primitivt rekursiv och alltså växer kontrollerat upp till den "normala" oändligheten..

Eftersom Ackermann-talen blir så stora går de heller inte att skriva i den form vi är vana vid, utan man använder en särskild sorts system med pilar istället för exponenter för att överhuvudtaget få överblick, s.k. *Knuth's up arrow notation.*

Här följer nu ett skissartat bevis för att Ackermann-Péters funktion inte är primitivt rekursiv.

$$\mathcal{A} : \mathbb{N}^2 \to \mathbb{N}$$

för varje m:

$$\mathcal{A}_m = \mathcal{A}(m, -) : \mathbb{N} \to \mathbb{N}$$

(Den m:te raden i tabellen, där varje rad växer fortare och fortare.)

Definition 5.9. (för $f, g : \mathbb{N} \to \mathbb{N}$): $f \leq g$ om $\exists N, \forall n \geq N, g(n) \geq f(n)$ och $f < g$ (g dominerar f).

Lemma 5.10. *För varje primitivt rekursiv funktion f finns det en rad $\mathcal{A}_m$ så att $f \leq \mathcal{A}_m$.*

Slutledning 5.11. *$n \to \mathcal{A}(m, n)$ dominerar varje primitiv funktion.*

Slutledning 5.12. *Detta är inte primitivt beräkningsbart och $\mathcal{A}(m, n)$ är inte primitivt rekursiv.*

För att kunna definiera en funktion som är total men ej primitivt rekursiv behövs den obegränsade μ-operatorn, d.v.s. en funktion av typen:

$$g(x) = \mu z < y((f(x)) = 0)$$

Det finns inte utrymme för mig att här beskriva minimalisationsfunktionen som definierar Ackermanns funktion som totalt rekursiv. Jag ska istället ta ett mer familjärt exempel vilket är...

5.3.7 Division; ett exempel på en Partiellt rekursiv funktion som definieras via den obegränsade μ-operatorn

Divisionsfunktionen är på många sätt knepigare än de tre andra räknesätten eftersom den hela tiden befinner sig på gränsen mellan naturliga tal och rationella tal och då har jag ändå bortsett från beräkningar på heltalsdomänen $\mathbb{Z}$. (Se avsnitt 5.2.4) Men även om en division inte går jämnt ut finns det definitionsmetoder för funktioner som också beräknar restens storlek och på så sätt ser till att varje divisionsberäkning i det avseendet blir total.

Ett problem som dock inte går att bortse ifrån är att talet 0 i nämnaren ger en kvot som ej är definierad. Vi kan komma runt också det genom att bestämma den kvoten som likvärdig med 0 och tänka bort det för övrigt. Men ett stiligare sätt att definiera division är att göra det via den obegränsade μ-operatorn.

Vi känner beskrivningen av division som exempelvis;

För $a, b \in \mathbb{N}$:

$div(a, b)$ = det unika a så att $a = bq + r$ för ett q och ett r som uppfyller följande villkor: $0 \leq r < b$.

$$div(a,b) = \begin{cases} \text{som ovan}, & \text{om } \ b \neq 0 \\ 0 & \text{om } \ b = 0 \end{cases}$$

Men om vi tittar på följande idé:

Vilket är det största q vi kan hitta för att denna olikhet ska gälla, $q \cdot b \leq a$?

En relevant idé då likheten gäller om $r = 0$ och olikheten gäller om $r < b$.

Detta är dock inget minimalisationsproblem.

Vi skriver istället

$$b(q+1) = bq + b > bq + r = a$$

vilket är samma sak som

$$a < bq + b.$$

Vi kan alltså nu ställa oss frågan, vilket är det minsta $q \to (q+1)b > a$?

Och nu kan vi ge en definition av division som möter upp vårt krav på att den ska vara på formen:

$$g(x) = \mu z < y((f(x)) = 0)$$

Vi skriver att

$$div^*(a, b) = \text{det minsta } q \text{ så att } (q + 1)b > a$$

Funktionen som vi ska adaptera till μ-operatorn går från, $f : \mathbb{N}^3 \rightharpoonup \mathbb{N}$ eftersom vi har tre variabler och den är partiellt rekursiv och vi skriver:

$$f(a, b, q) = \mathit{MindreänEllerLikamed}\ (\mathit{mult}(\mathrm{s}(q), b), a))$$

eller

$$f(a, b, q) = \mathit{MindreänEllerLikamed} \bullet \mathbf{[}\mathit{mult} \bullet \mathbf{[}S \bullet \mathbf{[}P_3^3\mathbf{]}, P_2^3\mathbf{]}, P_1^3\mathbf{]]}$$

Men vi vill ju komma hit!

$$\mu f(a, b, q)) = \begin{cases} 1 & \text{if } \ (q + 1)b \leq a \\ 0 & \text{if } \ (q + 1)b > a \end{cases}$$

Där vi, när vi erhåller svaret 0, vet att det var ingående värde på q som var rätt svar. Erhåller vi oupphörligen svaret 1 utan förändring anar vi (eller ser t.o.m.) att funktionen för det värdet är odefinierad.

Nu ska vi alltså definiera $\rho^2 f : \mathbb{N}^2 \rightharpoonup \mathbb{N} = div^*$

och det blir helt enkelt;

$$div^* = \rho^2\mathbf{[}\mathit{MindreänEllerLikamed} \bullet \mathbf{[}\mathit{mult} \bullet \mathbf{[}S \bullet \mathbf{[}\ P_3^3\mathbf{]}, P_2^3\mathbf{]}, P_1^3\mathbf{]]}$$

Ok! Vi prövar:

$div^*(13, 6) =$

$$g(13, 6, 0) = \text{MindreänEllerLikamed } (\text{Mult}(\overbrace{1, 6}^{6}), 13) = 1 \neq 0$$

$$g(13, 6, 1) = \text{MindreänEllerLikamed } (\text{Mult}(\overbrace{2, 6}^{12}), 13) = 1 \neq 0$$

$$g(13, 6, 2) = \text{MindreänEllerLikamed } (\text{Mult}(\overbrace{3, 6}^{18}), 13) \neq 1 = 0$$

Svar: $div^*(13, 6) = 2$ som alltså är den största delaren. Vilken resten blir får vi räkna ut, om vi inte ser det, men det är en total funktion så den ställer inte till det för oss.

Men låt oss titta på vad som händer om vi har följande ingångsvärden:

$div^*(8, 0) =$

$$g(8, 0, 0) = \text{MindreänEllerLikamed } (\text{Mult}(\overbrace{1, 0}^{0}), 8) = 1 \neq 0$$

$$g(8, 0, 1) = \text{MindreänEllerLikamed } (\text{Mult}(\overbrace{2, 0}^{0}), 8) = 1 \neq 0$$

$$g(8, 0, 2) = \text{MindreänEllerLikamed } (\text{Mult}(\overbrace{3, 0}^{0}), 8) = 1 \neq 0$$

$$\vdots$$

i evighet eftersom ingångsvärdet $b = 0$ inte är definierat.

Vi kommer aldrig att hitta ett q och därvidlag har vi visat varför det inte är möjligt att ha 0 i nämnaren.

6 Avgörbarhetsproblemet

Vi ska gå tillbaka till Alan Turing som låg och vilade på sin äng för att nu äntligen visa varför det inte finns någon lösning på das Entscheidungsproblem. Jag kommer att redovisa svaret både med hjälp av Diagonalbeviset via Georg Cantor men också redovisa den variant som ligger närmast Turings eget bevis; Problemet med *Självreferens* som återfinns i tidigare nämnda; On Computable numbers.. §8 på sidorna 246–247 (Turing 1936)

6.1 Diagonalbeviset

Georg Cantor påstod följande: Om du gör en (fiktiv) lista med oändliga Reella tal i decimalform så kommer det att ur listan ändå gå att hitta ett tal som inte är på listan! Han använde sig av följande teknik: Byt första decimalen, n i första talet mot $n + 1$. Byt sedan andra decimalen n i listans andra tal mot $n + 1$. Fortsätt så med tredje, fjärde och... Till slut har vi ett helt nytt tal på diagonalen som skiljer sig från alla tal på listan, ett som alltså inte var uppräknat från början.

Lite mer ”formellt” kan vi skriva beviset som:

1. Anta en lista $\mathcal{L}$ som innehåller reella tal.
2. Diagonalbeviset ger att vi hittar ett reellt tal som inte finns på listan $\mathcal{L}$.
3. Ingen lista med reella tal $\mathcal{L}$ kan innehålla samtliga reella tal.

Jag skulle själv uttrycka det så att det ”gömmer sig” oändligt med reella tal i varje lista som ej är listade. Detta visar att det inte finns någon en-till-en-relation mellan listans numrering och de reella talen.

Vi säger att

1. Det finns ingen surjektion $\mathbb{N} \to \mathbb{R}$
2. Alltså: Det finns ingen bijektion $\mathbb{N} \to \mathbb{R}$.

Vi har alltså bevisat att det finns fler reella tal än naturliga tal och att deras oändligheter skiljer sig. De reella talen är överuppräkneliga.

Alan Turing angrep problemet från ett annat håll.

Omigen kan vi uttrycka det ”formellt” som:

1. Anta en beräkningsbar lista $\mathcal{L}^*$ av beräkningsbara reella tal
2. Diagonalbeviset ger att vi hittar ett beräkningsbart tal som inte finns på listan $\mathcal{L}^*$
3. Ingen beräkningsbar lista $\mathcal{L}^*$ innehåller alla beräkningsbara tal.
4. Alltså är en lista med *alla* beräkningsbara reella tal inte själv beräkningsbar.

Skillnaden här är att Turing tänker sig att varje beräkningsbart tal motsvaras av en algoritm som i sin tur motsvaras av en Turingmaskin. Och att de är uppräkneligt många är ju inte så konstigt, d.v.s det borde gå att beräkna listan med en Turingmaskin som är programmerad med rätt algoritm.

Men det går inte att hitta en Turingmaskin som kan beräkna den där listan.. eller en algoritm som kan avgöra om andra algoritmer är beräkningsbara..

6.2 Självreferens

Utifrån denna kunskap fortsatte Alan Turing med ett skarpare bevis mot att det skulle finnas en algoritm som kunde avslöja om en funktion, ett tal eller ett logiskt påstående hade ett svar utan att testa det först. Han bevisade det med hjälp av Självreferens. Här kommer ett exempel på det som är en lite enklare variant då vi inte behöver gå in i exakt hur Turingmaskinen fungerar på detaljnivå..Exemplet är hämtat från Stanford encyclopedia of philosophy: *Computability and complexity 2.4* (Immerman 2021).

Om vi tänker oss Turingmaskiner som Partiella funktioner från $\mathbb{N} \rightharpoonup \mathbb{N}$ och också att $\mathbb{N} = \{0, 1, 2, ..n\}$. Om $\mathcal{M}$ är en Turingmaskin så säger vi då att $\mathcal{M}$:s remsa innehåller talet n om $\mathcal{M}$:s remsa börjar med en binär representation av n.

Om $\mathcal{M}$ på input n alltid stannar och ger ett svar m skriver vi $\mathcal{M}(n) = m$ och menar att $\mathcal{M}$ kan utföra den beräkningen. Om $\mathcal{M}$ istället aldrig stannar eller spottar ut ett uppenbart icke accepterbart svar, säger vi att $\mathcal{M}(n) \nearrow$ och är odefinierat. Varje Turingmaskin går att associera med en partiell funktion $\mathcal{M} : \mathbb{N} \to \mathbb{N} \cup \{\nearrow\}$. Dock är en funktion $\mathcal{M}$ total om för alla $n \in \mathbb{N}, \mathcal{M}(n) \in \mathbb{N}$.

Nu kan vi formellt definiera vad det innebär för en mängd ($\mathcal{S}$) att vara rekursivt uppräkningsbar (r.e)(eng. *recursively enumerable*). Låt $S \subseteq \mathbb{N}$. Då är $\mathcal{S}$ r.e. omm det finns en Turingmaskin, $\mathcal{M}$ som uppräknar $\mathcal{S}$ enligt följande;

$$\mathcal{S} = \{\mathcal{M}(n) \mid n \in \mathbb{N}; \mathcal{M}(n) \neq \nearrow\}.$$

Vi kan beskriva det som att $\mathcal{M}$ på indata n som utdata får exakt den mängd m som ingår i $\mathcal{S}$. Sålunda är $\mathcal{S}$ r.e. om den kan tolkas av en unik Turingmaskin.

Anta att $\mathcal{S}$ är r.e. och kan beskrivas enligt ovan. Nu kan vi beskriva en annan Turingmaskin som vi kallar $\mathcal{P}$ som på indata n kör $\mathcal{M}$:s program på indata i i en sorts *Round-Robin*-process [3] ända tills $\mathcal{M}$ slutligen matar ut n. Om detta händer stannar $\mathcal{P}$ och och matar ut 1, d.v.s. $\mathcal{P}(n) = 1$. Om $n \notin \mathcal{S}$ kommer aldrig $\mathcal{M}$ att mata ut n och $\mathcal{P}$ kommer aldrig att stanna d.v.s. $\mathcal{P}(n) \nearrow$. Vi kan kalla $\mathcal{P}$:s mängd för ($\mathcal{L}$) med egenskapen att vara *ibland-avgörbar* (fr. *eng. semi-decidable*). Och vi säger att ibland-avgörbar är synonymt med att vara rekursivt uppräkningsbar.

Låt notationen $\mathcal{P}(n) \downarrow$ betyda att Turingmaskinen $\mathcal{P}$ slutligen stannar. Definiera nu $\mathcal{L}(P)$, mängden accepterad av $\mathcal{P}$, som de tal n som gör att $\mathcal{P}$ stannar på indata n:

$$\mathcal{L}(\mathcal{P}) = \{n \mid \mathcal{P}(n) \downarrow\}$$

Vi får då självklart att $\mathcal{S} = \mathcal{L}(\mathcal{P})$ om $\mathcal{S}$ är r.e och lika självklart att om $\mathcal{S}$ accepteras av Turingmaskinen $\mathcal{P}$ så är $\mathcal{S}$ r.e!

Vi säger att en mängd är *avgörbar* omm det finns en Turingmaskin $\mathcal{P}$ som bestämmer huruvida $n \in \mathcal{S}$ eller $n \notin \mathcal{S}$ för alla $n \in \mathbb{N}$. Om ja betecknas som 1 och nej som 0 kommer alltså $\mathcal{P}(n) = 1$ betyda att $n \in \mathcal{S}$ och $\mathcal{P}(n) = 0$ betyda att $n \notin \mathcal{S}$. Synonyma med avgörbar är så klart *beräkningsbar*, *möjlig att lösa* och *rekursiv*.

Komplementet till $\mathcal{S} \subseteq \mathbb{N}$; $\mathbb{N} - \mathcal{S}$ är alla tal som inte är i $\mathcal{S}$ Vi säger då att mängden $\mathcal{S}$ är *komplementärt rekursivt uppräkningsbar*, co-r.e. omm komplementet är rekursivt uppräkningsbart, r.e.

Om en mängd $\mathcal{S}$ är r.e. och co-r.e. kan vi helt enkelt göra en lista med två kolumner med de olika alternativen och när vi undrar kan vi helt enkelt scanna listan och kolla i vilken kolumn vårt kontrollerade n dyker upp. Faktiskt är det så att en mängd är rekursiv omm den är både rekursivt uppräkningsbar och dess komplement också är det!

Turing frågade sig om alla mängder var avgörbara och besvarade själv frågan med ett besämt nej! eftersom varje avgörbar mängd motsvaras av exakt en av

[3] *Round Robin* är en sorts ”rättvis” process där alla indata körs simultant tills utdata dyker upp för varje enskilt värde, om de dyker upp...

alla uppräkningsbart många Turingmaskiner och det tyvärr finns överuppräkneligt många delmängder av $\mathbb{N}$.

För att bevisa detta skapade han en oavgörbar mängd användades sig av Diagonalbeviset.

Han hittade på en mängd:

$$\mathcal{K} = \{n \mid \mathcal{M}_n(n)\}$$

$\mathcal{K}$ består av de Turingmaskiner som stannar på indata av: **Sitt eget program**.

K är r.e. Anta nu att K också är co-r.e. och låt d vara numret på en Turingmaskin som accepterar komplementet till $\mathcal{K}$. Det blir för alla n

$$n \notin K = \mathcal{M}_d(n) \downarrow$$

Men vad händer om vi byter ut n mot d i ekvationen ovan?

$$d \notin K = \mathcal{M}_d(d) \downarrow$$

Från början sade vi dock att definitionen av K var

$$d \in K = \mathcal{M}_d(d) \downarrow .$$

Vi hamnar nu i situationen att

$$d \in \mathcal{K} \Leftrightarrow d \notin \mathcal{K}$$

som är en motsägelse. Vårt antagande att $\mathcal{K}$ är co-r.e. är falskt. Det följer här att $\mathcal{K}$ inte är rekursiv och att det inte går att bevisa om en Turingmaskin stannar på sitt eget program som indata. Faktum är att vi också då bevisat att det inte går att bevisa om en Turingmaskin kommer stanna på någon indata alls. Inte utan att köra programmet på den aktuella Turingmaskinen och se vad som händer. Das Entscheidungsproblem är verkligen olösbart.

David Hilbert blev nog ändå aldrig helt övertygad. Han säger själv i en radiointervju år 1930 som svar på det gamla uttrycket ”Ignoramus et Ignorambus”

(”Vi vet inte och vi kommer aldrig att veta”), myntat år 1872 av den tyske filosofen *Emil du Bois-Reymond*: ”Wir müssen wissen, wir werden wissen” (”Vi måste veta, vi kommer att veta”). Och han var envis ända in i det sista för citatet går att återfinna på hans gravsten i Göttingen, staden där han verkade som älskad lärare och professor i hela sitt liv!

7 Alonzo Church

Innan mitt avslutande kapitel i denna uppsats där jag redogör för den officiella definitionen av beräkningsbarhet ska jag i korthet nämna *Alonzo Church* (1903–1995) och *λ-kalkylen*.

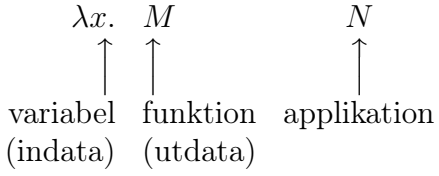
Både Alonzo Church och läsarna får förlåta mig att detta kapitel blir så rumphugget. λ-kalkylen är inte bara viktig för definitionen av beräkningsbarhet utan ligger också till grund för de flesta stora datorspråken som *LISP, Haskell* och *Javaskript*. Church själv var också skapare av den moderna logiken som vi känner den idag och än viktigare; han var Alan Turings lärare på *Princeton university* åren före 2:a världskriget. Det är alltså så att de båda ovedersägligen påverkat varandra och att båda kan ta åt sig äran av att ha lagt grunden till den explosionsartade utveckling av datorvetenskapen vi sett i modern tid.

Men viktigast av allt, det är helt klart att definitionen av beräkningsbarhet vilar på bådas axlar.

Även Alonzo Church hade reagerat på Hilberts idé om en algoritm som kunde avgöra om ett påstående i första ordningens logik var sann eller inte, men hans svar var att uppfinna en elegant variant av funktioner, λ-kalkylen med vars hjälp han kom till samma slutsats som Alan Turing, att det inte var möjligt.

7.1 Hur fungerar den?

λ-kalkylen består av endast 3 komponenter:



Ett mycket enkelt exempel är:

$$\lambda x(x^2 - 2x + 5)2 = 2^2 - 2 \cdot 2 + 5 = 5$$

Detta kan ju tyckas begränsat och framförallt intill förväxling likt hur vi räknar funktioner till vardags, men eftersom en funktion i λ-kalkyl kan ta in en applikation och som utdata ge en ny funktion kan proceduren pågå på ett strukturerat sätt oavsett mängd indata.

Ett exempel på detta som alltså blir betydligt intressantare är:

$$\begin{aligned} S(S(x)) &= (\lambda f.(\lambda x.f(fx)))(\lambda n.n + 1) \\ &= (\lambda x.(\lambda n.n + 1)((\lambda n.n + 1)x)) \\ &= (\lambda x.(\lambda n.n + 1)(x + 1)) \\ &= \lambda x.(x + 1) + 1 \\ &= \lambda x.x + 2 \end{aligned}$$

Proceduren att på detta sätt ersätta en variabel med en ny funktion alternativt ett tal kallas *β-omvandling* (eng. β-reduction)

7.2 Vad har detta med beräkningsbarhet att göra?

Det är viktigt att vara införstådd med att denna typ av funktionsbeskrivning skiljer sig från den vi är vana vid. Om vi i vår vanliga matematik ser funktioner utifrån ett *Extensionellt* perspektiv där två funktioner med samma indata och utdata sägs vara likvärdiga, som en mängd av par, kräver λ-kalkylen att funktionerna ska var exakt samma i varje möjlig värld. Varje funktion i λ-kalkylen är uppbyggd av regler och två uträkningar är alltså inte med nödvändighet likvärdiga även om in- och utdata är identiska. Vi säger att två funktioner måste vara *Intensionellt* lika för att räknas som likvärdiga. Vilket i praktiken blir att varje funktionsbeskrivning är exklusiv.

Se detta enkla men tydliga exempel:

$$\begin{aligned} \text{ADDERA ETT} &:= \lambda x.x + 1 \\ \text{ADDERA TVÅ OCH SUBTRAHERA ETT} &:= \lambda x[[x + 2] - 1] \end{aligned}$$

I extensionell mening identiska uträkningar, i intensionell mening mer tveksamt...

Vi kan tänka oss att λ-kalkylens krav på att se funktioner som regler med exklusiviteten att två funktioner aldrig kan likställas med varandra är det som gör den till;

1) en perfekt definition av beräkningsbarhet (likheten med rekursionsteorin

är slående, vilket även Church själv påstod);

2) en mycket säker grund för programmering.

(Alama och Korbmacher 2024) Stanford Encyclopedia of Philosophy.

8 Church-Turings hypotes

Vi har nu äntligen kommit fram till det som världen (hittills) lyckats enas om som en definition av beräkningsbarhet; *Church-Turings hypotes*. Även detta kapitel är hämtat ur Stanford Encyclopedia of Philosophy (Copeland 2026)

Hypotesen utgår från en konceptuell beskrivning av vad som kan kallas en *effektiv* eller *systematisk* eller *mekanisk* metod, M, för att kunna lösa logiska, matematiska eller datormässiga problem. Och då menar vi lösa alla problem som går att lösa.

Metoden M är i detta sammanhang effektiv, systematisk eller mekanisk endast om:

1. M utgörs av ett begränsat antal exakta instruktioner (där varje instruktion i sin tur uttrycks i ett ändligt antal symboler);
2. M kommer, om det utförs felfritt, att producera önskat resultat i ett begränsat antal steg;
3. M kan (i praktiken eller i princip) utföras av en människa som endast har papper och penna att tillgå;
4. M kräver ingen insikt, intuition eller uppfinningsrikedom hos den person som utför beräkningen.

Trots att ovanstående är en mycket informell beskrivning lyckades både Alonzo Church och Alan Turing [4] uttrycka varsin definition av M formellt och exakt. Deras definitioner visade sig vara ekvivalenta i så måtto att båda lyckades pricka samma mängd S av funktioner, nämligen den mängd som består av alla kända beräkningsbara funktioner. Och, ännu bättre, det visade sig att det i S inte heller fanns några funktioner vars värde INTE kunde erhållas med den metod som uppfyller ovanstående villkor för effektivitet. Därför är påståendet "Det finns en effektiv metod för att erhålla värden för funktionen f" utbytbart mot "f är en medlem av S".

8.1 Alan Turings definition

Alan Turings definition lyder:

Tal som är beräkningsbara kan beräknas med hjälp av en Turingmaskin. (Med risk för att bli långrandig påminner jag om att Turings påstående självklart gäller allt som är beräkningsbart, funktioner, mängder, booleska operationer, tal etc. Det enda som inte går att beräkna är de reella talen som inte går att med en Turingmaskin generera siffra för siffra. För, som tidigare visats, Turingmaskinerna är ju uppräkneligt många, medan de reella talen är överuppräkneliga så det är ju inte konstigt att det inte finns Turingmaskiner till alla de talen.)

Han tillägger; ... Beräkningsbara tal inkluderar alla tal som naturligt kan betraktas som beräkningsbara.

Och han betonar att allt som en Turingmaskin kan utföra kan också utföras av en *"räknande människa"* (eng. *computer*).

Men det som är det unika med Turings val av definition, det är just att Turingmaskinen är rent mekanisk, en strikt maskinell metod för beräkningar.

8.2 Alonzo Church's definition

Alonzo Church gav egentligen två definitioner som han ansåg likvärdiga. Även dessa gäller samtliga typer av beräkningsbarhet.

1. Effektiv beräkningsbarhet är samma sak som beräkningsbar med λ-kalkyl.

[4] Oberoende av varandra

2. Effektiv beräkningsbarhet är detsamma som definierad och beräkningsbar via rekursion

Det är av vikt att påminna om att den sista definitionen kom ur ett samarbete mellan Church, Kleene och den amerikanske logikern *John Barkley Rosser* (1907–1989). Rosser var som elev till Church även involverad i arbetet med λ-kalkylen.

8.3 Sammanfattning

Vad som är beräkningsbart går alltså att definiera men inte att bevisa rent matematiskt. Church-Turings hypotes är inte heller ett bevis. Dess styrka ligger i att de tre definitioner som mitt arbete handlat om och som i sublimerad form uttryckts av Alan Turing och Alonzo Church har visat sig extensionellt ekvivalenta. Det kan tyckas som att beräkningsbarhetens vara eller inte vara alltså vilar på en svag grund. Men om vi betänker att tesen stått sig under snart 100 år utan att någon kunnat visa på motsatsen är det svårt att inte vara hoppfull.

Det finns ingen känd beräkning som går att utföra med en Turingmaskin som inte även λ-kalkylen klarar av. Eller som är omöjlig att definiera via rekursion. Det finns däremot en diskussion inom matematikskråret vilken definition som är elegantast eller tydligast. Alonzo Church själv såg en fördel i att definiera beräkningsbarhet med ”beräkningsbar med en Turingmaskin” eftersom idén är intuitivt lättare att förstå, framförallt för en lekman. Alan Turing måste ha varit stolt över Church’s uttalande och än mer så över Kurt Gödels tre förtydliganden i samma fråga.

Gödel beskriver Turings analys av beräkningsbarhet som *. . . mycket tillfredsställande och korrekt. . . utan rimligt tvivel. . .*

. . . ingen har före Turing tänkt i termer av det stringenta begreppet mekanisk procedur, vilket givit oss rätt perspektiv. Och *. . . inte bara korrekt utan begreppet “möjligt att utföra med hjälp av en Turingmaskin” är också unikt.* (Wang 1996, s. 203) Men även den modeste Alan Turing tillstod själv redan 1937 att hans upptäckt kanske var lite mer övertygande.

Church-Turings hypotes bör inte heller missuppfattas som exklusivt giltig för dessa tre definitioner. Varje definition som är extensiellt ekvivalent med dem kan sägas ingå i tesen.

9 Charles Babbage och Ada Lovelace

Det finns två fantastiska personer som inte fått plats i mitt arbete trots att de absolut förtjänar det. Den brittiska matematikern *Charles Babbage* (1791–1871) och den brittiska matematikern *Ada Lovelace* (1815–1852).[5]

Babbage gjorde bland mycket annat ritningar på två matematiska maskiner som han kallade *Den Analytiska maskinen* och *Differensmaskinen*. Ritningarna finns att beskåda på *London Science Museum* och 1991 byggdes dessutom en modell av Differensmaskinen vilket visade att ritningarna är helt korrekta och att maskinen skulle ha fungerat redan 1791! Problemen som uppstod då berodde på vibrationer orsakade av kugghjulen samt att han blev tveksam kring konstruktionen medan han höll på. Det sägs att kostnaden för bygget uppgick till £17000 ...

I arbetet med Analysmaskinen hade Babbage Ada Lovelace till sin hjälp. De började arbeta tillsammans då hon 1842–1843 översatte en text av den italienska ingenjören *Luigi Menabrea* som handlade om just en analytisk matematisk maskin. Men arbetet fördjupades och Ada kompletterade översättningen med egna noter och tankar. Där återfinns även vad många matematiker anser vara det första datorprogrammet, d.v.s. en algoritm som kan ”köras” i den Analytiska maskinen. Om den Analytiska maskinen anses vara den första datormaskinen så anses Ada Lovelace vara den första datorprogrammeraren. Hon har också fått ge namn åt ett datorprogram skapat på 1970–talet som fortfarande används och just heter *Ada*. Lovelace hade många idéer kring maskinens roll i samhället som vi fortfarande idag brottas med. Vem bestämmer, människa eller maskin? och hur fungerar hjärnan egentligen? Går den att förstå rent matematiskt. Det är ingen överdrift att säga att hon var långt före sin tid!

10 Slutord

Jag har försökt att ge en bild av hur vi definierar det vi varje dag tar för givet; att det går att lita på matematiska uträkningar! Om vi räknat rätt..

Babbage, Lovelace, Fibonacci, Hilbert, Gödel, Kleene, Neumann, Ackerman, Péter, Church, Rosser... och alla andra, icke här nämnda, matematiker som

[5] Ada Lovelace var dessutom *Lord Byron´s* dotter men han kan knappast ta åt sig äran för dotterns framgångar.

bidragit till denna erövring, vad skulle vi gjort utan er? Men kanske ändå att det är rättvist att framförallt undra vad vi skulle gjort utan Alan Turing och inte att förglömma;

Vad skulle vi gjort utan Turingmaskiner?

Innehåll

Referenser

Alama, Jesse och Johannes Korbmacher (2024). ”The Lambda Calculus”. I: *The Stanford Encyclopedia of Philosophy*. Utg. av Edward N. Zalta och Uri Nodelman. Winter 2024. Metaphysics Research Lab, Stanford University.

Copeland, B. Jack (2026). ”The Church-Turing Thesis”. I: *The Stanford Encyclopedia of Philosophy*. Utg. av Edward N. Zalta och Uri Nodelman. Spring 2026. Metaphysics Research Lab, Stanford University.

Cori, René och Daniel Lascar (2000). *Mathematical logic. A course with exercises. Part I: Propositional calculus, Boolean algebras, predicate calculus, completeness theorems. Transl. from the French by Donald H. Pelletier*. English. Oxford: Oxford University Press. ISBN: 0-19-850049-1.

Coxeter, H. S. M. (1969). *Introduction to geometry. 2nd ed.* English. New York etc.: John Wiley and Sons, Inc. XVI, 469 p. (1969).

Cutland, Nigel (1980). *Computability. An introduction to recursive function theory*. English. Cambridge University Press.

Dean, Walter och Alberto Naibo (2025). ”Recursive Functions”. I: *The Stanford Encyclopedia of Philosophy*. Utg. av Edward N. Zalta och Uri Nodelman. Summer 2025. Metaphysics Research Lab, Stanford University.

Gödel, K. (1931). ”Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme. I.” German. I: *Monatsh. Math. Phys.* 38, s. 173–198. ISSN: 0026-9255. DOI: 10.1007/BF01700692.

Hilbert, D. (1917). ”Axiomatisches Denken. Vortrag gehalten in der Schweizerischen mathematischen Gesellschaft, am 11. Sept. 1917 in Zürich.” German. I: *Math. Ann.* 78, s. 405–415. ISSN: 0025-5831. DOI: 10.1007/BF01457115. URL: https://eudml.org/doc/158776.

– (1927). ”Über das Unendliche.” German. I: *Jahresber. Dtsch. Math.-Ver.* 36, s. 201–215. ISSN: 0012-0456. URL: https://eudml.org/doc/145764.

Hodges, A. (1983). *Alan Turing: The Enigma*. Archivo Elena Asins: Serie Biblioteca personal. Simon och Schuster. ISBN: 9780671492076. URL: https://books.google.se/books?id=BlPZ3sHHWpMC.

Immerman, Neil (2021). "Computability and Complexity". I: *The Stanford Encyclopedia of Philosophy*. Utg. av Edward N. Zalta. Winter 2021. Metaphysics Research Lab, Stanford University.

Kleene, S. C. (1936). "General recursive functions of natural numbers". English. I: *Math. Ann.* 112, s. 727–742. ISSN: 0025-5831. DOI: 10.1007/BF01565439. URL: https://eudml.org/doc/159849.

Turing, A. M. (1936). "On computable numbers, with an application to the Entscheidungsproblem." English. I: *Proc. Lond. Math. Soc. (2)* 42, s. 230–265. ISSN: 0024-6115. DOI: 10.1112/plms/s2-42.1.230.

Wang, Hao (1996). *A logical journey. From Gödel to philosophy. Final editing and with an addition to the preface by Palle Yourgrau and Leigh Cauman*. English. Cambridge, MA: MIT Press. ISBN: 0-262-23189-1.

Westerståhl, Dag (1981). *Rekursionsteori. En introduktion*. Tekn. rapport. Göteborgs universitet.