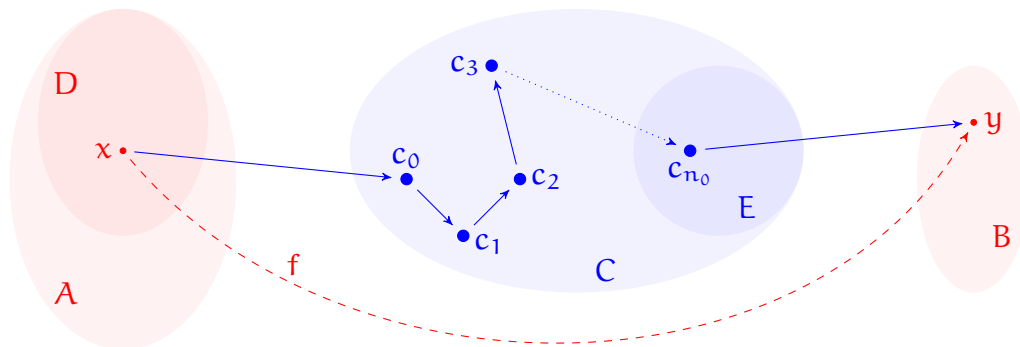


COMPUTONOMICON



Izračunljivost za računare

Vedran Čačić

8. veljače 2019.

Sadržaj

Predgovor	v
a. O knjizi	vi
b. Motivacija	vii
c. Opća i univerzalna izračunljivost	ix
d. Pregled po poglavljima	x
1. RAM-izračunljivost	1
1.1. Pojam izračunljive funkcije	1
1.1.1. Parcijalnost	4
1.2. Notacija	6
1.3. RAM-stroj i RAM-program	8
1.3.1. Skup Comp	11
1.3.2. Primjeri RAM-programa	13
1.4. Makro-izračunljivost	15
1.4.1. Spljoštenje	18
1.4.2. Ekvivalencija RAM-programa i makro-programa	21
1.4.3. Primjeri makroa	22
1.4.4. Funkcijski makro	24
2. Rekurzivne funkcije	27
2.1. Kompozicija	29
2.1.1. RAM-izračunljivost kompozicije	32
2.2. Primitivna rekurzija	33
2.2.1. Primitivno rekurzivne funkcije	36
2.2.2. Primjeri primitivno rekurzivnih funkcija i relacija	38
2.3. Minimizacija	41
2.3.1. Kompajler za funkcijski jezik	43
2.4. Tehnike za rad s (primitivno) rekurzivnim funkcijama	46
2.4.1. Višestruke operacije	48
2.4.2. Teorem o grananju za totalne funkcije	50
2.4.3. Ograničene sume, produkti i brojenje	52
2.4.4. Ograničena kvantifikacija i minimizacija	55

3. Univerzalna izračunljivost	58
3.1. Kodiranje	58
3.1.1. Kodiranje konačnih nizova	60
3.1.2. Potrebni rezultati iz teorije brojeva	63
3.1.3. Rastav na prim-faktore	65
3.2. Funkcije definirane pomoću operatora povijesti	67
3.2.1. Primitivna rekurzija kroz prostor i vrijeme	69
3.2.2. Primjeri korištenja rekurzije s poviješću	72
3.3. Kodiranje RAM-modela izračunljivosti	74
3.3.1. Kodiranje stanja registara i RAM-konfiguracija	77
3.3.2. Kodiranje RAM-izračunavanja	80
3.3.3. Prepoznavanje završne konfiguracije i čitanje rezultata	82
3.4. Kleenejev teorem o normalnoj formi	85
3.4.1. Univerzalne funkcije comp_k	88
3.4.2. Indeksi izračunljivih funkcija	91
3.4.3. Parcijalno rekurzivna verzija teorema o grananju	94
4. Turing-izračunljivost	96
4.1. Zašto nam treba još jedan model?	96
4.1.1. Računanje bez računala	98
4.1.2. Primjer Turing-izračunavanja	101
4.2. Prateće funkcije jezičnih funkcija	103
4.2.1. Kodiranje znakova i riječi	104
4.2.2. Parcijalna rekurzivnost jezičnih funkcija	107
4.2.3. Kodiranje funkcije prijelaza i Turing-izračunavanja	110
4.3. Pretvorba RAM-stroja u Turingov stroj	113
4.3.1. Inicijalizacija RAM-stroja	114
4.3.2. Registri kao tragovi trake	117
4.3.3. Prepoznavanje završne konfiguracije	120
4.3.4. Dekodiranje izlazne vrijednosti	122
4.3.5. Objedinjavanje	125
4.4. Turing-izračunljivost brojevni funkcija	127
4.4.1. Izračunljivost jezika	129
4.4.2. Turing-izračunljivost višemjesnih funkcija	132
4.4.3. Funkcije bin^k — binarno kodiranje	134
4.4.4. Standardna biblioteka za Σ_β^*	137

5. Neodlučivost	140
5.1. Church–Turingova teza	142
5.1.1. Fiksiranje pojma algoritma	145
5.2. Neizračunljive funkcije	147
5.2.1. Svođenje i problemi zaustavljanja	149
5.3. Neodlučivost u logici prvog reda	152
5.3.1. Reprezentacija RAM-konfiguracija formulama prvog reda	154
5.3.2. Zaključivanje kao zaustavljanje	156
5.3.3. Formule prvog reda kao formalni jezik	159
5.3.4. Kodiranje formula prvog reda	161
5.3.5. Formalna izreka Churchovog teorema	164
5.4. Prema Gödelovom prvom teoremu nepotpunosti	166
5.4.1. Problemi u skici dokaza Gödelovog teorema	169
6. Metaprogramiranje	171
6.1. Specijalizacija	171
6.1.1. Teorem o parametru	173
6.1.2. Teorem o parametrima	175
6.2. Opće rekurzije	178
6.2.1. Dijagonalna funkcija	181
6.2.2. Ackermannova funkcija	184
6.2.3. Rekurzivnost Ackermannove funkcije	187
6.3. Ekvivalentnost indeksa	189
6.4. Invarijantnost	192
6.4.1. Riceov teorem	195
6.4.2. Sintaksna i semantička svojstva RAM-programa	197
7. Rekurzivna prebrojivost	199
7.1. Karakterizacija preko projekcije	202
7.1.1. Kontrakcija relacije	203
7.1.2. Kontrakcija kvantifikatora	206
7.2. Teorem o grafu za parcijalne funkcije	207
7.2.1. Primjene teorema o grafu	210
7.3. Postov teorem	213
7.4. Rekurzivno prebrojivi jezici	216

Predgovor

Već nekoliko godina držim, na Matematičkom odsjeku Prirodoslovno-matematičkog fakulteta u Zagrebu, kolegij Izračunljivost. Taj kolegij je izvorno uveden kao prirodni nastavak kolegija Matematička logika (čijim je dijelom dugo bio), s ciljem dokaza Churchovog teorema o neodlučivosti logike prvog reda. Tako je nastala knjiga [16], izdavanjem iz knjige [17], koja je bila prvenstveno namijenjena studentima teorijske matematike koji žele produbiti svoje znanje o matematičkoj logici.

U međuvremenu, zbog raznih okolnosti, kolegij su počeli upisivati mahom studenti računarstva, kojima pojam algoritma predstavlja puno općenitiji i intuitivno bliži pojam od onog potrebnog da bi se dokazao Churchov teorem. U suvremenom svijetu okruženi smo računalima raznih vrsta, često ih programiramo da bismo ih prilagodili svojim potrebama, i algoritamske sustave više ne doživljavamo kao nešto apstraktno. Pojam izračunljive funkcije (funkcije implementirane u nekom programskom jeziku) počeo je već u umovima studenata računarstva istiskivati skupovnoteorijsku ideju uređene trojke (domena, kodomena, graf) kao asocijaciju na pojam „funkcija”. Rekurzija više nije egzotična matematička konstrukcija, već sasvim uobičajen alat u repertoaru gotovo svakog programera. Jezici više nisu reprezentirani črkarijama na papiru ili otiscima na traci (kao što su bili u Turingovo vrijeme), već tekstnim datotekama, nizovima bajtova u određenom *encodingu*, koji se sasvim prirodno obrađuju programskim alatima. Strukture više nisu dijagrami matematičkih simbola povezanih strelicama, nego memorijski blokovi objekata povezanih pokazivačima ili referencama. *Halting problem* nije više nešto maglovito i daleko od svakodnevnog iskustva: ta svi smo doživjeli da se računalo privremeno smrzne, i bili u nedoumici koliko dugo čekati prije nego što dobijemo nekakav odziv, ili zaključimo da se permanentno smrznulo i ne preostaje nam drugo doli posegnuti za gumbom za ponovo pokretanje.

U tom svjetlu, počeli su se pokazivati određeni nedostaci knjige [16]. Zahvaljujući njenom pokušaju da izgradi matematičku intuiciju, potpuno zaboravljajući ili čak namjerno potiskujući intuiciju koju računarci već imaju o tim pojmovima, konačni učinak za većinu studenata bio je vrlo sličan onom koji je primijetio Eric Mazur [8] u svojoj nastavi fizike:

Professor, how should I answer these questions: according to what you taught me, or according to the way I usually think about these things?




Nažalost, znao sam i ja dobivati takva pitanja. Ili sam jednostavno uočio da studenti

pri programiranju koriste jedan mentalni model, a pri rješavanju zadataka sasvim drugačiji. I naravno, pritom čine puno više početničkih pogrešaka — jer taj drugi model izgrađuju tek nekoliko mjeseci, dok prvi izgrađuju desetak godina.

Ova knjiga pokušaj je ispravljanja tog dojma. **Ne postoje dva svijeta**, svijet modernog računarstva i svijet klasične teorije izračunljivosti. To je jedan te isti svijet, samo što je u matematičkom modelu pojednostavljen (slično kao i model njutnovske mehanike: vakuum, linearno trenje, materijalne točke, ...) — ali svi bitni pojmovi teorijskog računarstva se u njemu mogu modelirati, svaka intuicija se može validirati, i svaki fenomen se može uočiti. Ako ste „računarac u duši”, sve potrebne ideje već imate. I najvažnije, sistematizacija i razumijevanje koje iz toga proizlaze su nezamjenjivi.

Što ako *niste* računarac u duši? Knjiga [16] je fantastična za izgradnju matematičke intuicije. Praktički jedini njen nedostatak je invalidacija računarske intuicije — ako tu intuiciju nemate, nedostatka nema. Izuzetno sam se trudio održati *backward compatibility*, tako da čak možete neke pojmove naučiti otamo, a neke odavdje.

a. O knjizi

Iako je knjigu sasvim moguće isprintati na papir i šarati po njoj, prvenstveno je namijenjena digitalnom čitanju. Zato ima puno referenci (označenih posebnom bojom), na svaku od kojih je moguće kliknuti da bi se vidjelo na što se odnosi. Ako koristite „pravi” PDF-čitač (za razliku od ovih što dođu s *browserima*), možete se i vratiti tipkom , ili kombinacijom  + . Pravi čitač omogućuje vam i navigaciju kroz naslove, bolji *rendering*, označavanje omiljenih stranica i još neke stvari zbog kojih se svakako isplati instalirati ga. Moja preporuka je SumatraPDF na Windowsima, Okular na Linuxu, te Document Viewer na Androidu. Naravno, ako imate dovoljno RAM-a, Adobe Reader je također opcija. Ako baš morate čitati u internetskom pregledniku, čujem da Firefox ima relativno dobar *plugin*.

Knjiga je pisana u L^AT_EXu, klasa KOMA-Script book, koristeći internetsku uslugu *Overleaf* koja je vjerojatno najbolji besplatni način za produkciju visoko zahtjevnih dokumenata „u oblaku”. Font je Knuthov Concrete iz serije *Concrete Mathematics*, a za matematiku Zapfov $\mathcal{A}\mathcal{M}\mathcal{S}$ Euler. Ako vas zanima išta detaljnije o produkciji knjige, možete mi pisati na veky@math.hr — ili pogledati u (ne sasvim ažuran) repozitorij na github.com/vedgar/izr. Naravno, ako uočite bilo kakvu grešku u knjizi, ili smatrate da bi nešto trebalo drugačije prezentirati, pošaljite *email* ili *pull request*.

Duljina knjige prvenstveno je posljedica moje želje da gotovo sve dokaze raspišem do sitnih detalja, kako bih sebi i vama pokazao da ništa nije provučeno „ispod stola”, te da biste uočili da osnovnih ideja zapravo nema puno. Ogroman broj dokaza provodi se matematičkom indukcijom, rastavom na slučajeve, ili pak eksplicitnom konstrukcijom (*programiranjem*) objekata koji zadovoljavaju traženu specifikaciju. Ako vam se bilo

koji dokaz učini prelaganim (ili preteškim), ne morate ga čitati — ali tako se izlažete riziku da propustite uočiti sličnu ideju u nekom idućem dokazu. Drugi uzrok veličine knjige je velik broj primjera. Vjerojatno najčešći prijedlog studenata za poboljšanje nastave na evaluacijskim anketama bio je da stavim više primjera. Čuo sam vas, i nadam se da je ovo dovoljno — ako nije, recite.

Sastavni dio ove knjige je i Zbirka zadataka, u kojoj se nalaze brojni zadaci, od šablonskih za vježbu, preko svih zadataka sa starih kolokvija i pismenih ispita do kojih sam uspio doći (pri čemu zahvaljujem Zvonku Iljazoviću i Marku Doki — bivšim asistentima iz Izračunljivosti — na ustupanju zadataka), pa do zadataka koji na određeni način nadopunjuju teoriju izloženu u ovoj knjizi, ali nisu nužni za njeno razumijevanje.

Nadimak ove knjige — *Computonomicon* — relativno je doslovni prijevod njene svrhe: prikaz (εἰκόνα) zakona (νόμος) računanja (COMPVTVS). Miješanje grčkih i latinskih korijena je omiljena razonoda računaraca — promotrite pridjev „heksadecimalni“.

Zahvalan sam kolegama i studentima koji su čitali rane *draftove* ove knjige, i brojnim prijedlozima pridonijeli njenom poboljšanju. Među njima svakako bih istaknuo Marka Horvata, koji još uvijek nije uvjeren da je računarac, ali je pristao igrati tu ulogu za potrebe čitanja ove knjige.

b. Motivacija

Izračunljivost: matematička obrada pojma algoritma. Čemu to služi? Zar ne znamo pisati algoritme i bez matematičke formalizacije? Koga je zapravo briga za definicije poput „Algoritam je uređena sedmorka, čiji elementi su skupovi ...“, i beskorisne propozicije poput „Postoji algoritam za sortiranje liste brojeva“? Dva su moguća odgovora na to pitanje: praktični i teorijski.

Kažemo da znamo pisati algoritme. Ali kako to činimo? Zapravo ih najčešće *implementiramo* u nekom programskom jeziku, prešutno podrazumijevajući da je ono što taj jezik omogućava izraziti, ni više ni manje nego algoritam. S tim shvaćanjem postoje dva problema. Prvo, programski jezici nastaju godišnjim ritmom, a jezik koji postane toliko popularan da se u njemu počnu pisati općeniti algoritmi, nastane možda svakih desetak godina. Zatrpani smo knjigama koje objašnjavaju besmrtne algoritamske koncepte, pokušavajući nam ih „približiti“ implementirajući ih u jeziku koji je odavno mrtav. Neke od tih knjiga su toliko popularne da su autori gotovo primorani pisati nova izdanja, u kojima su algoritmi potpuno isti, ali je programski jezik promijenjen. Tu se krije implicitna pretpostavka da, što god jedan programski jezik može izraziti, može i drugi. No kako možemo biti sigurni u to? Na primjer, originalni FORTRAN nije dopuštao rekurziju, dok ALGOL jest [5]. Znamo li da se svaki rekurzivni algoritam može zapisati nerekurzivno? Možemo li to dokazati? Također, ako jest tako, zašto

cijelo vrijeme stvaramo nove jezike? Razuman odgovor je da se razlikuju u *nečem drugom*, ne u algoritmima koje prezentiraju. Možemo li „to drugo” eliminirati, svodeći algoritme samo na „čistu esenciju”?

Drugi problem sastoji se u tome da programski jezici nastaju s raznim svrhama, ali izuzetno rijetko s primarnom svrhom modeliranja matematičkih objekata. Još rjeđe takvi jezici postanu planetarno popularni. Popularni jezici su obično opterećeni performansama: optimalnom upotrebom procesora (vremena) i memorije (prostora), i kao posljedica toga njihov dizajn čini razne ustupke hardveru, koji se teško mogu matematički opravdati. Izuzetno je česta pojava, na primjer, da cijele brojeve računala ne reprezentiraju kao elemente skupa \mathbb{Z} , već kao elemente skupa $\mathbb{Z}/2^{64}\mathbb{Z}$. Također, često se instrukcije ne izvršavaju redom kojim se nalaze u izvornom kodu, u svrhu bržeg izvršavanja na višejezgrenim procesorima. Tako nastaje raskorak između algoritma i implementacije, koji ima važne praktične posljedice [1].

Teorijski odgovor na motivacijsko pitanje dobijemo kad se zapitamo što, u općenitom smislu, matematičare navede na formalizaciju nekog pojma. Ponekad je to otkriće paradoksa, ali češće se radi o potrebi da se dokaže *nepostojanje* objekta neke klase \mathcal{K} s nekim svojstvima. Iako smo se za dokaze postojanja mogli osloniti na intuitivni osjećaj da objekte klase \mathcal{K} „prepoznamo kad ih vidimo”, to nam očito više nije dovoljno ako slutimo da traženi objekt ne postoji, i želimo to dokazati. A u svakom se području s vremenom pojave problemi koji odolijevaju svim poznatim metodama, i počne se vjerovati da su možda nerješivi.

Dok nitko nije dovodio u pitanje Euklidove konstrukcije, puno je preciznija formulacija geometrijske konstruktibilnosti bila potrebna da se dokaže da je trisekcija kuta ravnalom i šestarom nemoguća. Dok je za pronalazak Cardanove ili Ferrarijeve formule bilo dovoljno znati nekoliko jednostavnih algebarskih manipulacija, tek je Galoisova teorija omogućila dokaz da analogni postupci nisu mogući za algebarske jednadžbe petog i višeg stupnja. Dok je već Galileo vidio da prirodnih brojeva i njihovih kvadrata ima jednako mnogo koristeći intuitivni pojam bijekcije, bitno je stroža formulacija bila potrebna Cantoru za dijagonalni argument kojim je dokazao da bijekcija između \mathbb{N} i \mathbb{R} ne postoji. Na meta-razini također: Cantor je uspio naći dokaze za mnoge tvrdnje ili njihove negacije u svojoj teoriji skupova, ali tek je formalna aksiomatizacija omogućila da se dokaže da takvi dokazi za neke tvrdnje (kao što je hipoteza kontinuuma), niti za njihove negacije, jednostavno ne postoje.

Od davnina je poznat problem rješavanja *diofantskih jednadžbi* — nalaženja prirodnih brojeva koji zajedno s još nekim fiksnim prirodnim brojevima, zbrajanjem i množenjem, čine dva izraza jednakima. Modernim jezikom, zadan je polinom s cjelobrojnim koeficijentima u k varijabli (recimo $x_2^3 - x_1^2 - 1$), i želimo ustanoviti ima li nultočku u \mathbb{N}^k — ili u \mathbb{Z}^k , što se može svesti na prirodni slučaj. Za mnoge specijalne polinome znali smo odgovor, za mnoge specijalne potklase (recimo kad je broj varijabli

k, ili stupanj polinoma, jednak 1) poznavali smo od davnina algoritme za nalaženje nultočaka, ali opći algoritam, koji bi za svaki takav polinom u konačno mnogo koraka odgovarao na pitanje ima li prirodnu nultoku, nismo imali. Na slavenoj Hilbertovoj listi od 23 velika matematička problema, deseti je pronalazak takvog algoritma. Protokom vremena, iskristalizirala se mogućnost da algoritam ne postoji, ali za pravi dokaz toga trebalo je prvo formalizirati pojam algoritma. Nakon što je to učinjeno, relativno brzo (uzevši u obzir da su diofantske jednačbe bile poznate tisućama godina) je riješen i deseti Hilbertov problem — naravno, dokazom nepostojanja takvog algoritma.

Nije to bio jedini takav problem: nađeni su brojni drugi problemi za koje se sličnim metodama dokazalo da su algoritamski nerješivi. Danas znamo da je neizračunljivost „posvuda”, i nismo njome više toliko fascinirani, ali to je samo znak ogromnog puta koji smo prešli u shvaćanju algoritama tijekom dvadesetog stoljeća. Jedan dio tog puta prikazan je u ovoj knjizi.

c. Opća i univerzalna izračunljivost

Jedan od velikih ciljeva teorije izračunljivosti je pokazati da pojam izračunljive funkcije zapravo ne ovisi o podlozi na kojoj se njen algoritam izvršava. Iako je lako naći prejednostavne sustave (kao što su na primjer konačni automati, koji ne mogu čak niti uspoređivati proizvoljno velike prirodne brojeve), nakon neke točke dovoljne kompleksnosti svi mehanički sustavi postaju *ekvivalentni* po pitanju toga koje funkcije, uz razumno kodiranje njihovih ulaza i izlaza, računaju. To se vidi iz činjenice da je svaki od njih sposoban *simulirati* sve druge, odnosno reprezentirati njihove konfiguracije (ili barem njihove kodove) unutar svojih, i izvršavati korake njihovog računanja kao (možda komplicirane) procedure na svojim konfiguracijama. Suvremeno računarstvo poznaje taj fenomen pod imenom „virtualizacija”. *Church–Turingova teza* ide i dalje: kaže da se ne samo svi algoritmi izvršivi na svim formalnim modelima izračunljivosti, već i svi „intuitivno zamislivi” algoritmi, mogu izvršavati na nekom konkretnom modelu izračunljivosti, primjerice na Turingovom stroju. Tu tezu je očito nemoguće formalno dokazati — sve dok se ne dogovorimo oko općih aksioma izračunljivosti [2] — ali svaki dokaz ekvivalencije sustava izračunljivosti pruža dodatnu empirijsku potvrdu za nju.

Drugim riječima, izračunljivost je *opći* fenomen: u kojem god modelu da je definiramo, ona će obuhvatiti iste funkcije — ili bar iste s obzirom na prirodno kodiranje ulaza i izlaza. Na primjer, algoritmi za zbrajanje dekadski zapisanih i binarno zapisanih prirodnih brojeva očito su različiti, ali jednako tako je očito da su oba zapravo samo reprezentacije brojevnih funkcija `add`², s obzirom na različite zapise (dekadski odnosno binarni) samih prirodnih brojeva.

Također, jedan od tih modela (pa onda i svi ostali, putem simulacije) zapravo posjeduje svojstvo *univerzalnosti*: ne samo da je za svaku izračunljivu funkciju

moguće naći algoritam unutar tog modela, već je moguće naći *jedan* algoritam koji, ovisno o ulazima, može računati *sve* izračunljive funkcije, odnosno može simulirati sve ostale algoritme. Štoviše, ta „granica dovoljne kompleksnosti”, na kojoj se postiže opća izračunljivost i univerzalnost, je za neke modele začuđujuće nisko. Promotrit ćemo tri vrste takvih sustava: RAM-strojeve, Turingove strojeve, te parcijalno rekurzivne funkcije.

d. Pregled po poglavljima

Ovdje slijedi grubi prikaz rezultata obrađenih u pojedinom poglavlju knjige. Za više detalja pogledajte sadržaj.

U prvom poglavlju uvodimo brojevni model izračunavanja (računanje funkcija koje rade s prirodnim brojevima), kroz imperativno programiranje — RAM-strojeve i makro-strojeve u stilu Shoenfielda [11]. Opisujemo tehniku spljoštenja (*inlining*) kojom se makro-program može pretvoriti u ekvivalentni RAM-program.

U drugom poglavlju dajemo alternativni brojevni model, kroz funkcijsko programiranje — rekurzivne funkcije u stilu Kleeneja. Koristeći funkcijski makro i spljoštenje, konstruiramo kompajler parcijalno rekurzivnih funkcija u RAM-programe, pokazujući da imperativna paradigma može simulirati funkcijsku (računajući iste funkcije).

U trećem poglavlju uvodimo kodiranje, pomoću kojeg možemo u brojevnom modelu računati i funkcije na objektima koji nisu prirodni brojevi. Koristeći kodiranje, konstruiramo interpreter RAM-programa u funkcijskom jeziku, pokazujući ekvivalentnost imperativne i funkcijske paradigme (Kleenejev teorem o normalnoj formi).

U četvrtom poglavlju uvodimo jezični model izračunavanja (računanje funkcija koje rade s nizovima znakova), kroz Turingove strojeve u stilu Sipsera [12]. Opisujemo Turing-izračunavanje parcijalno rekurzivnim funkcijama i transpiliramo RAM-strojeve u Turingove strojeve, čime dokazujemo ekvivalentnost brojevnog i jezičnog izračunljivosti.

U petom poglavlju generaliziramo dobivene rezultate ekvivalentnosti raznih modela izračunavanja kroz Church–Turingovu tezu, te napokon dokazujemo rezultate o nepostojanju algoritma za pojedine probleme, kao što je Churchov teorem o neodlučivosti logike prvog reda. Skiciramo kako se iz toga može dobiti Gödelov prvi teorem nepotpunosti.

U šestom poglavlju dokazujemo četiri velika teorema o metaprogramiranju: teorem o parametru (specijalni slučaj spljoštenja), teorem rekurzije (simuliranje općih rekurzija), teorem o fiksnoj točki (izračunljiva transformacija RAM-programa ne mijenja semantiku nekoga od njih) i Riceov teorem (semantička svojstva RAM-programa nisu odlučiva).

U sedmom poglavlju uvodimo rekurzivnu prebrojivost kao formalizaciju poluodlučivosti, te je karakteriziramo na razne načine kako u brojevnom tako i u jezičnom modelu: kao projekcije rekurzivnih relacija, kao čekanje na zaustavljanje izračunavanja, kao enumeracije (slike rekurzivnih nizova) i kao grafove parcijalno rekurzivnih funkcija.

Poglavlje 1.

RAM-izračunljivost

1.1. Pojam izračunljive funkcije

Da bismo odgovorili na pitanje što je algoritam, zapitajmo se za početak što algoritam *radi* — ili, što mi algoritmom radimo. Očito, algoritam možemo *pokrenuti* na nekim *ulaznim podacima*, izvršavati njegove korake preciznim redom, te u nekom trenutku, kad algoritam to zatraži, zaustaviti postupak, i dobiti *izlazne podatke*. Algoritam, u tom pogledu, obavlja nekakvu *transformaciju* podataka. Štoviše, algoritam bi trebao biti *deterministički*: isti ulazni podaci trebali bi proizvesti iste izlazne podatke. Matematička formalizacija te transformacije je pojam *funkcije*: algoritam *preslikava* ulazne podatke u izlazne. Kažemo da algoritam *računa* funkciju, i takve funkcije (za koje imamo algoritme) zovemo *izračunljivima*. Tako pitanje „Koji su algoritmi mogući?” postaje nešto preciznije pitanje „Koje su funkcije izračunljive?”.

Da bismo funkciju mogli matematički zapisati, moramo imati preciziranu domenu i kodomenu. Što su naši podaci? Na prvi pogled, mogu biti bilo što: imamo algoritme koji rade na cijelim brojevima, realnim brojevima (preciznije, njihovim aproksimacijama — vidjet ćemo zašto je to bitno), tekstnim podacima (*strings*), datotekama, mrežnim vezama (*sockets*), drugim algoritmima (*higher order programming*), grafovima, objektima, regularnim izrazima, i tko zna čemu. No iskustvo programiranja nas uči da se svi ti raznorazni *tipovi* podataka uvijek mogu — i moraju, ako želimo nešto raditi s njima — reprezentirati u memoriji računala kao neki binarni podaci: konačni nizovi nula i jedinica.

Dakle, mogli bismo uzeti $\{0, 1\}^* := \bigcup_{k \in \mathbb{N}} \{0, 1\}^k$ kao univerzalni skup naših podataka — ali pokazuje se da je zgodnije ako umjesto $\{0, 1\}$ uzmemo proizvoljni konačni neprazni skup („abecedu”) Σ . Funkcije iz Σ^* u Σ^* zovemo *jezične funkcije*, i to je povijesno bio prvi pokušaj formalizacije algoritma: Turingov stroj, kojim ćemo se baviti u poglavlju 4.

Ipak, skup $\{0, 1\}^*$, kao i općeniti Σ^* , matematički je nespretni; recimo, ako hoćemo nešto o njemu dokazati indukcijom, moramo u koraku posebno razmatrati dodavanje nule, a posebno dodavanje jedinice. Ako hoćemo napraviti neku petlju kroz njega, nije baš lako odrediti sljedbenika zadanog elementa. Nezgodna je i u tome što uobičajenim leksikografskim uređajem nije dobro uređen: na primjer, skup $\{0^n 1 \mid n \in \mathbb{N}\}$ nema

najmanji element.

Za dokazivanje teorema bolje je uzeti jednostavniji skup, te ćemo u najvećem dijelu knjige promatrati *brojevine* funkcije, za koje su ulazni i izlazni podaci **prirodni brojevi**. U nekom smislu, skup prirodnih brojeva je najjednostavniji mogući skup na kojem se može raditi teorija izračunljivosti — svakako je najjednostavniji među beskonačnim skupovima, a izračunljivost na konačnim skupovima je trivijalna: svaki algoritam može se napisati jednostavno kao tablica (*lookup table*).

Odabir skupa \mathbb{N} kao osnovnog isplatit će se kroz jednostavnost mnogih dokaza (jer imamo matematičku indukciju, jasan početak i sljedbenika, dobar uređaj, ...), ali s druge strane, zato će biti kompliciranije *kodirati* razne druge matematičke objekte kao prirodne brojeve. Za usporedbu, skup Σ^* je zgodniji za kodiranje, jer već imamo intuitivnu predodžbu raznih objekata kao nizova znakova ($\Sigma = \text{ASCII}$): recimo, nitko nam ne mora objasniti kodiranje da bismo znali koji element od \mathbb{Q} predstavlja ' -22/3 '.

Ipak, neintuitivnost kodiranja nadomjestit će lakoća pisanja algoritama: dok je, uz odgovarajuće tehnike [9], lako napisati algoritam za npr. zbrajanje racionalnih brojeva kodiranih prirodnim brojevima, analogni algoritam za ASCII-kodirane razlomke gotovo nikada ne stigne dalje od grubog pseudokoda. Ponegdje, gdje su naši objekti već *definirani* kao nizovi znakova (najvažniji primjer su formule logike prvog reda), svakako ćemo koristiti njihovu jezičnu reprezentaciju, ali to će biti nakon što objasnimo općenito kodiranje sa Σ^* u \mathbb{N} (i obrnuto).

Smatramo li nulu prirodnim brojem? Treba li brojenje početi od 0 ili od 1, dilema je stara koliko i samo računarstvo [4]. Kao i drugdje u matematici, postoje dobri razlozi za obje opcije. Zato ćemo koristiti oba skupa, no kako će nam češće trebati nula među prirodnim brojevima (pogledajmo definiciju od $\{0, 1\}^*$, na primjer), skup s nulom imat će kraću oznaku.

$$\mathbb{N} := \{0, 1, 2, 3, \dots\} \quad (1.1)$$

$$\mathbb{N}_+ := \{1, 2, 3, 4, \dots\} \quad (1.2)$$

Napomena 1.1. Pričali smo o izlaznim podacima u množini, no lako je vidjeti da — s obzirom na to da nas samo zanima postojanje algoritma, ne i njegove performanse — ništa ne gubimo fiksiranjem broja izlaznih podataka na 1. Algoritam s k ulaznih i l izlaznih podataka uvijek možemo promatrati kao l algoritama s istih k ulaznih podataka i s po jednim izlaznim podatkom.

Na primjer, u nekim programskim jezicima postoji operacija **divmod** iz \mathbb{Z}^2 u \mathbb{Z}^2 , koja provodi dijeljenje s ostatkom u \mathbb{Z} , te vraća količnik i ostatak. Nju uvijek možemo, čak i da je nemamo kao osnovnu, emulirati pomoću dvije operacije, $//$ i mod , koje vraćaju količnik i ostatak istog dijeljenja zasebno. Naravno, razlog zašto neki jezici imaju **divmod** kao posebnu funkciju leži u tome da algoritmi za te dvije operacije imaju mnogo zajedničkih koraka, te ako smo odredili količnik, obično je lako iz postupka kojim smo to učinili pročitati i ostatak (sjetite se npr. algoritma za dijeljenje višeznamenastih

brojeva). Zato bismo ponovnim provođenjem algoritma ispočetka za ostatak nepotrebno duplicirali korake. No ako nas samo zanima koje su funkcije izračunljive, očito postoji algoritam za `divmod` ako i samo ako postoje algoritmi za `//` i za `mod`, te nam je dovoljno baviti se pitanjem jesu li *koordinatne funkcije* `//` i `mod` izračunljive — u ovom slučaju, dakako, jesu. \triangleleft

Kad promatramo broj *ulaznih* podataka (tzv. *mjesnost*) algoritma, situacija je bitno drugačija. Jasno je da algoritam za npr. potenciranje prirodnih brojeva prima bazu i eksponent kao dva ulazna podatka, i ne može se jednostavno zapisati pomoću algoritama koji primaju po jedan ulazni podatak. Zato ćemo promatrati algoritme proizvoljnih mjesnosti $k \in \mathbb{N}_+$, i smatrati da mjesnost čini važan dio identiteta algoritma. Primjerice, „zbroji 2 broja” i „zbroji 5 brojeva” su različiti algoritmi, štoviše ovaj prvi pojavljuje se kao korak (nekoliko puta) u ovom drugom.

Direktna posljedica toga je da u našem modelu ne postoje algoritmi s „varijabilnim brojem” ulaznih podataka, u računarstvu poznati kao *varargs*. Na nekoliko mjesta gdje nam budu potrebni, modelirat ćemo ih pomoću familije algoritama svih mogućih mjesnosti, recimo zbrajanje kao `addk`, $k \in \mathbb{N}_+$. Mjesnost algoritma ili funkcije ćemo obično pisati u superskriptu ako je želimo naglasiti — neće dolaziti do zabune s eksponentima jer algoritme niti funkcije nećemo potencirati, niti s oznakom f^{-1} za inverznu funkciju jer mjesnost ne može biti negativna.

Iako mjesnost smatramo neodvojivim dijelom funkcije odnosno algoritma, u slučaju nespecificirane mjesnosti k nespretno je pisati x_1, x_2, \dots, x_k svugdje gdje trebamo napisati argumente odnosno ulazne podatke. Zato ćemo često tih k prirodnih brojeva skraćeno pisati \vec{x} , ili \vec{x}^k ako želimo naglasiti koliko ih ima — no najčešće će se to moći zaključiti iz konteksta: recimo, u $f^7(\vec{x}, y, z)$, očito je duljina od \vec{x} jednaka 5.

Napomena 1.2. S obzirom na to da promatramo samo determinističke algoritme, naglasimo da nema „implicitnih argumenata”: sve vrijednosti o kojima ovisi izlaz funkcije (ako se doista mijenjaju od poziva do poziva) moraju biti prenesene u nju kao argumenti. Često ćemo pisati opće funkcijske pozive kao $f(\vec{x}, y, z)$ — gdje su y i z „lokalne varijable” s kojima doista nešto radimo u konkretnom algoritmu, a \vec{x} predstavlja samo kontekst (*environment*) nekog vanjskog algoritma koji je pozvao f — koji također moramo prenijeti u f ako želimo da mu ona može pristupiti. \triangleleft

Pažljiv čitatelj će primijetiti da zahtijevamo da mjesnost bude pozitivan prirodan broj, odnosno ne promatramo algoritme s 0 ulaznih podataka. Ovo nije bitna restrikcija (možete se zabaviti pokušavajući otkriti koje sve tehničke detalje u knjizi treba promijeniti da bismo uključili i takve algoritme u razmatranje), ali komplicira izlaganje, a opet, takvi algoritmi nam nisu zanimljivi iz perspektive izračunljivosti: budući da zahtijevamo determinističnost, nul-mjesni algoritmi mogu računati jedino konstante, a one su svakako izračunljive bez obzira na formalizam.

1.1.1. Parcijalnost

Gdje smo u dosadašnjem tekstu pričali o općenitim izračunljivim funkcijama, pazili smo da upotrijebimo prijedlog „iz”: funkcija *iz* A u B . Općenito u matematici, takva fraza označava *parcijalne* funkcije, koje ne moraju biti definirane u svim točkama od A (precizno, domena im je podskup od A). Recimo, tangens je parcijalna funkcija iz \mathbb{R} u \mathbb{R} , jer $\frac{\pi}{2} \in \mathbb{R} \setminus \mathcal{D}_{\text{tg}}$. Takve funkcije označavamo oznakom $f: A \rightarrow B$, za razliku od *totalnih* funkcija koje označavamo $f: A \rightarrow B$ i zovemo ih funkcije *sa* A u B .

Dopuštajući algoritmima da računaju parcijalne funkcije, zapravo im omogućavamo da je za neke ulazne podatke njihov rad sasvim dobro definiran (dakle, ovdje ne mislimo na izuzetke, *exceptions*, kao što je dijeljenje nulom), ali da ipak ne postoji završna konfiguracija iz koje bismo mogli pročitati izlazni podatak. Nakon malo razmišljanja dolazimo do zaključka da je to jedino moguće tako da algoritam za neke ulaze beskonačno radi, odnosno nikada ne stane.

Nije li to u kontradikciji s naivnom definicijom algoritma, koja kaže da se radi o *konačnom* postupku? Jest, ali to samo pokazuje da naivne definicije nisu dovoljne, i da nam treba formalizacija. Naime, naivna definicija algoritma, baš kao i naivna definicija skupa (Russell!), dovodi do paradoksa. *Moramo* u obzir uzeti i parcijalne funkcije, odnosno algoritme koji ne stanu uvijek, ako želimo konzistentnu teoriju. Evo kratke skice argumenta — precizno ćemo ga provesti kad precizno definiramo pojmove.

Budući da želimo algoritme moći reprezentirati u računalu, moramo ih moći prikazati kao konačne nizove nula i jedinica. Ta reprezentacija mora biti injekcija ako želimo išta raditi s tim algoritmima, a iz teorije skupova znamo da je $\{0, 1\}^*$ prebrojiv, dakle **svih algoritama ima prebrojivo mnogo**. Specijalno, svih jednomjesnih algoritama ima prebrojivo mnogo (naravno da ih ima beskonačno mnogo). Poredajmo sve jednomjesne algoritme u niz. Pogledajmo sada ovaj jednomjesni algoritam: „Za ulaz $x \in \mathbb{N}$, nađi x -ti algoritam u nizu, i primijeni ga na x . Izlaz tog algoritma (s ulazom x) označi s y . Vрати $y + 1$.” Jasno je da *taj* algoritam ne može biti na popisu, ako algoritmi računaju totalne funkcije (ako je r -ti po redu, tada s ulazom r mora davati i y i $y + 1$); ali ako računaju parcijalne funkcije, nema kontradikcije — jednostavno, y može biti nedefiniran, jer x -ti algoritam s ulazom x ne mora stati.

Ipak, bitno je lakše raditi s algoritmima koji računaju totalne funkcije. Parcijalne funkcije moramo dozvoliti u krajnjoj općenitosti, ali mnoge funkcije koje ćemo koristiti u izgradnji teorije bit će ne samo totalne, nego i *sintaksno* totalne unutar teorije koju gradimo: već iz njihovog oblika bit će jasno da algoritmi koji ih računaju uvijek stanu. Takve funkcije zvat ćemo *primitivno rekurzivnima*.

Kad smo već kod toga, recimo nekoliko riječi i o klasičnim izuzecima poput dijeljenja nulom. Primitivno rekurzivne funkcije po definiciji moraju biti totalne, pa si ne možemo priuštiti jednostavno reći nešto poput „ $3 // 0$ nije definirano” (dokazat ćemo da je $//$ primitivno rekurzivna operacija). U mnogim slučajevima to ćemo rješavati

jednostavno tako da kažemo „*postoji* primitivno rekurzivna funkcija f koja se podudara s traženom funkcijom g na domeni \mathcal{D}_g “, ne govoreći ništa o vrijednostima $f(\vec{x})$ za $\vec{x} \notin \mathcal{D}_g$. Još općenitije, ponekad ćemo umjesto funkcije g navesti samo neko *svojstvo* koje vrijednosti od f moraju zadovoljavati na nekom skupu. Kazat ćemo tada da smo *parcijalno specificirali* (totalnu) funkciju f : u smislu, f je definirana svuda, ali nas zanimaju samo vrijednosti na nekom užem skupu.

Treba napomenuti da je važna odlika ove knjige da će svi algoritmi biti precizno specificirani — ništa neće ostati na pseudokodu. Dakle, uvijek ćemo moći precizno izračunati vrijednosti primitivno rekurzivne funkcije i izvan skupa na kojem je specificirana — na primjer, algoritam za \parallel reći će nam da je $3 \parallel 0 = 3$. Ponekad će čak te vrijednosti (*undocumented behavior*) imati neko značenje, u smislu da će takvu funkciju biti lakše uklopiti u kasnije definicije bez puno rastava na slučajeve. No to nećemo često koristiti, i svaki put ćemo naglasiti kad se to dogodi.

Iako sva računanja možemo shvatiti kao računanja funkcija, izlaganje je jednostavnije ako uvedemo i *relacije*, koje ćemo računati kao specijalni slučaj računanja funkcija. Iz standardne skupovnoteorijske perspektive to se čini čudnim: nisu li relacije općenit pojam, a funkcije samo specijalni slučaj, relacije s funkcijskim svojstvom?

Iz algoritamske perspektive, nisu. Ako izračunljivu funkciju f reprezentiramo pomoću algoritma koji za dani \vec{x} računa njenu vrijednost $f(\vec{x})$, izračunljivu relaciju R prirodno je predstaviti algoritmom koji za dani \vec{x} računa *istinitosnu* vrijednost (bool: *true* ili *false*), već prema tome je li $\vec{x} \in R$ ili nije. Većina programskih jezika uopće nema mogućnost programiranja relacija kao zasebnog tipa algoritma, već ih reprezentiraju funkcijama čiji povratni tip je bool.

Na primjer, reći da je dvomjesna relacija uređaja $<$ na prirodnim brojevima izračunljiva zapravo znači reći da postoji algoritam koji za sve ulaze $(x, y) \in \mathbb{N}^2$ u konačno mnogo koraka vraća *true* ako je $x < y$, a *false* inače. Ili, skup prim-brojeva (jednomjesna relacija \mathbb{P}) je izračunljiv jer možemo napisati algoritam `isPrime`: $\mathbb{N} \rightarrow \text{bool}$, koji za svaki x u konačno mnogo koraka vraća *true* ako je $x \in \mathbb{P}$, a *false* ako $x \notin \mathbb{P}$. Iz navedenih primjera vidimo da je o relacijama prirodno katkad pričati pomoću formula s relacijskim simbolima ($R(\vec{x})$, ili $x R y$ za dvomjesne relacije), a katkad pomoću skupova ($\vec{x} \in R$).

U skladu s uobičajenom praksom modernih programskih jezika, prešutno koristimo standardno ulaganje skupa bool u \mathbb{N} , tako da preslikamo *false* $\mapsto 0$, te *true* $\mapsto 1$. Drugim riječima, na izračunljivost relacije R gledamo kao na izračunljivost njene *karakteristične funkcije* χ_R , koja je iste mjesnosti kao i R . U suprotnom smjeru, kad želimo interpretirati proizvoljni prirodni broj kao bool, koristimo (opet standardnu) interpretaciju po kojoj se 0 interpretira kao *false*, a svi ostali prirodni brojevi kao *true*: drugim riječima, koristimo kompoziciju s karakterističnom funkcijom $\chi_{\mathbb{N}_+}$ (za koju ćemo dokazati da je izračunljiva).

Dakle, relacijama ćemo pripisivati svojstva izračunljivosti koja imaju njihove ka-

rakteristične funkcije: na primjer, reći ćemo da je R primitivno rekurzivna ako je χ_R primitivno rekurzivna. Primijetimo da kod relacija ne moramo razmišljati o parcijalnosti: karakteristična funkcija je uvijek totalna. Relacije imaju drugi način iskazivanja parcijalnosti, takozvanu *rekurzivnu prebrojivost*, o čemu ćemo više reći u poglavlju 7.

1.2. Notacija

Iz prethodne točke zaključujemo da ćemo promatrati (algoritme za) dvije vrste funkcija: jezične i brojevne. Brojevne funkcije su nam važnije i uglavnom ćemo raditi s njima, ali na nekoliko mjesta dobro će nam doći i formalizacija izračunljivosti jezičnih funkcija.

Svaka brojevena funkcija je oblika $f: S \rightarrow \mathbb{N}$, gdje je $S \subseteq \mathbb{N}^k$ za neki $k \in \mathbb{N}_+$. Skraćeno pišemo $f: \mathbb{N}^k \rightarrow \mathbb{N}$, ili još kraće f^k , ako nam nije bitan skup $S = \mathcal{D}_f$. Broj k zovemo *mjesnost* funkcije f . Svaka brojevena funkcija ima jedinstvenu mjesnost — osim prazne funkcije \emptyset , čija domena je prazan skup \emptyset . Radi jednostavnosti izlaganja, smatrat ćemo da i prazne funkcije imaju fiksnu mjesnost, odnosno umjesto jedne funkcije \emptyset promatrat ćemo familiju $\emptyset^k, k \in \mathbb{N}_+$, i smatrat ćemo da su, na primjer, \emptyset^3 i \emptyset^8 različite funkcije. Formalno, to možemo napraviti tako da nam „funkcija” znači uređen par, kojem je prva komponenta uobičajena reprezentacija funkcije (skup uređenih parova s nekim svojstvom), a druga komponenta mjesnost — ali nećemo imati potrebu biti toliko formalni, tim više što prazne funkcije nisu zanimljive iz perspektive izračunljivosti: algoritmi koji ih računaju su jednostavno beskonačne petlje.

(Brojevena) relacija je oblika $R \subseteq \mathbb{N}^k$ za neki $k \in \mathbb{N}_+$. Po analogiji s funkcijama, k zovemo *mjesnost* relacije, i pišemo R^k ako ga želimo naglasiti. Kao i za funkcije, iako su sve prazne relacije skupovno jednake (postoji samo jedan prazan skup), promatrat ćemo familiju $\emptyset^k, k \in \mathbb{N}_+$, i smatrati sve njene elemente različitim relacijama. Na kraju krajeva, njihove karakteristične funkcije *jesu* različite, jer imaju različite domene: recimo, $\mathcal{D}_{\chi_{\emptyset^3}} = \mathbb{N}^3$. (Radi se o nulfunkciji $\mathbb{C}_0^3: \mathbb{N}^3 \rightarrow \mathbb{N}$; potrebno je razlikovati nulfunkciju, koja je totalna, od prazne funkcije koja nije definirana nigdje!)

Jezične funkcije ćemo uvijek definirati nad nekom *abecedom* (konačan neprazan skup) Σ , i to će nam biti funkcije $\varphi: \Sigma^* \rightarrow \Sigma^*$. Elementi od $\Sigma^* := \bigcup_{k \in \mathbb{N}} \Sigma^k$ su konačni nizovi *znakova* iz Σ , koje zovemo *riječi* i pišemo jednostavno konkatencijom: recimo, *aab* umjesto (a, a, b) . *Prazna riječ* je niz duljine 0: označavamo je s ε . Iz oblika jezičnih funkcija vidimo da su one jednomjesne: u svrhu reprezentacije funkcija veće mjesnosti, obično se abecedi dodaje *separator*, znak koji služi razdvajanju argumenata. Recimo, višemjesne funkcije nad $\{a, b\}$ možemo reprezentirati kao jednomjesne funkcije nad tročlanom abecedom $\{a, b, ,\}$ — tako da primjerice $\varphi^4(a, abb, \varepsilon, ba)$ računamo kao $\varphi^1(a, abb, , ba)$. Kažemo da je ϕ dobivena *kontrakcijom* iz φ . Primijetimo da je ovo generalnije od brojevnih višemjesnih funkcija jer možemo imati *varargs* (mjesnost možemo zaključiti jednostavnim brojenjem separatora u ulaznoj riječi), ali i dalje

nemamo mogućnost prikazivanja nulmjesnih funkcija: $\phi(\varepsilon)$ je jednostavno $\phi^1(\varepsilon)$, ne $\phi^0()$. Sličan trik možemo primijeniti i kod brojevnih funkcija, nakon što definiramo kodiranje skupa \mathbb{N}^* .

Analogon pojmu relacije u jezičnom slučaju, dakle podskup od Σ^* , zove se jednostavno *jezik*. Iako karakteristična funkcija jezika nije ni brojevnica ni jezična funkcija (ide sa Σ^* u bool), svejedno možemo pomoću kodiranja skupa Σ^* reprezentirati i izračunljivost jezika. O tome ćemo također više reći kasnije.

Domenu, sliku i graf funkcije f označavamo redom s \mathcal{D}_f , \mathcal{J}_f i \mathcal{G}_f . Primijetimo da su sve to relacije: za funkciju mjesnosti k , domena je mjesnosti k , slika je mjesnosti 1 , a graf je mjesnosti $k + 1$. Naravno, iste oznake koristimo i za domene, slike i grafove funkcija koje nisu brojevne. Restrikciju funkcije f na skup S (zapravo na $S \cap \mathcal{D}_f$) označavamo s $f|_S$. Sliku te restrikcije za $S \subseteq \mathcal{D}_f$ označavamo s $f[S] := \{f(x) \mid x \in S\}$. Prasliku skupa T označavamo s $f^{-1}[T] := \{x \in \mathcal{D}_f \mid f(x) \in T\}$. Ako je f^k brojevnica, oznakom \tilde{f} označavamo njeno proširenje nulom: totalnu funkciju $\tilde{f}: \mathbb{N}^k \rightarrow \mathbb{N}$, koja svaki $\vec{x} \in \mathcal{D}_f$ preslika u $f(\vec{x})$, a preostale $\vec{x} \in \mathbb{N}^k \setminus \mathcal{D}_f$ preslika u 0 . *Nosač* brojevne funkcije f je $f^{-1}[\mathbb{N}_+]$, dakle podskup domene na kojem f nije 0 .

Za skupove brojeva koristimo standardne oznake $\mathbb{P} \subset \mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R}$. Često koristimo *diskretne intervale*, koje označavamo oznakama $[a..b]$ ili $[a..b)$, gdje su $a, b \in \mathbb{Z}$ (zapravo \mathbb{N}). Svaki takav interval skup je cijelih brojeva iz odgovarajućeg realnog intervala: na primjer, $[1..5) = [1..4] = \{1, 2, 3, 4\}$.

Brojevne izračunljive funkcije i relacije označavamo posebnim fontom: dok nam g označava proizvoljnu funkciju, ***g*** nam označava funkciju za koju imamo neku vrstu algoritma. U tom smislu, $g(x)$ označava uobičajenu funkcijsku vrijednost (drugu komponentu uređenog para u g čija je prva komponenta x), dok ***g***(x) označava izlazni podatak algoritma za ***g*** pokrenutog s ulaznim podatkom x . Ipak, to se odnosi samo na funkcijski zapis: inače ćemo koristiti uobičajene matematičke oznake gdje god možemo. Recimo, pisat ćemo $x + y + z$ za zbroj tri broja, a^b za potenciranje, $m \mid n$ za djeljivost, ili pak $p \in \mathbb{P}$ za prim-brojeve. No treba imati na umu da su to izračunljive funkcije i relacije (što ćemo dokazati), te da u pozadini stoje algoritmi za ***add***³, ***pow***², ***Divides***², odnosno ***isPrime***¹.

Napomena 1.3. Algoritamsku jednakost (izračunljivu dvomjesnu brojevnu relaciju, koja se u modernim programskim jezicima često označava ‘==’) označavamo uobičajenim simbolom ‘=’, koji i inače koristimo za jednakost matematičkih objekata (funkcija, relacija, skupova, ...). Kod definicija skupova, i funkcija s prethodno specificiranom domenom (što uključuje totalne funkcije), koristimo simbol ‘:=’. Relacije definiramo formulama koristeći ‘ \iff ’. Često imamo potrebu vrijednosti funkcije specificirati izrazom, uz prešutnu pretpostavku „prirodne domene” (sve ulazne vrijednosti za koje izraz ima smisla). Tada pišemo $f(\vec{x}) \simeq$ izraz. Ovisno o obliku izraza, definirat ćemo precizno značenje fraze „imati smisla”. ◁

Ponekad ćemo imati potrebu korištenja znaka \simeq između dva izraza, što će značiti da su oni jednaki za one vrijednosti varijabli za koje imaju smisla, te da oba izraza imaju smisla za iste vrijednosti varijabli. Drugačije rečeno, $\text{izraz1} \simeq \text{izraz2}$ znači da su definicije $f(\vec{x}) := \text{izraz1}$ i $f(\vec{x}) := \text{izraz2}$ ekvivalentne (definiraju istu funkciju), gdje su u \vec{x} sve varijable koje se pojavljuju bilo u izraz1 , bilo u izraz2 . Razlog za izbjegavanje korištenja znaka $=$ i u takvom slučaju leži u tome što relacija \simeq , kao i svojstvo „imati smisla”, nisu izračunljive. Još jedan razlog za korištenje neuobičajenog znaka je što mnoga „instinktivna pojednostavljenja” više nisu validna: na primjer, ako je f^3 totalna a g^3 nije, $f(\vec{x}) + 0 \cdot g(\vec{x}) \not\simeq f(\vec{x})$ — jer izraz zdesna ima smisla za sve $\vec{x} \in \mathbb{N}^3$, dok izraz slijeva ima smisla samo za $\vec{x} \in \mathcal{D}_g$.

1.3. RAM-stroj i RAM-program

Prvi model izračunavanja koji ćemo promotriti — *RAM-stroj* — dobiven je kao pojednostavljenje (gotovo karikatura) modernih računalnih procesora. S jedne strane, radi se o „krajnje RISC procesoru”, sa samo tri tipa instrukcija (a i jedan od ta tri tipa je trivijalno eliminabilan; ipak, zadržat ćemo ga zbog jednostavnosti izlaganja).

S druge pak strane, ne pretpostavljamo nikakva ograničenja na broj dostupnih registara (pretpostavljamo da ih ima dovoljno za spremanje ulaznih i izlaznih podataka, te za odvijanje programa — svaki konkretni algoritam koristit će konačno mnogo *relevantnih* registara, ali ne postavljamo gornju granicu s obzirom na sve algoritme), niti na veličinu pojedinog registra (u svakom trenutku izvršavanja algoritma, u svakom relevantnom registru može se nalaziti proizvoljni prirodni broj). Obje značajke nužne su već za reprezentaciju ulaza: postoje algoritmi proizvoljno velike mjesnosti, te ih je moguće pozvati s proizvoljno velikim ulaznim podacima.

Još jedno bitno pojednostavljenje sastoji se u tome da ćemo pretpostavljati da je program *fiksni*: ne može se mijenjati (tzv. *harvardska* arhitektura). Iako kôd koji sam sebe mijenja za vrijeme izvršavanja nije baš popularan na modernim računalnim arhitekturama (prvenstveno iz sigurnosnih razloga), osnovna ideja modernog računala je da „dovoljno nisko” imamo jedan procesor koji je sposoban izvršavati razne programe (*von Neumannova* arhitektura). Da bismo počeli koristiti drugi operacijski sustav, dovoljno je instalirati ga i ponovo pokrenuti računalno; ne moramo kupovati novi procesor.

Razlog zašto radimo s ograničenijim modelom je što *von Neumannova* arhitektura *pretpostavlja* univerzalnost, koju mi tek trebamo dokazati. To ćemo svakako učiniti, ali tek u poglavlju 3. Krenimo s osnovnim definicijama.

Definicija 1.4. *RAM-stroj* je matematički (idealizirani) stroj, koji sadrži:

- *RAM-program*: fiksni konačni niz *instrukcija* $P := (I_0, I_1, I_2, \dots, I_{n-1})$;

- *registre*: za svaki $j \in \mathbb{N}$, registar \mathcal{R}_j , koji može sadržavati bilo koji prirodni broj;
- *programski brojač* (PC): još jedan „registar”, koji u svakom trenutku izračunavanja sadrži broj iz intervala $[0..n]$. \triangleleft

RAM-program najčešće pišemo kao

$$P := \begin{bmatrix} 0. I_0 \\ 1. I_1 \\ \vdots \\ (n-1). I_{n-1} \end{bmatrix}, \text{ ili skraćeno } P := \left[t. I_t \right]_{t < n}. \quad (1.3)$$

Broj instrukcija programa P zovemo još *duljinom* programa P , i označavamo ga s n_P .

Sadržaj registara se može mijenjati za vrijeme izvršavanja programa, ovisno o instrukcijama. Početni sadržaj određen je ulaznim podacima. Irelevantni registri (koji se ne spominju u instrukcijama, niti služe za ulaz) formalno sadrže vrijednost 0, iako (po definiciji irelevantnosti) zapravo nije bitno koju vrijednost sadrže.

Sadržaj programskog brojača također se mijenja, tako da se izvršavanjem svake instrukcije poveća za 1, osim ako sama instrukcija kaže drugačije. Početna vrijednost programskog brojača je 0. U svakom trenutku sadržaj programskog brojača je redni broj instrukcije koja se trenutno izvršava, dok vrijednost n označava kraj izvođenja programa.

Definicija 1.5. Svaka *RAM-instrukcija* ima:

- (ako je dio RAM-programa P) *redni broj*, element skupa $[0..n_P)$;
- *tip*, koji može biti jedan od tri tipa: INC, DEC ili GO TO;
- (ako je tipa INC ili DEC) registar na kojem djeluje: \mathcal{R}_j za neki $j \in \mathbb{N}$;
- (ako je tipa DEC ili GO TO, te je dio RAM-programa P) *odredište*: element skupa $[0..n_P]$. \triangleleft

Dakle, RAM-instrukcija može biti jednog od tri oblika, čiji efekti su:

INC \mathcal{R}_j : Povećava sadržaj registra \mathcal{R}_j za 1.

DEC \mathcal{R}_j, l : Ako je sadržaj od \mathcal{R}_j pozitivan, smanjuje ga za 1. Inače postavlja PC na l .

GO TO l : Postavlja PC na l .

Lema 1.6. *Skup Ins svih RAM-instrukcija je prebrojiv.*

Dokaz. Skup Ins_{INC} svih instrukcija tipa INC je prebrojiv: preslikavanje $f_1: \mathbb{N} \rightarrow \text{Ins}_{\text{INC}}$ zadano s $f_1(j) := (\text{INC } \mathcal{R}_j)$ je bijekcija. Analogno, koristeći odredište (iako je broj odredišta ograničen za fiksni program P , svaka instrukcija GO TO l se može pojaviti u nekom programu), skup $\text{Ins}_{\text{GO TO}}$ je prebrojiv, a skup Ins_{DEC} je ekvipotentan sa $\mathbb{N} \times \mathbb{N}$, te je i on prebrojiv. Sada je Ins prebrojiv kao (disjunktna) unija tih triju prebrojivih skupova. \square

Korolar 1.7. *Skup Prog svih RAM-programa je prebrojiv.*

Dokaz. Direktno iz činjenice da je $\text{Ins}_{\text{INC}}^* \subseteq \text{Prog} \subseteq \text{Ins}^*$, i leme 1.6. Iz teorije skupova znamo da je skup A^* prebrojiv ako je A prebrojiv. \square

Jednom kad imamo definirane instrukcije, program i stroj, možemo preciznije definirati kako stroj izvršava program, odnosno o kakvom se točno algoritmu tu radi.

Definicija 1.8. Neka je \mathcal{S} RAM-stroj s programom P , registrima $\mathcal{R}_j, j \in \mathbb{N}$, te programskim brojačem PC . *Konfiguracija* RAM-stroja \mathcal{S} je bilo koje preslikavanje

$$c: \{\mathcal{R}_j \mid j \in \mathbb{N}\} \cup \{PC\} \rightarrow \mathbb{N} \quad (1.4)$$

takvo da je skoro svuda 0 (skup $c^{-1}[\mathbb{N}_+]$ je konačan), te je $c(PC) \leq n_P$. Skraćeno je pišemo kao $c = (c(\mathcal{R}_0), c(\mathcal{R}_1), \dots, c(PC))$. Konfiguracija c je *završna* ako je $c(PC) = n_P$. *Početna konfiguracija* s ulazom $\vec{x} = (x_1, x_2, \dots, x_k) \in \mathbb{N}^k$ je $(0, x_1, x_2, \dots, x_k, 0, 0, \dots, 0)$ (za $1 \leq j \leq k$, u \mathcal{R}_j je x_j ; svugdje drugdje su nule).

Za konfiguracije $c = (r_0, r_1, \dots, pc)$ i $d = (r'_0, r'_1, \dots, pc')$ istog RAM-stroja s programom $P = (I_0, \dots, I_{n_P-1})$, kažemo da c *prelazi* u d (*po programu* P , ili *po instrukciji* I_{pc}), i pišemo $c \rightsquigarrow d$, ako je c završna i $c = d$, ili vrijedi jedno od sljedećeg:

1. $I_{pc} = \text{INC } \mathcal{R}_j$ (za neki j), $r'_j = r_j + 1$, $pc' = pc + 1$, te $r'_i = r_i$ za sve $i \neq j$;
2. $I_{pc} = \text{DEC } \mathcal{R}_j, l$ (za neke j i l), $r'_j = r_j - 1$, $pc' = pc + 1$, te $r'_i = r_i$ za sve $i \neq j$;
3. $I_{pc} = \text{DEC } \mathcal{R}_j, l$ (za neke j i l), $r_j = 0$, $pc' = l$, te $r'_i = r_i$ za sve i ;
4. $I_{pc} = \text{GO TO } l$ (za neki l), $pc' = l$, te $r'_i = r_i$ za sve i . \triangleleft

Često ćemo objašnjavati semantiku instrukcija (kad uvedemo dodatne instrukcije) na gornji način. Pri tome se držimo dogovora da je konfiguracija prije izvođenja instrukcije označena oznakama bez crtica, a ona nakon izvođenja instrukcije oznakama s crticama. Također smatramo da je $r'_i = r_i$ za sve i koji nisu navedeni, a $pc' = pc + 1$ ako nije rečeno drugačije. Uz taj dogovor, semantika instrukcije $\text{INC } \mathcal{R}_j$ se može zapisati jednostavno kao $r'_j = r_j + 1$, semantika instrukcije $\text{GO TO } l$ kao $pc' = l$, a semantika instrukcije $\text{DEC } \mathcal{R}_j, l$ kao: ako je $r_j > 0$, tada $r'_j = r_j - 1$, a inače $pc' = l$.

Determinističnost RAM-stroja sada možemo formalizirati.

Lema 1.9. *Svaka konfiguracija prelazi u neku, jedinstvenu, konfiguraciju.*

Dokaz. Neka je S proizvoljni RAM-stroj, čiji program označimo $(I_0, I_1, \dots, I_{n-1})$, te $c = (r_0, r_1, \dots, p_c)$ proizvoljna njegova konfiguracija. Po definiciji je $p_c \leq n$, te ako vrijedi jednakost, c je završna, te prelazi u samu sebe (i nijednu drugu konfiguraciju, jer I_{p_c} ne postoji). Ako je pak $p_c < n$, pogledajmo tip od I_{p_c} . Ako je ona tipa INC ili GO TO, definicija 1.8 točno propisuje novu konfiguraciju u koju c prelazi.

Inače, I_{p_c} je tipa DEC, recimo DEC \mathcal{R}_j, l , i tada je opet nova konfiguracija jedinstveno određena, s obzirom na r_j . Ako je $r_j > 0$ („istina”), tada je primjenjivo samo pravilo sa $r'_j = r_j - 1$, a ako je $r_j = 0$ („laž”), tada je primjenjivo samo pravilo sa $r_j = 0$; pravilo $r'_j = r_j - 1$ nije primjenjivo jer po definiciji konfiguracije mora biti $r'_j \in \mathbb{N}$, a ovdje bi bilo $r'_j = -1$. Svako od tih pravila također jednoznačno određuje novu konfiguraciju. \square

Definicija 1.10. *RAM-algoritam* je uređen par RAM-programa P i mjesnosti $k \in \mathbb{N}_+$. Umjesto (P, k) pišemo P^k .

Neka je P^k RAM-algoritam, te $\vec{x} = (x_1, x_2, \dots, x_k) \in \mathbb{N}^k$. *P-izračunavanje s \vec{x}* je niz konfiguracija $(c_n)_{n \in \mathbb{N}}$, takvih da je c_0 početna konfiguracija (stroja s programom P) s ulazom \vec{x} , te za svaki n , c_n prelazi u c_{n+1} . Kažemo da to izračunavanje *stane*, ako postoji $n_0 \in \mathbb{N}$ takav da je c_{n_0} završna.

Neka je P^k RAM-algoritam, te f^k brojeva funkcija iste mjesnosti. Kažemo da P^k *računa* funkciju f ako za sve $\vec{x} \in \mathbb{N}^k$ vrijedi:

- ako je $\vec{x} \in \mathcal{D}_f$, tada *P-izračunavanje s \vec{x}* stane u konfiguraciji oblika $(f(\vec{x}), \dots, n_P)$;
- u suprotnom (ako $\vec{x} \notin \mathcal{D}_f$), *P-izračunavanje s \vec{x}* ne stane. \triangleleft

Drugim riječima, *P-izračunavanje s \vec{x}* stane točno onda kada je $\vec{x} \in \mathcal{D}_f$, te u tom slučaju, u završnoj konfiguraciji, vrijednost registra \mathcal{R}_0 je upravo vrijednost funkcije f u točki \vec{x} .

1.3.1. Skup Comp

Navodimo tri lagane posljedice determinizma.

Propozicija 1.11. *Za svaki RAM-algoritam P^k , za svaki $\vec{x} \in \mathbb{N}^k$, postoji jedinstveno P-izračunavanje s \vec{x} .*

Dokaz. Za postojanje, induktivno definiramo

$$c_0 := \text{početna konfiguracija s ulazom } \vec{x}, \quad (1.5)$$

$$c_{n+1} := \text{jedinstvena konfiguracija u koju } c_n \text{ prelazi (prema lemi 1.9)}. \quad (1.6)$$

Po Dedekindovom teoremu rekurzije, time je dobro definiran niz, i taj niz je po definiciji *P-izračunavanje s \vec{x}* .

Za jedinstvenost, pretpostavimo da postoje dva P-izračunavanja s \vec{x} , $(c_i)_{i \in \mathbb{N}}$ i $(c'_i)_{i \in \mathbb{N}}$. Kako je $c \neq c'$, postoji neki $i \in \mathbb{N}$ takav da je $c_i \neq c'_i$, a zbog dobre uređenosti od \mathbb{N} postoji najmanji takav: označimo ga s i_0 . Taj i_0 nije 0, jer je $c_0 = c'_0 =$ početna konfiguracija s ulazom \vec{x} . Dakle, konfiguracija $c_{i_0-1} = c'_{i_0-1}$ prelazi u dvije različite konfiguracije c_{i_0} i c'_{i_0} , što je kontradikcija s lemom 1.9. \square

Propozicija 1.12. *U svakom RAM-izračunavanju koje stane postoji jedinstvena završna konfiguracija.*

Dokaz. Pretpostavimo da je $(c_i)_{i \in \mathbb{N}}$ P-izračunavanje s \vec{x} u kojem postoje dvije završne konfiguracije, i označimo indekse na kojima se one prvi put pojavljuju sa i_1 i i_2 . Bez smanjenja općenitosti (različitost je simetrična) možemo pretpostaviti $i_1 < i_2$. No budući da je c_{i_1} završna, ona prelazi (samo) u samu sebe, pa indukcijom imamo

$$c_{i_1} = c_{i_1+1} = c_{i_1+2} = \dots = c_{i_2}, \quad (1.7)$$

kontradikcija. \square

Korolar 1.13. *Svaki RAM-algoritam računa jedinstvenu brojevnú funkciju.*

Dokaz. Neka je P proizvoljni RAM-program, te $k \in \mathbb{N}_+$. Definirajmo skup

$$S := \{\vec{x} \in \mathbb{N}^k \mid \text{P-izračunavanje s } \vec{x} \text{ stane}\} \quad (1.8)$$

i na njemu funkciju

$$f(\vec{x}) := c(\mathcal{R}_0), \text{ gdje je } c \text{ završna konfiguracija P-izračunavanja s } \vec{x}. \quad (1.9)$$

Iz definicije slijedi da je $f: S \rightarrow \mathbb{N}$ (k -mjesna) brojevná funkcija, te P^k računa f .

Za jedinstvenost, primijetimo da je mjesnost funkcije određena mjesnošću algoritma (uz prethodni dogovor da se prazne funkcije različitih mjesnosti razlikuju), njena domena je određena stajanjem izračunavanja (koje je jedinstveno zbog propozicije 1.11), a vrijednost funkcije u svakoj točki domene određena je završnom konfiguracijom (koja je jedinstvena zbog propozicije 1.12). \square

Važna posljedica prethodnog rezultata je ograničenje broja izračunljivih funkcija.

Definicija 1.14. Neka je $k \in \mathbb{N}_+$, te f^k brojevná funkcija. Kažemo da je f^k RAM-izračunljiva ako postoji RAM-algoritam P^k koji je računa. Za svaki $k \in \mathbb{N}_+$, oznakom Comp_k označavamo skup svih RAM-izračunljivih funkcija mjesnosti k . \triangleleft

Oznaka Comp_k namjerno ne spominje RAM-model izračunavanja — pokazat ćemo da se isti skup brojevnih funkcija dobije i u drugim modelima koje ćemo razmatrati.

Teorem 1.15. *Za svaki $k \in \mathbb{N}_+$, skup Comp_k je prebrojiv. Skup Comp svih RAM-izračunljivih brojevnih funkcija (svih mjesnosti) je također prebrojiv.*

Dokaz. Neka je k fiksna mjesnost. Preslikavanje sa skupa Prog na skup Comp_k , koje svakom RAM-programu P pridružuje funkciju koju algoritam P^k računa, je dobro definirano prema korolaru 1.13. Iz toga je $\text{card}(\text{Comp}_k) \leq \text{card}(\text{Prog})$, što je \aleph_0 po korolaru 1.7.

Za drugu nejednakost, primijetimo da su za sve $n \in \mathbb{N}$ i $k \in \mathbb{N}_+$, konstantne funkcije C_n^k , zadane sa $C_n^k(\vec{x}) := n$, RAM-izračunljive: doista, računaju ih RAM-algoritmi

$$P_n^k := \left[\begin{array}{c} t. \text{ INC } \mathcal{R}_0 \end{array} \right]_{t < n}^k = \left[\begin{array}{c} 0. \text{ INC } \mathcal{R}_0 \\ 1. \text{ INC } \mathcal{R}_0 \\ \vdots \\ (n-1). \text{ INC } \mathcal{R}_0 \end{array} \right]^k \quad (1.10)$$

(što se može vidjeti indukcijom po n). Iz toga slijedi da je $\{C_n^k \mid n \in \mathbb{N}\} \subseteq \text{Comp}_k$, a kako je taj skup prebrojiv (sve konstante su različite), slijedi $\aleph_0 \leq \text{card}(\text{Comp}_k)$, što zajedno s gornjim daje $\text{card}(\text{Comp}_k) = \aleph_0$.

Sada je, naravno, $\text{Comp} = \bigcup_{k \in \mathbb{N}_+} \text{Comp}_k$ prebrojiv kao unija prebrojivo mnogo prebrojivih skupova. \square

Korolar 1.16. *Za svaki $k \in \mathbb{N}_+$, postoji brojevena funkcija mjesnosti k koja nije RAM-izračunljiva.*

Dokaz. Opet, fiksirajmo mjesnost $k \in \mathbb{N}_+$. Skup svih k -mjesnih brojevnih funkcija Func_k je neprebrojiv, jer je nadskup skupa svih *totalnih* k -mjesnih brojevnih funkcija, čija je kardinalnost

$$\text{card}(\mathbb{N}^{\aleph_0^k}) = \aleph_0^{\aleph_0^k} = \aleph_0^{\aleph_0} = \mathfrak{c} > \aleph_0. \quad (1.11)$$

Iz toga slijedi $\text{Func}_k \not\subseteq \text{Comp}_k$, pa postoji funkcija iz $\text{Func}_k \setminus \text{Comp}_k$, što smo trebali. \square

1.3.2. Primjeri RAM-programa

Jednu familiju primjera, RAM-programe za konstantne funkcije, vidjeli smo već u dokazu teorema 1.15. Specijalno, za $n = 0$, *prazan program* $[]$ računa *nulfunkciju* C_0^k za svaki $k \in \mathbb{N}_+$. Dakle, prazan program nažalost ne računa praznu funkciju — što bi bilo lako zapamtiti — ali računa *praznu relaciju* \emptyset^k , odnosno njenu karakterističnu funkciju. Također, za $n = 1$, program $[0. \text{ INC } \mathcal{R}_0]$ računa univerzalnu relaciju \mathbb{N}^k .

Napišimo sada i RAM-program koji računa praznu funkciju \emptyset^k . Iz definicije zaključujemo da je to program čije izračunavanje s \vec{x} ne stane ni za koji \vec{x} (bez obzira na mjesnost). Dakle, moramo spriječiti PC da dođe do n , odnosno treba nam instrukcija s odredištem, koja će vratiti PC na staru vrijednost tako da se ne poveća za 1 (*petlja*), i to ona koja se izvrši uvijek (*beskonačna petlja*). Najjednostavniji takav program je $[0. \text{ GO TO } 0]$.

Dosadašnji programi nisu uopće koristili svoje ulaze. Vjerojatno najjednostavniji primjer funkcije koja koristi svoj ulaz je identiteta (mjesnosti 1 — označena s I_1^1). Da bismo je izračunali, moramo prebaciti vrijednost iz \mathcal{R}_1 (ulazni registar za jednomjesne funkcije) u \mathcal{R}_0 (izlazni registar). Lako se vidi, koristeći naše razumijevanje izvršavanja imperativnih programa, da tu svrhu ispunjava RAM-algoritam

$$P_{I_1^1}^1 := \left[\begin{array}{l} 0. \text{ DEC } \mathcal{R}_1, 3 \\ 1. \text{ INC } \mathcal{R}_0 \\ 2. \text{ GO TO } 0 \end{array} \right]^1. \quad (1.12)$$

Neki formalni dokaz bi vjerojatno išao tako da se pokaže da svaki prolaz kroz petlju (čitavanje instrukcija redom) počevši od konfiguracije u kojoj je $r_1 > 0 \wedge pc = 0$ ima semantiku $r'_1 = r_1 - 1 \wedge r'_0 = r_0 + 1 \wedge pc' = 0$. Ako je $r_1 = 0$, izvršavanje instrukcije rednog broja 0 završava izračunavanje, jer pc postane jednak duljini programa, 3. Iz toga se onda indukcijom po r_1 može zaključiti da je semantika čitavog programa $r'_1 = r_1 - r_1 = 0 \wedge r'_0 = r_0 + r_1$, te iz početne konfiguracije s ulazom x : $(0, x, 0, \dots, 0)$, dolazimo u završnu konfiguraciju $(x, 0, 0, \dots, 3)$, iz čega dobivamo izlazni podatak x . Na primjer, za $x = 2$ imamo sljedeću „šetnju” kroz konfiguracije:

$$\begin{aligned} (0, 2, 0, \dots, 0) &\rightsquigarrow (0, 1, 0, \dots, 1) \rightsquigarrow (1, 1, 0, \dots, 2) \rightsquigarrow (1, 1, 0, \dots, 0) \rightsquigarrow \\ &\rightsquigarrow (1, 0, 0, \dots, 1) \rightsquigarrow (2, 0, 0, \dots, 2) \rightsquigarrow (2, 0, 0, \dots, 0) \rightsquigarrow (2, 0, 0, \dots, 3). \end{aligned} \quad (1.13)$$

Ubuduće nećemo biti tako precizni (upravo jer imamo razvijenu intuiciju „programiranja” u imperativnim jezicima), ali ćemo navesti „najvažnije trenutke” u izračunavanju kako bi bilo lakše pratiti što se događa.

Prethodni dokaz (ili programerska intuicija) daje nam i više: ako „naslažemo” (konkateniramo) više takvih blokova za različite ulazne registre, možemo dobiti RAM-programe za zbrajanje. Konkretno, recimo, funkcija add^3 , zadana s $\text{add}(x, y, z) := x + y + z$, je RAM-izračunljiva, jer je računa RAM-algoritam

$$P_{\text{add}^3} := \left[\begin{array}{l} 0. \text{ DEC } \mathcal{R}_1, 3 \\ 1. \text{ INC } \mathcal{R}_0 \\ 2. \text{ GO TO } 0 \\ 3. \text{ DEC } \mathcal{R}_2, 6 \\ 4. \text{ INC } \mathcal{R}_0 \\ 5. \text{ GO TO } 3 \\ 6. \text{ DEC } \mathcal{R}_3, 9 \\ 7. \text{ INC } \mathcal{R}_0 \\ 8. \text{ GO TO } 6 \end{array} \right]^3. \quad (1.14)$$

Ovdje vidimo jednu dobru stranu naizgled čudne konvencije da izračunavanje završava kad programski brojač postane jednak duljini programa: prilikom ovakve konkatenacije

ne trebamo mijenjati odredišta već napisanih instrukcija. Odredište 3 instrukcije rednog broja 0 jednako je označavalo kraj programa za l_1^1 , kao i kraj *tog dijela* programa za add^3 . Primijetite sličnost sa standardnom konvencijom o end-iteratoru u biblioteci STL jezika C++.

Vidimo da su mnoge jednostavne funkcije: prazna, konstante, identiteta, zbrajanje, ... RAM-izračunljive. Ipak, pisati RAM-programe može biti dosta zamorno (recimo za add^8 — mnogi dijelovi se ponavljaju uz neznatne izmjene u odredištima ili adresama registara) ili teško (recimo za množenje, ili potenciranje — povremeno bismo htjeli iskoristiti registar kao brojač za neku petlju, ali istodobno i sačuvati njegovu vrijednost). Prvi problem ćemo riješiti makroima, a drugi funkcijskim programiranjem.

1.4. Makro-izračunljivost

Osnovna ideja makroa je jednostavna: izvršavanje RAM-programa na RAM-stroju, pored toga što prevodi početnu konfiguraciju (s nekim ulazom \vec{x}) u završnu (s nekim izlazom $f(\vec{x})$), proizvede mnoge „popratne efekte” (*side-effects*) na njegovim registrima. Te efekte možemo objediniti (*enkapsulirati*) tako da čitav RAM-program P shvatimo kao jednu instrukciju P^* nekog kompliciranijeg stroja. Oznaka sugerira dualnu upotrebu RAM-programa P : ako ga koristimo za računanje k -mjesne funkcije (k ulaznih registara), promatramo ga kao algoritam P^k , a ako ga koristimo radi efekata na registre (svi registri su „ulazni”), promatramo ga kao *makro* P^* .

Primjerice, za svaki $j \in \mathbb{N}$, RAM-program $P_j := \begin{bmatrix} 0. \text{DEC } \mathcal{R}_j, 2 \\ 1. \text{GO TO } 0 \end{bmatrix}$ ima semantiku $r'_j = 0$ (njegovo izvršavanje postavlja \mathcal{R}_j na nulu — kažemo da *resetira* \mathcal{R}_j). To znači da imamo makro P_j^* koji kasnije možemo koristiti u *makro-programima* kad god želimo resetirati neki registar, ne mijenjajući ostale registre. Taj makro označavat ćemo sa $\text{ZERO } \mathcal{R}_j$.

Definirat ćemo brojne makroe, što će kulminirati *funkcijskim makroom* — koji pruža mogućnost da naš programski jezik, kojim pišemo makro-programe, izvršava prave funkcijske pozive, sa zasebnim okvirom (*scope*) lokalnih varijabli, prijenosom argumenata po vrijednosti, i zapisivanjem povratne vrijednosti u po volji odabrani registar, čuvajući sadržaje registara koji su nam bitni. No do tada je još dug put. Za početak, navedimo osnovne definicije i tvrdnje koje vrijede za makro-paradigmu. Gotovo sve one su sasvim analogne onima u RAM-paradigmi, pa ih nećemo detaljno motivirati odnosno obrazlagati.

Definicija 1.17. *Makro-stroj* je matematički stroj koji sadrži:

- *makro-program*: fiksni konačni niz *makro-instrukcija* $Q := (I_0, I_1, \dots, I_{n-1})$, svaka od kojih je jednog od dva oblika:

- obična RAM-instrukcija (tipa INC, DEC ili GO TO), ili
- makro oblika P^* , gdje je P RAM-program;
- registre $(\mathcal{R}_j)_{j \in \mathbb{N}}$, iste kao i kod RAM-stroja;
- programski brojač PC, isti kao i kod RAM-stroja;
- pomoćni programski brojač AC, čije moguće vrijednosti ac ovise o makro-instrukciji koja se trenutno izvršava (I_{pc} , gdje je pc vrijednost od PC):
 - ako je $I_{pc} = P^*$ za RAM-program P , tada je $ac \in [0..n_P]$.
 - inače, $ac = 0$.

◁

Jednako kao lemu 1.6 i korolar 1.7 (i koristeći korolar 1.7 za broj makroa), možemo dokazati da su skupovi

$$M\mathcal{I}ns := \mathcal{I}ns \dot{\cup} \{P^* \mid P \in \text{Prog}\}, \quad (1.15)$$

$$M\text{Prog} := \{Q \in M\mathcal{I}ns^* \mid \text{sva odredišta u } Q \text{ su manja ili jednaka } n_Q\}, \quad (1.16)$$

svih makro-instrukcija, i svih makro-programa, prebrojivi. Ti rezultati nisu toliko bitni zbog tehnika koje ćemo uskoro razviti, ali predstavljaju dobru vježbu. Sljedeći korak u tom smjeru je konstatacija da makro-izračunljivih funkcija ima prebrojivo mnogo, i kao posljedica toga, postoje brojevnje funkcije koje nisu ni makro-izračunljive. No prvo moramo definirati pojam makro-konfiguracije i makro-izračunavanja.

Definicija 1.18. *Konfiguracija makro-stroja* s programom $Q = (I_0, I_1, \dots, I_{n_Q-1})$, registrima \mathcal{R}_j , $j \in \mathbb{N}$, te programskim brojačima PC i AC, je bilo koje preslikavanje $c: \{\mathcal{R}_j \mid j \in \mathbb{N}\} \cup \{PC, AC\} \rightarrow \mathbb{N}$, takvo da je $c^{-1}[\mathbb{N}_+]$ konačan skup, $c(PC) \leq n_Q$, te vrijedi $c(AC) = 0$, osim u slučaju $I_{c(PC)} = P^*$, kada je $c(AC) \leq n_P$. Skraćeno pišemo $c = (c(\mathcal{R}_0), c(\mathcal{R}_1), \dots, c(PC), c(AC))$.

Početna makro-konfiguracija s ulazom \vec{x} definira se jednako kao i početna RAM-konfiguracija: svuda osim na ulaznim registrima je 0, pa tako i na AC. Također, završna makro-konfiguracija definira se jednako kao i u RAM-slučaju: uvjetom $c(PC) = n_Q$ (tada mora biti $c(AC) = 0$, jer $I_{c(PC)}$ uopće ne postoji). ◁

Definicija 1.19. Za konfiguracije $c = (r_0, r_1, \dots, pc, ac)$ i $d = (r'_0, r'_1, \dots, pc', ac')$ istog makro-stroja s makro-programom $Q = (I_0, I_1, \dots, I_{n_Q-1})$, kažemo da c prelazi u d (po programu Q), i pišemo $c \rightsquigarrow d$, ako vrijedi jedno od sljedećeg:

1. $c = d$, i c je završna konfiguracija ($pc = n_Q$);
2. $ac = ac' = 0$, I_{pc} je RAM-instrukcija, te RAM-konfiguracija (r_0, r_1, \dots, pc) nije završna ($pc < n_Q$) i prelazi u RAM-konfiguraciju (r'_0, r'_1, \dots, pc') po programu Q (odnosno njegovoj instrukciji s rednim brojem pc);

3. $pc' = pc$, I_{pc} je makro P^* , te RAM-konfiguracija (r_0, r_1, \dots, ac) nije završna ($ac < n_P$) i prelazi u RAM-konfiguraciju (r'_0, r'_1, \dots, ac') po programu P (odnosno njegovoj instrukciji s rednim brojem ac);
4. $pc' = pc + 1$, $I_{pc} = P^*$, $ac = n_P$ ((r_0, r_1, \dots, ac) je završna) i $ac' = 0$. \triangleleft

Drugim riječima, makro-stroj funkcioniše na dvije razine. Na „gornjoj”, izvršava RAM-instrukcije u vlastitom makro-programu, koristeći vlastite registre i programski brojač baš kao RAM-stroj. Dolaskom do instrukcije P^* prebacuje se na „donju” razinu, gdje izvršava RAM-instrukcije u RAM-programu P koristeći *iste* registre i pomoćni programski brojač. Dolaskom tog RAM-stroja $(P, (\mathcal{R}_j)_{j \in \mathbb{N}}, AC)$ u završnu konfiguraciju, makro-stroj se vraća na „gornju” razinu: resetira AC na nulu, poveća PC za jedan, i nastavlja izvršavati vlastite instrukcije.

Vidimo da za makro-stroj postoje dva načina da radi u beskonačnoj petlji. Prvi je na gornjoj razini, gdje se svaka makro-instrukcija (barem svaka do koje programski brojač dođe) izvrši u konačno mnogo koraka (prijelaza), ali PC nikad ne dosegne vrijednost n_Q . Drugi je na donjoj razini: u nekom trenutku PC postane i , i makro-stroj počne izvršavati makro $I_i = P^*$ — no s registrima kakvi su bili u tom trenutku, izvršavanje programa P nikad ne završi: AC nikad ne postane n_P , čime PC ostaje na istoj vrijednosti $i < n_Q$ zauvijek. Ako se pak ne dogodi nijedno od toga, makro-stroj će doći u završnu konfiguraciju $(r_0, r_1, \dots, n_Q, 0)$, u kojoj će r_0 predstavljati izlazni podatak.

Napomena 1.20. Makro-program bez ijednog makroa jest RAM-program (konačni niz RAM-instrukcija), ali se ne izvršava na RAM-stroju, nego na makro-stroju. Ipak, definicija 1.18 kaže da u tom slučaju svaka konfiguracija mora preslikavati AC u 0, te definicija 1.19, točka 2, kaže da se u tom slučaju makro-stroj ponaša isto kao i RAM-stroj. Drugim riječima, pojam P -izračunavanja s \vec{x} je dobro definiran bez obzira na to na kojem stroju se izvršava. (Ovu napomenu smo mogli izbjeći tako da uopće ne definiramo RAM-stroj nego samo makro-stroj, no uvođenje pomoćnog brojača koji ništa ne „radi” i čitavo vrijeme RAM-izračunavanja stoji na 0 djelovalo bi čudno.) \triangleleft

Primjer 1.21. Uzmimo RAM-program P_{add^3} iz algoritma (1.14) (za zbrajanje tri broja), i promotrimo makro-stroj s programom

$$Q := \begin{bmatrix} 0. \text{ZERO } \mathcal{R}_1 \\ 1. [\]^* \\ 2. \text{DEC } \mathcal{R}_2, 1 \\ 3. P_{add^3}^* \end{bmatrix}. \quad (1.17)$$

Neki prijelazi između konfiguracija tog stroja su:

$$\begin{aligned}
 (0, 2, 4, 0, \dots, 0, 0) &\rightsquigarrow (0, 1, 4, 0, \dots, 0, 1) \rightsquigarrow (0, 1, 4, 0, \dots, 0, 0) \rightsquigarrow (0, 0, 4, 0, \dots, 0, 1) \\
 &\rightsquigarrow (0, 0, 4, 0, \dots, 0, 0) \rightsquigarrow (0, 0, 4, 0, \dots, 0, 2) \rightsquigarrow (0, 0, 4, 0, \dots, 1, 0) \rightsquigarrow (0, 0, 4, 0, \dots, 2, 0) \\
 &\rightsquigarrow (0, 0, 3, 0, \dots, 3, 0) \rightsquigarrow (0, 0, 3, 0, \dots, 3, 3) \rightsquigarrow (0, 0, 2, 0, \dots, 3, 4) \rightsquigarrow (1, 0, 2, 0, \dots, 3, 5) \\
 &\rightsquigarrow (1, 0, 2, 0, \dots, 3, 3) \rightsquigarrow (1, 0, 1, 0, \dots, 3, 4) \rightsquigarrow (2, 0, 1, 0, \dots, 3, 5) \rightsquigarrow (2, 0, 1, 0, \dots, 3, 3) \\
 &\rightsquigarrow (2, 0, 0, 0, \dots, 3, 4) \rightsquigarrow (3, 0, 0, 0, \dots, 3, 5) \rightsquigarrow (3, 0, 0, 0, \dots, 3, 3) \rightsquigarrow (3, 0, 0, 0, \dots, 3, 6) \\
 &\rightsquigarrow (3, 0, 0, 0, \dots, 3, 9) \rightsquigarrow (3, 0, 0, 0, \dots, 4, 0) \rightsquigarrow (3, 0, 0, 0, \dots, 4, 0) \rightsquigarrow \dots \quad (1.18)
 \end{aligned}$$

Primijetimo da je konfiguracija $(3, 0, 0, 0, \dots, 4, 0)$ završna.

Također, možemo imati

$$(0, \dots, 0, 0) \rightsquigarrow (0, \dots, 0, 2) \rightsquigarrow (0, \dots, 1, 0) \rightsquigarrow (0, \dots, 2, 0) \rightsquigarrow (0, \dots, 1, 0) \rightsquigarrow \dots \quad (1.19)$$

Primijetimo da nijedna od gornjih konfiguracija nije završna. \triangleleft

Sada se, jednako kao za RAM-model, može definirati *makro-algoritam*, *makro-izračunavanje*, izreka da „makro-algoritam računa brojevu funkciju”, te pojam *makro-izračunljive* funkcije. Na primjer, niz (1.18) pokazuje da Q-izračunavanje s $(2, 4)$ stane s izlaznim podatkom 3, dok niz (1.19) pokazuje da Q-izračunavanje s (0) ne stane. Općenitije, recimo, makro-algoritam Q^4 računa funkciju $f: \mathbb{N} \times \mathbb{N}_+ \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, zadanu s $f(x, y, z, t) := y + z - 1$.

Također se mogu (uz malo više tehnikacija) dokazati rezultati o determinističnosti, prebrojivosti skupa makro-izračunljivih funkcija, te postojanju brojevnih funkcija koje nisu takve. Iako to predstavlja dobru vježbu, nećemo ići na taj način — naš cilj je dobiti sve te rezultate s druge strane, tako da dokažemo da se svaki makro-stroj može *simulirati* RAM-strojem, te je skup makro-izračunljivih funkcija *jednak* skupu RAM-izračunljivih funkcija Comp .

1.4.1. Spljoštenje

Dakle, cilj nam je opisati postupak za pretvorbu makro-strojeva u RAM-strojeve koji za iste ulaze prolaze kroz „iste” konfiguracije. One ne mogu biti doslovno iste jer makro-stroj ima dva programska brojača a RAM-stroj samo jedan, ali to je zapravo jedini detalj koji je različit. Sadržaj registara makro-stroja i RAM-stroja bit će isti kako se krećemo kroz izračunavanje, i izvršavat ćemo iste instrukcije (istog tipa nad istim registrima) istim redom, samo će one biti u različitim programima, s različitim rednim brojevima, te će njihova odredišta trebati biti drugačija kako bi se odnosila na odgovarajuće instrukcije u drugom programu.

Ideju konstrukcije je lako opisati intuitivno: „spljoštimo” gornju razinu (na kojoj su makroi P^*) i donju razinu (na kojoj su pojedinačne instrukcije programa P) u

jednu razinu. U modernom računarstvu ta se tehnika zove *inlining*: umjesto makro-instrukcije P^* , na isto „mjesto” (relativnu poziciju u programu u odnosu na ostale instrukcije) stavimo sve instrukcije od P redom. Naravno, time su neki redni brojevi instrukcija prestali biti sinkronizirani s odredištima: prvo, svi redni brojevi instrukcija u P (osim ako je makro P^* bio baš na početku makro-programa), a drugo, svi redni brojevi nakon onog koji je imao makro P^* (osim ako P ima točno jednu instrukciju). Sve njih treba popraviti, a jednako tako i odredišta koja se odnose na njih. Sada je još samo preostalo precizirati taj postupak.

Definicija 1.22. Neka je Q makro-program. *Spljoštenje* od Q definiramo kao RAM-program Q^b , dobiven iz Q sljedećim postupkom:

Dok god postoji barem jedan makro u Q :

1. makni prvi makro iz Q : neka je to i . P^* ;
2. u programu Q , svaki redni broj veći od i , i svako odredište veće od i , povećaj za $n_P - 1$ (tj. smanji za 1 ako je P prazan program);
3. za svaku instrukciju programa P , dodaj u program Q instrukciju istog tipa nad istim registrom, kojoj su redni broj i i odredište (ako ga ima) povećani za i . ◁

Propozicija 1.23. *Preslikavanje b je totalna surjekcija sa skupa $\mathcal{M}\text{Prog}$ na skup Prog .*

Dokaz. Treba vidjeti da za proizvoljni makro-program Q , postupak iz definicije 1.22 uvijek stane u konačno mnogo koraka, i pritom proizvede RAM-program.

Kako je u svakom makrou P^* , P RAM-program, u koraku 3 ne dodajemo nove makroe. S druge strane, u koraku 1 uklanjamo jedan makro, a u koraku 2 ne mijenjamo broj makroa, dakle svaki prolaz kroz petlju smanjuje broj makroa za 1. Kako svaki makro-program ima konačno mnogo makroa, postupak će sigurno završiti (nakon najviše n_Q prolaza kroz petlju). A kada završi, uvjet petlje neće biti ispunjen, dakle u Q više neće biti makroa: drugim riječima, pretvorili smo Q u RAM-program.

Surjektivnost slijedi iz činjenice da je $\text{Ins} \subset \mathcal{M}\text{Ins}$, dakle $\text{Prog} \subset \mathcal{M}\text{Prog}$, te je b na RAM-programima identiteta: uvjet petlje već na početku nije ispunjen, pa se program uopće ne mijenja. Dakle za svaki RAM-program P vrijedi $P^b = P$. ◻

Primjer 1.24. Spljoštimo program Q iz primjera 1.21. Prvi makro u Q nalazi se odmah na početku ($i = 0$) pa ne moramo renumerirati instrukcije koje implementiraju `ZERO` \mathcal{R}_1 — samo ove ispod njih: trebamo im povećati odredišta i redne brojeve za $2 - 1 = 1$. Nakon prvog prolaza kroz petlju tako dobijemo makro-program Q' iz (1.20) lijevo.

Sljedeći makro je onaj koji odgovara praznom RAM-programu, na rednom broju $i = 2$. Za njega očito ne treba provoditi korak 3, samo ga uklonimo i smanjimo redne

brojeve i odredišta veće od 2 za 1. Specijalno, to znači da u instrukciji (3. DEC $\mathcal{R}_2, 2$), odredište ostaje 2, dok se redni broj smanjuje za 1 i postaje također 2. Dobivamo

$$Q' := \begin{bmatrix} 0. \text{DEC } \mathcal{R}_1, 2 \\ 1. \text{GO TO } 0 \\ 2. []^* \\ 3. \text{DEC } \mathcal{R}_2, 2 \\ 4. P_{\text{add}^3}^* \end{bmatrix}, \quad Q'' := \begin{bmatrix} 0. \text{DEC } \mathcal{R}_1, 2 \\ 1. \text{GO TO } 0 \\ 2. \text{DEC } \mathcal{R}_2, 2 \\ 3. P_{\text{add}^3}^* \end{bmatrix}. \quad (1.20)$$

Ostao nam je još jedan makro, koji je ovaj put zadnja instrukcija ($i = 3$). To znači da u koraku 2 ne radimo ništa, samo moramo provesti korak 3. Nakon njega dobijemo

$$Q''' := \begin{bmatrix} 0. \text{DEC } \mathcal{R}_1, 2 \\ 1. \text{GO TO } 0 \\ 2. \text{DEC } \mathcal{R}_2, 2 \\ 3. \text{DEC } \mathcal{R}_1, 6 \\ 4. \text{INC } \mathcal{R}_0 \\ 5. \text{GO TO } 3 \\ 6. \text{DEC } \mathcal{R}_2, 9 \\ 7. \text{INC } \mathcal{R}_0 \\ 8. \text{GO TO } 6 \\ 9. \text{DEC } \mathcal{R}_3, 12 \\ 10. \text{INC } \mathcal{R}_0 \\ 11. \text{GO TO } 9 \end{bmatrix} \quad \begin{array}{l} v(0, 0) := 0 \\ v(0, 1) := 1 \\ v(0, 2) := v(1, 0) := v(2, 0) := 2 \\ v(3, 0) := 3 \\ v(3, 1) := 4 \\ v(3, 2) := 5 \\ v(3, 3) := 6 \\ v(3, 4) := 7 \\ v(3, 5) := 8 \\ v(3, 6) := 9 \\ v(3, 7) := 10 \\ v(3, 8) := 11 \\ v(3, 9) := v(4, 0) := 12 \end{array} \quad (1.21)$$

i gotovi smo: $Q''' = Q^b$, jer više nema makroa u programu. (Za objašnjenje funkcije v čije vrijednosti su napisane pored Q^b , pogledajte skicu dokaza teorema 1.26.) \triangleleft

Postupak za određivanje spljoštenja zapravo je neformalni algoritam, čiji je ulazni podatak makro-program, a izlazni RAM-program. Taj algoritam bismo mogli i formalizirati, tako da razvijemo kodiranja za skupove $\mathcal{M}\text{Prog}$ i Prog — no nema potrebe. Sve za što će nam trebati makro-programi je dokaz da se funkcijski programi mogu zapisati imperativno; a ta pretvorba, iako je svakako mehanička i programabilna, je na meta-razini, „iznad” samih algoritama koji rade na prirodnim brojevima. Iako je jedan od važnih rezultata teorije izračunljivosti da se meta-algoritmi također mogu prikazati kao algoritmi, odnekud moramo početi i zadovoljiti se neformalnim objašnjenjima. Svakako napominjemo da ćemo se na ovaj postupak vratiti u formalnom okruženju, kad budemo imali razvijeno kodiranje RAM-programa, u jednom specijalnom slučaju (dokaz teorema o parametru), za koji ćemo poslije pokazati da je zapravo dovoljan za sva spljoštenja koja ćemo ovdje neformalno napraviti. Može se činiti cirkularnim, ali zapravo nije; baš kao ni npr. govor o modelu teorije skupova ZF kao o skupu — koristimo neformalne pojmove da bismo opisali formalne.

1.4.2. Ekvivalencija RAM-programa i makro-programa

Kad smo već kod neformalnih objašnjenja, izrecimo i osnovni rezultat — koji se doduše može formalno dokazati, ali je vrlo mukotrpno i zapetljano, a zapravo dokaz ne daje ništa novo ako već imamo intuiciju *inlininga* kao programske tehnike.

Definicija 1.25. Za dva (makro- ili RAM-) programa P i Q kažemo da su *ekvivalentni* ako za svaku mjesnost $k \in \mathbb{N}_+$, algoritmi P^k i Q^k računaju istu funkciju. \triangleleft

Teorem 1.26. *Za svaki makro-program Q , RAM-program Q^b je ekvivalentan s Q .*

Skica dokaza. Treba definirati funkciju v iz $\mathbb{N} \times \mathbb{N}$ u \mathbb{N} , takvu da prijelaz između RAM-konfiguracija $(r_0, r_1, \dots, v(pc, ac))$ i $(r'_0, r'_1, \dots, v(pc', ac'))$ po programu Q^b odgovara nekoliko (jednom ili više) prijelaza između makro-konfiguracija $(r_0, r_1, \dots, pc, ac)$ i $(r'_0, r'_1, \dots, pc', ac')$ po programu Q . Intuitivno, funkcija v treba opisivati kako se točno transformiraju redni brojevi instrukcija pri spljoštenju, te preslikavati „završnu konfiguraciju programskih brojača” $(n_Q, 0)$ u n_{Q^b} . Recimo, ako je (i, P^*) prvi makro u Q , znamo da je $v(j, 0) := j$ za sve $j < i$. Na kraju primjera 1.24, u (1.21), navedena je funkcija v za konkretan makro-program Q iz primjera 1.21.

Iz toga onda slijedi da se pri izvršavanju programa i njegovog spljoštenja zapravo izvršavaju iste instrukcije, samo su im odredišta i redni brojevi transformirani po funkciji v . Iz toga pak slijedi da su semantike tih instrukcija — promjene sadržaja registara — iste i odvijaju se na istim registrima, istim redom. To pak znači da ako počnemo od iste konfiguracije (početna konfiguracija s ulazom \vec{x}) što se registara tiče, registri će mijenjati svoje vrijednosti na isti način prilikom izvršavanja Q i Q^b , te će specijalno i sadržaj registra \mathcal{R}_0 biti isti. Štoviše, jer je $v(n_Q, 0) = n_{Q^b}$, Q^b -izračunavanje s \vec{x} će stati ako i samo ako Q -izračunavanje s \vec{x} stane, te će tada u \mathcal{R}_0 biti isti broj. Kako je \vec{x} bio proizvoljan, zaključujemo da su Q i Q^b ekvivalentni. \square

Teorem 1.26 ima dvije važne posljedice. Prvu možemo uobličiti kao korolar.

Korolar 1.27. *Neka je $k \in \mathbb{N}_+$ i f^k funkcija. Tada je f RAM-izračunljiva ako i samo ako je makro-izračunljiva.*

Dokaz. Za jedan smjer, ako je f RAM-izračunljiva, postoji RAM-algoritam iste mjesnosti P^k koji je računa. RAM-program P je i makro-program, a vidjeli smo u napomeni 1.20 da je svejedno izvršava li se na makro-stroju ili RAM-stroju. Drugim riječima, P na makro-stroju također računa funkciju f , odnosno makro-algoritam P^k računa f , pa je f makro-izračunljiva.

Za drugi smjer, ako je f makro-izračunljiva, postoji makro-algoritam Q^k koji je računa. Po teoremu 1.26, Q^b je ekvivalentan s Q , dakle za svaki k pa specijalno i za mjesnost funkcije f , Q^k i $(Q^b)^k$ (pišemo skraćeno Q^{b^k}) računaju istu funkciju. Drugim riječima, RAM-algoritam Q^{b^k} računa funkciju f , pa je ona RAM-izračunljiva. \square

Napomena 1.28. Druga posljedica teorema 1.26 je programska tehnika koja će bitno povećati izražajnost makro-programa koje pišemo. Rekli smo da je makro uvijek oblika P^* gdje je P RAM-program, no zbog teorema 1.26 smijemo se ponašati kao da P može biti i *makro*-program, koji koristi već napisane makroe. Formalno, pri tome mislimo na P^{b*} , koji ima istu semantiku što se efekata na registrima tiče. \triangleleft

1.4.3. Primjeri makroa

Jedan važan primjer smo već vidjeli: prisjetimo se, za svaki $j \in \mathbb{N}$,

$$(\text{ZERO } \mathcal{R}_j) := \left[\begin{array}{l} 0. \text{ DEC } \mathcal{R}_j, 2 \\ 1. \text{ GO TO } 0 \end{array} \right]^* \text{ ima semantiku } r'_j = 0. \quad (1.22)$$

Nije teško vidjeti, sličnom tehnikom kao za `ZERO`, da za sve *različite* $i, j \in \mathbb{N}$, makro

$$(\text{REMOVE } \mathcal{R}_i \text{ TO } \mathcal{R}_j) := \left[\begin{array}{l} 0. \text{ ZERO } \mathcal{R}_j \\ 1. \text{ DEC } \mathcal{R}_i, 4 \\ 2. \text{ INC } \mathcal{R}_j \\ 3. \text{ GO TO } 1 \end{array} \right]^{b*} \quad (1.23)$$

ima semantiku $r'_i = 0 \wedge r'_j = r_i$: riječima, prebacuje sadržaj \mathcal{R}_i u \mathcal{R}_j i pritom resetira \mathcal{R}_i . Ovdje je bitan uvjet $i \neq j$; pokušajte odrediti što se događa kad taj uvjet nije ispunjen. Općenito ćemo imati uvjete na „parametre” makroa pod kojima on ima traženu semantiku. Naravno, onda smo dužni pri svakom korištenju makroa u programu provjeriti da konkretne vrijednosti parametara zadovoljavaju te uvjete.

Evo primjera makroa s malo kompliciranijim uvjetom: neka su $i, j, n \in \mathbb{N}$ takvi da je $|i - j| \geq n$. Definiramo makro

$$(\text{MMOVE } n \text{ FROM } \mathcal{R}_{i..} \text{ TO } \mathcal{R}_{j..}) := \left[t. \text{ REMOVE } \mathcal{R}_{i+t} \text{ TO } \mathcal{R}_{j+t} \right]_{t < n}^{b*} \quad (1.24)$$

sa semantikom $(\forall t < n)(r'_{j+t} = r_{i+t} \wedge r'_{i+t} = 0)$ — koristimo $(\forall t < n)$ kao pokratu za $(\forall t \in [0..n))$. Riječima, `MMOVE` prebacuje komad memorije duljine n registara počevši od \mathcal{R}_i , na drugo mjesto koje počinje od \mathcal{R}_j , ostavljajući nule na originalnim lokacijama. Svrha korištenja te instrukcije bit će emulacija *stoga* pri funkcijskim pozivima.

Moderna računala rezerviraju poseban dio svoje memorije za stog poziva (*call stack*) koji se dijeli na okvire (*frames*), u kojima se drže podaci o lokalnim varijablama funkcije koja se trenutno izvršava. Pozivom funkcije, pokazivač stoga se pomiče, otvarajući novi okvir u kojem će se funkcija izvršavati. Povratkom iz funkcije, pokazivač stoga se vraća na staro mjesto, eliminirajući taj okvir tako da od njega ostane jedino povratna vrijednost.

Na RAM-arhitekturi nemamo stog poziva kao zasebnu strukturu, ali vidjet ćemo da ga je moguće emulirati pomoću registara. Otvaranje okvira duljine n realizirat ćemo

kao pomak prvih n registara za n mjesta udesno (od \mathcal{R}_n do uključivo \mathcal{R}_{2n-1}), a njegovo zatvaranje kao pomak u suprotnom smjeru. Osigurat ćemo da je n uvijek dovoljno velik da time sačuvamo sve relevantne registre pozivatelja, te da osiguramo dovoljno nulâ na početku za sve relevantne registre pozvane funkcije, tako da je „uvjerimo” da se izvršava na zasebnom RAM-stroju.

Razlog zašto moderna računala ne implementiraju stog na taj način je što je takav pristup nevjerojatno rastrošan, kako u prostoru (troši puno registara) tako i u vremenu (troši puno koraka u računanju). No budući da smo rekli da nas u teoriji izračunljivosti zanima samo (uglavnom konstruktivno) postojanje algoritama, a ne i njihova složenost (pretpostavljamo da na raspolaganju imamo registara i koraka koliko god nam treba), taj pristup će nam biti dovoljno dobar.

Upravo konstruirani makroi su u redu ako nam je prihvatljivo da se registar resetira u postupku prijenosa, ali što ako ga želimo sačuvati? Na prvi pogled, to je neizvedivo. Jedina usporedba koju imamo je ona s nulom u instrukciji tipa DEC, dakle jedini način da unutar RAM-programa saznamo sadržaj registra je da ga dekrementiramo do nule. Na drugi pogled, dekrementiranjem možemo inkrementirati 1 registar (kao u REMOVE), 0 registara (kao u ZERO), ili 2 registra: ako su $i, j, k \in \mathbb{N}$ svi različiti, makro

$$(\text{MOVE } \mathcal{R}_i \text{ TO } \mathcal{R}_j \text{ USING } \mathcal{R}_k) := \left[\begin{array}{l} 0. \text{ ZERO } \mathcal{R}_j \\ 1. \text{ ZERO } \mathcal{R}_k \\ 2. \text{ DEC } \mathcal{R}_i, 6 \\ 3. \text{ INC } \mathcal{R}_j \\ 4. \text{ INC } \mathcal{R}_k \\ 5. \text{ GO TO } 2 \\ 6. \text{ REMOVE } \mathcal{R}_k \text{ TO } \mathcal{R}_i \end{array} \right]^{b*} \quad (1.25)$$

ima semantiku $r'_j = r_i \wedge r'_k = 0$ (primijetimo da nismo napisali r'_i , što u skladu s našom konvencijom znači da je $r'_i = r_i$). To se dokaže po koracima, praćenjem stanja relevantnih registara kroz instrukcije:

	r_i	r_j	r_k
0. ZERO \mathcal{R}_j	r_i	0	r_k
1. ZERO \mathcal{R}_k	r_i	0	0
2.–5. (petlja)	0	r_i	r_i
6. REMOVE \mathcal{R}_k TO \mathcal{R}_i	r_i	r_i	0

(1.26)

Primijetimo da smo za tu operaciju morali „žrtvovati” jedan registar sa strane. Zbog $i \neq k$, uvjet na parametre makroa REMOVE je zadovoljen.

Napomena 1.29. Terminološki detalj: ako ste navikli na rad s modernim sustavima datoteka, vjerojatno smatrate čudnim naziv MOVE za ono što biste vjerojatno intuitivno zvali COPY (dok se ono što biste intuitivno zvali MOVE ovdje zove REMOVE, a ono

što biste zvali `REMOVE` ovdje se zove `ZERO`). Niste jedini: pogledajte recimo [19]. Pravi razlozi su vjerojatno zauvijek izgubljeni u dubinama povijesti, ali je činjenica da moderne računalne arhitekture uglavnom terminološki prate arhitekturu x86, koja standardno instrukciju za kopiranje podataka između registara (ili drugih lokacija) zove `MOV`. Tu terminologiju i mi slijedimo ovdje. \triangleleft

Glavna svrha upravo definirane instrukcije je prijenos argumenata u funkciju: kad pri funkcijskom pozivu otvorimo novi okvir u kojem će se računati pozvana funkcija, želimo u taj okvir na standardna mjesta ulaznih podataka (\mathcal{R}_1 do \mathcal{R}_k , gdje je k mjesnost pozvane funkcije) staviti argumente s kojima je pozvana. S druge strane, želimo da zatvaranjem tog okvira i vraćanjem kontrole pozivatelju, registri koji su poslužili za funkcijski poziv zadrže svoje stare vrijednosti — tako da ih pozvana funkcija može mijenjati bez straha. To je osnovna ideja *prijenosa po vrijednosti*, koji koristi većina imperativnih programskih jezika niže razine (kao što je C), pa čak i moderniji jezici (Java, Ruby) kad se radi o primitivnim tipovima podataka kao što su cijeli brojevi.

U tu svrhu, neka je $k \in \mathbb{N}_+$, te $j_1, j_2, \dots, j_k > k$ prirodni brojevi (ne moraju biti međusobno različiti, ali moraju biti veći od k). Definiramo makro

$$(\text{ARGS } \mathcal{R}_{j_1}, \mathcal{R}_{j_2}, \dots, \mathcal{R}_{j_k}) := \left[\text{t. MOVE } \mathcal{R}_{j_{t+1}} \text{ TO } \mathcal{R}_{t+1} \text{ USING } \mathcal{R}_0 \right]_{t < k}^{b*}, \quad (1.27)$$

čija je semantika $(\forall t \in [1..k])(r'_t = r_{j_t}) \wedge r'_0 = 0$. Primijetimo da je svaki uvjet za parametre od `MOVE` zadovoljen, jer za svaki $t \in [1..k]$ vrijedi $0 < 1 \leq t \leq k < j_t$, pa su \mathcal{R}_0 , \mathcal{R}_t i \mathcal{R}_{j_t} različiti registri. Također primijetimo da u procesu prijenosa argumenata resetiramo izlazni registar, što je u redu jer ionako nakon prijenosa argumenata slijedi prijenos kontrole na pozvanu funkciju, koja će očekivati nulu tamo.

1.4.4. Funkcijski makro

Napokon možemo, kako je najavljeno, definirati makro koji će nam omogućiti funkcijske pozive za bilo koju RAM-izračunljivu funkciju (tako da imamo RAM-program za nju), na bilo kojim registrima kao argumentima, spremajući rezultat u po volji odabran registar, i čuvajući po volji velik početni komad memorije.

Prvo definiramo jedan koristan pojam. Kako svaka RAM-instrukcija djeluje na najviše jednom registru, čitav RAM-program kao konačan niz instrukcija djeluje na konačno mnogo registara. To znači da za svaki RAM-program P postoji tzv. *širina* — najmanji broj $m_P \in \mathbb{N}$ takav da P ne koristi nijedan registar \mathcal{R}_i za $i \geq m_P$. Primijetimo da može biti i $m_P = 0$, ako program uopće ne koristi registre (prazan program, ili onaj koji se sastoji samo od instrukcija tipa `GO TO`).

Za makro-program Q , možemo prirodno definirati $m_Q := m_{Q^b}$ — iako nam to zapravo neće trebati. Ali (RAM- i makro-) algoritmi P^k , pored registara koje koriste u instrukcijama, koriste i registre \mathcal{R}_1 do \mathcal{R}_k za ulazne podatke. Moguće je da bude

$m_P \leq k$, ako računamo funkciju koja ne ovisi o zadnjih nekoliko argumenata. Ipak, registar \mathcal{R}_k jest bitan za postupak računanja te funkcije jer, iako ga ne postavlja nijedna instrukcija, postavlja ga sam rad stroja koji u početnoj konfiguraciji u njega spremi argument x_k . Zato definiramo širinu algoritma kao $m_{P^k} := \max\{m_P, k + 1\}$. Primijetimo da je, zbog $k \in \mathbb{N}_+$, uvijek $m_{P^k} \geq 2$.

Definicija 1.30. Neka je $k \in \mathbb{N}_+$, $f^k \in \text{Comp}_k$, te P_f^k RAM-algoritam koji računa f^k . Neka su $m, j_0, j_1, \dots, j_k \in \mathbb{N}$. Definiramo

$$b := 1 + \max\{m_{P_f}, m, k, j_0, j_1, \dots, j_k\} \quad (1.28)$$

i pomoću njega *funkcijski makro*

$$(P_f(\mathcal{R}_{j_1}, \mathcal{R}_{j_2}, \dots, \mathcal{R}_{j_k}) \rightarrow \mathcal{R}_{j_0} \text{ USING } \mathcal{R}_{m..}) :=$$

$$:= \left[\begin{array}{l} 0. \text{MMOVE } b \text{ FROM } \mathcal{R}_{0..} \text{ TO } \mathcal{R}_{b..} \\ 1. \text{ARGS } \mathcal{R}_{b+j_1}, \mathcal{R}_{b+j_2}, \dots, \mathcal{R}_{b+j_k} \\ 2. P_f^* \\ 3. \text{REMOVE } \mathcal{R}_0 \text{ TO } \mathcal{R}_{b+j_0} \\ 4. \text{MMOVE } b \text{ FROM } \mathcal{R}_{b..} \text{ TO } \mathcal{R}_{0..} \end{array} \right]^{b*} . \quad (1.29)$$

Propozicija 1.31. *Semantika funkcijskog makroa, uz oznake iz definicije 1.30, te pokratu $\vec{r} := (r_{j_1}, r_{j_2}, \dots, r_{j_k})$, jest:*

1. Ako je $\vec{r} \in \mathcal{D}_f$, tada je $r'_{j_0} = f(\vec{r}) \wedge (\forall t \in [b..2b])(r'_t = 0)$.
(Specijalno, zbog $b > m$, za sve $i \in [0..m) \setminus \{j_0\}$ vrijedi $r'_i = r_i$.)
2. Ako $\vec{r} \notin \mathcal{D}_f$, izvršavanje funkcijskog makroa ne stane.

Dokaz. Prvo primijetimo da su svi uvjeti na parametre korištenih makroa zadovoljeni: za prvu i zadnju instrukciju to je $|b - 0| = |0 - b| = b \geq b$, za prijenos argumenata je $b + j_t \geq b > k$, a za prijenos povratne vrijednosti je $b + j_0 \geq b \geq 1 > 0$.

Za tvrdnju 1, pogledajmo redom efekte pojedinih makro-instrukcija iz (1.29).

Nakon prve instrukcije **MMOVE**, u prvih b registara bit će nule, a u idućih b registara bit će *backup* starih vrijednosti prvih b registara $(r_0, r_1, \dots, r_{b-1})$. Konkretno, za svaki $t \in [1..k]$, u \mathcal{R}_{b+j_t} nalazit će se r_{j_t} .

Dakle, instrukcija **ARGS** će u ulazne registre $\mathcal{R}_1, \dots, \mathcal{R}_k$ zapisati upravo vrijednosti \vec{r} . Ostale registre neće mijenjati, pa će u \mathcal{R}_0 i dalje biti 0, kao i u svim registrima \mathcal{R}_i za $i \in [k..b)$, te će u idućih b registara i dalje biti *backup*.

Sada slijedi izvršavanje makroa P_f^* , odnosno RAM-programa P_f na trenutnom stanju registara. Kako je to RAM-program, po napomeni 1.20 slijedi da će imati iste efekte na registre kao da se izvršava na RAM-stroju, a iz $b \geq m_{P_f^k}$ i prethodnog odlomka slijedi da će njegovo izvršavanje biti isto kao da se izvršava na RAM-stroju u početnoj

konfiguraciji. Kako P_f^k računa funkciju f , a u „početnoj” konfiguraciji mu se u ulaznim registrima nalazi \vec{r} , koji je prema pretpostavci element domene \mathcal{D}_f , slijedi da će izvršavanje tog makroa (zapravo P_f -izračunavanje s \vec{r}) stati, i u „završnoj” konfiguraciji sadržaj registra \mathcal{R}_0 će biti $f(\vec{r})$.

Nakon toga izvršavanje funkcijskog makroa prijeći će na instrukciju `REMOVE`, koja će tu vrijednost $f(\vec{r})$ zapisati u registar \mathcal{R}_{b+j_0} , koji se nalazi u bloku $(\mathcal{R}_i)_{i \in [b..2b]}$ jer je $j_0 < b$. Svi ostali registri iz tog bloka i dalje će držati *backup* početnih vrijednosti prvih b registara. Ne znamo što će biti u prvih b registara (osim što će u \mathcal{R}_0 biti 0) jer to ovisi o konkretnom programu P_f , ali zapravo to nije ni bitno.

Naime, zadnja instrukcija `MMOVE` će čitav taj blok prepisati *backup*-blokom, te će se svih b prvih registara vratiti na originalne vrijednosti (konkretno, zanimat će nas da se sačuva prvih $m < b$ registara), osim što će u \mathcal{R}_{j_0} pisati vraćena vrijednost iz \mathcal{R}_{b+j_0} , dakle $f(\vec{r})$. *Backup*-blok (registri od \mathcal{R}_b do \mathcal{R}_{2b-1}) će time biti resetiran. Tablično to možemo prikazati otprilike ovako:

	r_0	r_1	r_k	r_{j_0}	r_b	r_{b+j_0}	r_{2b-1}	
0. <code>MMOVE b FROM $\mathcal{R}_0..$ TO $\mathcal{R}_b..$</code>	0	0	0	0	r_0	r_{j_0}	r_{b-1}	. (1.30)
1. <code>ARGS $\mathcal{R}_{b+j_1}, \mathcal{R}_{b+j_2}, \dots, \mathcal{R}_{b+j_k}$</code>	0	r_{j_1}	r_{j_k}	0	r_0	r_{j_0}	r_{b-1}	
2. P_f^*	$f(\vec{r})$?	?	?	r_0	r_{j_0}	r_{b-1}	
3. <code>REMOVE \mathcal{R}_0 TO \mathcal{R}_{b+j_0}</code>	0	?	?	?	r_0	$f(\vec{r})$	r_{b-1}	
4. <code>MMOVE b FROM $\mathcal{R}_b..$ TO $\mathcal{R}_0..$</code>	r_0	r_1	r_k	$f(\vec{r})$	0	0	0	

Naravno, tablica nije dovoljno precizna za sve mogućnosti: recimo, može biti $j_0 = 1$, ako želimo promijeniti \mathcal{R}_1 *in-place*. No zajedno s gornjim tekstom, tablica pruža dobar uvid u sve što se zbiva pri izvršavanju funkcijskog makroa.

Za tvrdnju 2, svo zaključivanje izgleda isto do trenutka kada moramo zaključiti $\vec{r} \in \mathcal{D}_f$. No u ovom slučaju to ne vrijedi, pa po definiciji računanja funkcije znamo da to znači da P_f -izračunavanje s \vec{r} neće stati. To pak znači da makro-stroj koji izvršava funkcijski makro, pa onda ni RAM-stroj koji izvršava njegovo spljoštenje, neće stati (zapat će u beskonačnoj petlji „na donjoj razini”, izvršavajući instrukciju 2. P_f^*). \square

Definicijom funkcijskog makroa pripremili smo teren za bitno drugačiji model izračunljivosti: *funkcijsku* paradigmu, gdje je puno teže vizualizirati strojeve, algoritme, konfiguracije i izračunavanja, ali je zato puno lakše dokazati da su pojedine konkretne funkcije izračunljive (pokušajte recimo dokazati da je skup \mathbb{P} RAM-izračunljiv pisanjem RAM-programa za $\chi_{\mathbb{P}}^1$). Dokazom ekvivalentnosti ta dva modela imat ćemo onda najbolje od oba svijeta.

Poglavlje 2.

Rekurzivne funkcije

Iz uglavnom povijesnih razloga, prvi doticaj s programiranjem većine ljudi bude kroz *imperativno* programiranje: algoritmi kao *programi*, nizovi *naredaba* koje mijenjaju stanje neke zajedničke *memorije* nad kojom se izvršavaju. Velik broj *mainstream* programskih jezika spada u tu paradigmu: gotovo svi jezici niske razine, C, C++, Python, Rust, ... (Java preko tog imperativnog sloja prostire „objektno-orijentirani veo”, ali fundamentalno, provođenje algoritma je i dalje izvršavanje naredaba i mijenjanje memorije). Kontrola toka (izvršavanje određenih naredbi nula ili više puta, što može ovisiti o stanju memorije) se u takvim jezicima obično iskazuje *skokovima* (uvjetnim ili bezuvjetnim, kao što su `DEC` ili `GO TO` u RAM-stroju), ili na višoj razini, *petljama* koje mijenjaju *kontrolnu varijablu* i testiraju je da bi ustanovile trebaju li se nastaviti izvršavati, ili zaustaviti.

Ipak, postoji i drugi pristup: matematičke formule kojima se definiraju funkcije često mogu poslužiti kao vrsta algoritama za njihovo računanje. Važno je primijetiti da se u tom slučaju nikakve vrijednosti ne mijenjaju: $f(x, y) := x^2 + 3y + 2$ nije naredba koja mijenja x niti y , već matematička definicija, koja kazuje kako izračunati vrijednosti funkcije f , recimo na prirodnim brojevima, ako ih znamo potencirati, množiti i zbrajati.

U toj paradigmi, umjesto praznog programa, aksiomatski je zadano da je nulfunkcija izračunljiva. Umjesto instrukcije `INC` koja mijenja sadržaj registra na kojem djeluje, imamo funkciju *sljedbenika*, koja proizvodi novi prirodni broj koji je sljedbenik ulaznog podatka. Umjesto ulaznih registara, imamo *koordinatne projekcije* koje vraćaju pojedini ulazni podatak. Umjesto pomoćnih registara imamo pomoćne funkcije, kojima korak po korak gradimo ono što nam treba. Umjesto slijednog izvršavanja naredaba ovdje imamo *kompoziciju*, kojom npr. iz funkcija `add`³, `mul`² i `pow`², te konstanti C_2^2 i C_3^3 , i koordinatnih projekcija l_1^2 i l_2^2 , dobivamo funkciju f iz prethodnog odlomka. Umjesto grananja ovdje imamo definiciju funkcije *po slučajevima*, gdje su uvjeti disjunktne relacije (vrijedi najviše jedan od njih — ako ne vrijedi nijedan, funkcija nije definirana). Umjesto petlji (primijetimo da ne možemo mijenjati kontrolnu varijablu), funkcijsko programiranje koristi *rekurziju*, kao način da se elegantno opiše izračunavanje funkcija više puta s različitim vrijednostima argumenata, a da nikakve varijable pritom ne mijenjaju svoje vrijednosti.

Specijalni slučaj — *primitivna* rekurzija — odgovara petljama koje se izvršavaju unaprijed određen broj puta, i kao takve ne mogu biti beskonačne. To znači da algoritmi dobiveni primitivnom rekurzijom uvijek stanu, te su funkcije koje oni računaju (*primitivno rekurzivne* funkcije) uvijek totalne. To je dobro za programiranje konkretnih funkcija, ali vidjeli smo već u uvodu da totalni algoritmi nisu dovoljni da bi opisali *sve* izračunljive funkcije. Zato nam treba *opća rekurzija*, odnosno sasvim općenit način da unutar definicije neke funkcije koristimo (najčešće pozivamo) istu tu funkciju. Do razvoja te tehnike ima još puno, te ćemo za početak uvesti jedan specijalni oblik koji odgovara *minimizaciji* relacije — traženju najmanjeg prirodnog broja s nekim svojstvom. To neće nužno dati totalnu funkciju — recimo u slučaju prazne relacije — ali ponekad hoće. Funkcije nastale tim postupkom iz izračunljivih relacija (eventualno još komponirane s nekim izračunljivim funkcijama) zovemo *parcijalno rekurzivnim* funkcijama, a one među njima koje su totalne zovemo jednostavno *rekurzivnim* funkcijama. Naša intuicija o nužnosti razmatranja parcijalnih funkcija da bismo dobili sve totalne izračunljive funkcije, sada se može formalizirati kao: postoje rekurzivne funkcije koje nisu primitivno rekurzivne. Dokaz te tvrdnje može se pronaći recimo u [16, dodatak]. Važniji rezultat, koji ćemo dokazati, je da se skup parcijalno rekurzivnih funkcija podudara sa skupom Comp RAM-izračunljivih funkcija.

Ako želimo dobivati izračunljive funkcije slaganjem (kompozicijom, primitivnom rekurzijom, ...) jednostavnijih funkcija, odnekud moramo početi: neke *najjednostavnije* brojevne funkcije moramo aksiomatski prihvatiti kao izračunljive. S jedne strane, te će funkcije biti toliko jednostavne da neće biti sumnje u njihovu izračunljivost, a s druge strane, moći ćemo navesti i formalniji razlog zašto ih smatramo izračunljivima u ovom modelu: dokazat ćemo da su makro-izračunljive, pa time i RAM-izračunljive.

Definicija 2.1. *Inicijalne funkcije* su sljedeće:

- *nulfunkcija* Z^1 , zadana sa $Z(x) := 0$;
- *sljedbenik* Sc^1 , zadana sa $Sc(x) := x + 1$;
- za svaki $k \in \mathbb{N}_+$, za svaki $n \in [1..k]$, n -ta k -mjesna koordinatna projekcija I_n^k , zadana sa $I_n(\vec{x}^k) := I_n(x_1, x_2, \dots, x_k) := x_n$. \triangleleft

Treća stavka zapravo govori da su sve identitete $id_{\mathbb{N}^k}$ izračunljive, samo u skladu s napomenom 1.1 svaku $id_{\mathbb{N}^k}$ prikazujemo kao k koordinatnih funkcija $I_n^k, n \in [1..k]$ s istim ulaznim podacima.

Napomena 2.2. Sve inicijalne funkcije su totalne: $\mathcal{D}_Z = \mathcal{D}_{Sc} = \mathbb{N}$, a $\mathcal{D}_{I_n^k} = \mathbb{N}^k$. \triangleleft

Vidimo da inicijalnih funkcija zapravo ima beskonačno (prebrojivo) mnogo, ali su samo tri moguća tipa. Nulfunkcija i sljedbenik pružaju nam mogućnost kompozicijom uhvatiti sve prirodne brojeve u sustavu (kao konstantne jednomjesne funkcije), a

koordinatne projekcije pružaju mogućnost individualnog rada sa svakim pojedinim argumentom funkcije prema njegovom rednom broju n (od k njih ukupno).

Propozicija 2.3. *Svaka inicijalna funkcija je makro-izračunljiva.*

Dokaz. Već smo vidjeli da prazan (makro ili RAM) program (formalno, $[]^1$) računa funkciju Z . Semantika instrukcije **REMOVE** odmah nam daje da za sve $1 \leq n \leq k$, makro-algoritam $[0. \text{REMOVE } \mathcal{R}_n \text{ TO } \mathcal{R}_0]^k$ računa I_n^k . Jednako tako, doista nije teško

vidjeti da makro-algoritam $\left[\begin{array}{l} 0. \text{REMOVE } \mathcal{R}_1 \text{ TO } \mathcal{R}_0 \\ 1. \text{INC } \mathcal{R}_0 \end{array} \right]^1$ računa Sc :

$$\begin{array}{c|cc} & \mathcal{R}_0 & \mathcal{R}_1 \\ \hline & 0 & x \\ 0. \text{REMOVE } \mathcal{R}_1 \text{ TO } \mathcal{R}_0 & x & 0 \\ 1. \text{INC } \mathcal{R}_0 & x+1 & 0 \end{array} \quad \square \quad (2.1)$$

Korolar 2.4. *Svaka inicijalna funkcija je RAM-izračunljiva.*

Dokaz. Direktno iz propozicije 2.3 i korolara 1.27. Konkretno, spljoštenja makro-programa iz dokaza propozicije 2.3 daju RAM-algoritme za inicijalne funkcije. \square

2.1. Kompozicija

Kompozicija je intuitivno vrlo jednostavna operacija (baš kao i slijedno izvršavanje naredaba u imperativnim programima): izračunamo vrijednost jedne funkcije, i uvrstimo je u definiciju druge funkcije. Ipak, precizna definicija je dosta tehnička, između ostalog zbog nekih odluka koje smo donijeli na početku, o tome kako ćemo reprezentirati algoritme.

Konkretno, htjeli bismo reći: „kompozicija $H \circ G$ dviju izračunljivih funkcija, G^k i H^l , je izračunljiva funkcija”. Ali da bi ta kompozicija imala smisla, kodomena od G^k bi morala biti \mathbb{N}^l . U skladu s napomenom 1.1, to zapravo znači da imamo l izračunljivih koordinatnih funkcija, G_1^k, \dots, G_l^k , i svaka od njih daje po jedan ulazni podatak za funkciju H^l . Drugim riječima, komponiranje više nije nužno binarna operacija: zato kompoziciju označavamo $H \circ (G_1, \dots, G_l)$.

Drugi tehnički detalj je suptilniji, i tiče se domene tako definirane kompozicije. Da bismo uočili problem, promotrimo kompoziciju $F^1 := I_1^2 \circ (Z^1, \emptyset^1)$. Ta funkcija je svakako izračunljiva, ali *što* je ona? Konkretno, koliko je $F(5)$? S jedne strane, čini se da je $F(5) = I_1(Z(5), \emptyset(5)) = Z(5) = 0$. Općenito bi se to dogodilo za svaki ulaz, pa bismo zaključili $F = Z$. S druge strane, ako to doista pokušamo mehanički *izračunati* na intuitivno očit način, vidjet ćemo problem: da bismo izračunali $F(5)$, moramo izračunati $Z(5) =: y_1$, zatim „izračunati” $\emptyset(5) =: y_2$, i na kraju izračunati $I_1(y_1, y_2) = y_1$. U tom smislu, algoritam za računanje F ne stane s ulazom 5 (niti s

ikojim drugim ulazom), jer njegov drugi korak ne stane. Dakle po tome bi bilo $F = \emptyset$. Što je od toga?

Ta dilema je dobro poznata u modernom računarstvu, i oba pristupa nalazimo u današnjim programskim jezicima. Prvi pristup zove se *lijena* evaluacija (*lazy evaluation*) i koriste ga neki čisto funkcijski jezici poput Haskell. Prednost je bogatija semantika (više izraza ima smisla), te činjenica da beskonačne strukture nisu nužno problem ako nam treba samo njihov konačni početak. Pogledajmo kako to izgleda u Haskellu:

```
Prelude> let i12(x, y) = x
Prelude|      z(x) = 0
Prelude|      prazna(x) = undefined
Prelude|      f(x) = i12(z(x), prazna(x))
Prelude| in f(5)
0
```

Drugi pristup zove se *marljiva* evaluacija (*eager evaluation*) i prisutan je u gotovo svim imperativnim jezicima, pa i mnogim funkcijskima (kao što je ML). Prednost je lakša implementacija, i lakše razmišljanje o kodu (a time i lakši *debugging*).

```
>>> def i12(x, y): return x
>>> def z(x): return 0
>>> def prazna(x):
...     while True: pass
>>> def f(x): return i12(z(x), prazna(x))
>>> f(5)
^C
KeyboardInterrupt
```

Važno je napomenuti da, kao i inače kad je riječ o implementaciji algoritama (princip opće izračunljivosti), bilo koji od tih pristupa može *simulirati* onaj drugi. S obzirom na to da nas performanse ne zanimaju, odabrat ćemo **marljivu** evaluaciju jer je jednostavnija za implementaciju (u našem slučaju, jednostavnija za dokaz ekvivalentnosti s RAM-izračunljivošću, odnosno za konstrukciju kompajlera u RAM-programe), a poslije ćemo opisati kako simulirati lijenu evaluaciju kad nam bude trebala. Jedno od mjesta gdje će nam svakako trebati je implementacija grananja gdje nisu sve grane totalne, poput onog što radi operator $?:$ u programskom jeziku C; recimo, $1?z(5):prazna(5)$ ima vrijednost 0. U našoj terminologiji, to će biti funkcija definirana po slučajevima iz parcijalno rekurzivnih funkcija. U početku ćemo se baviti samo totalnim (primitivno rekurzivnim i rekurzivnim) funkcijama, pa nam to neće trebati — no dobro je odmah pravilno formalizirati domenu kompozicije, da ne bismo morali poslije mijenjati definiciju.

Dakle, želimo da je $H \circ (G_1, \dots, G_l)$ definirana u \vec{x} ako je *svaka* G_i definirana u \vec{x} , te ako označimo $g_i := G_i(\vec{x})$, još je H definirana u \vec{g}^l . Matematički rečeno, u domeni kompozicije su sve one k -torke iz presjeka domena pojedinih G_i , takve da je l -torka njihovih vrijednosti element domene od H .

Definicija 2.5. Neka su $k, l \in \mathbb{N}_+$, te neka su $G_1^k, G_2^k, \dots, G_l^k$ i H^l funkcije. Za funkciju F^k definiranu s

$$\mathcal{D}_F := \left\{ \vec{x} \in \bigcap_{i=1}^l \mathcal{D}_{G_i} \mid (G_1(\vec{x}), G_2(\vec{x}), \dots, G_l(\vec{x})) \in \mathcal{D}_H \right\}, \quad (2.2)$$

$$F(\vec{x}) := H(G_1(\vec{x}), G_2(\vec{x}), \dots, G_l(\vec{x})), \text{ za sve } \vec{x} \in \mathcal{D}_F, \quad (2.3)$$

kažemo da je dobivena *kompozicijom* iz funkcija G_1, G_2, \dots, G_l i H . Skraćeno pišemo $F := H \circ (G_1, G_2, \dots, G_l)$.

Za skup funkcija \mathcal{F} kažemo da je *zatuoren na kompoziciju* ako za sve mjesnosti $k, l \in \mathbb{N}_+$, za sve k -mjesne $G_1, G_2, \dots, G_l \in \mathcal{F}$, te za sve l -mjesne $H \in \mathcal{F}$, vrijedi $H \circ (G_1, G_2, \dots, G_l) \in \mathcal{F}$. \triangleleft

U skladu s napomenom 1.3, izraze (2.2) i (2.3) zajedno skraćeno pišemo kao

$$F(\vec{x}) := H(G_1(\vec{x}), G_2(\vec{x}), \dots, G_l(\vec{x})), \quad (2.4)$$

gdje podrazumijevamo da izraz s ugniježđenim funkcijskim pozivima $f(\vec{v}^k)$ ima smisla ako (rekurzivno) svaki v_i ima smisla, te je k -torka njihovih vrijednosti element od \mathcal{D}_f .

Dakle, kod komponiranja općenitih parcijalnih funkcija moramo voditi računa o domeni, ali dobro je vidjeti da kompozicija sama po sebi ne može narušiti totalnost.

Propozicija 2.6. *Svaka kompozicija totalnih funkcija je totalna.*

Dokaz. Neka su $k, l \in \mathbb{N}_+$, te neka su $G_1^k, G_2^k, \dots, G_l^k, H^l$ totalne funkcije. To znači da je $\mathcal{D}_{G_i} = \mathbb{N}^k$ za sve $i \in [1..l]$, te $\mathcal{D}_H = \mathbb{N}^l$. Uvrštavajući to u (2.2) dobijemo

$$\mathcal{D}_{H \circ (G_1, \dots, G_l)} = \{ \vec{x} \in \mathbb{N}^k \mid (G_1(\vec{x}), \dots, G_l(\vec{x})) \in \mathbb{N}^l \} = \mathbb{N}^k, \quad (2.5)$$

pa je $H \circ (G_1, \dots, G_l)$ totalna. \square

Primjer 2.7. $C_2^3 = Sc \circ Sc \circ Z \circ I_1^3$. Doista,

$$\begin{aligned} (Sc \circ Sc \circ Z \circ I_1^3)(x, y, z) &= (Sc \circ Sc \circ Z)(I_1^3(x, y, z)) = (Sc \circ Sc \circ Z)(x) = \\ &= (Sc \circ Sc)(Z(x)) = (Sc \circ Sc)(0) = Sc(Sc(0)) = Sc(1) = 2 = C_2^3(x, y, z). \end{aligned} \quad (2.6)$$

Sve konstante C_n^k se mogu slično tako prikazati (propozicija 2.19). \triangleleft

2.1.1. RAM-izračunljivost kompozicije

Lema 2.8. *Skup RAM-izračunljivih funkcija, Comp , zatvoren je na kompoziciju.*

Dokaz. Pomoću funkcijskog makroa i uz marljivu evaluaciju, algoritam je očit: prvo izračunamo sve G_i u istim argumentima \vec{x} , spremimo njihove povratne vrijednosti u l pomoćnih registara nakon ulaznih, te na kraju izračunamo H s tako dobivenim argumentima.

Precizno, neka su $k, l \in \mathbb{N}_+$ proizvoljni, te neka su $G_1^k, G_2^k, \dots, G_l^k$ i H^l proizvoljne RAM-izračunljive funkcije. To znači da postoje RAM-algoritmi $P_{G_1}^k, P_{G_2}^k, \dots, P_{G_l}^k$ i P_H^l , koji ih redom računaju. Tvrdimo da makro-program

$$Q_F := \left[\begin{array}{l} 0. P_{G_1}(\mathcal{R}_1, \mathcal{R}_2, \dots, \mathcal{R}_k) \rightarrow \mathcal{R}_{k+1} \text{ USING } \mathcal{R}_{k+l+1}.. \\ 1. P_{G_2}(\mathcal{R}_1, \mathcal{R}_2, \dots, \mathcal{R}_k) \rightarrow \mathcal{R}_{k+2} \text{ USING } \mathcal{R}_{k+l+1}.. \\ \vdots \\ (l-1). P_{G_l}(\mathcal{R}_1, \mathcal{R}_2, \dots, \mathcal{R}_k) \rightarrow \mathcal{R}_{k+l} \text{ USING } \mathcal{R}_{k+l+1}.. \\ l. P_H(\mathcal{R}_{k+1}, \mathcal{R}_{k+2}, \dots, \mathcal{R}_{k+l}) \rightarrow \mathcal{R}_0 \text{ USING } \mathcal{R}_{k+l+1}.. \end{array} \right] \quad (2.7)$$

računa $F := H \circ (G_1, G_2, \dots, G_l)$. Prema definiciji 1.10, neka je $\vec{x} \in \mathbb{N}^k$ proizvoljan. Ako je $\vec{x} \in \mathcal{D}_F$, prema definiciji (2.2) je za sve $i \in [1..l]$, $\vec{x} \in \mathcal{D}_{G_i}$, pa izvršavanje svake od prvih l instrukcija programa Q_F stane, i u \mathcal{R}_{k+i} zapiše vrijednost $G_i(\vec{x}) =: y_i$. Pritom je važno da će, prema propoziciji 1.31, ulazni podaci ostati u ulaznim registrima, te računanje kasnijih y_j neće uništiti one ranije izračunate. Uvjet $\vec{y}^l \in \mathcal{D}_H$, također iz (2.2), znači da će i izvršavanje posljednje instrukcije u Q_F stati, i u \mathcal{R}_0 zapisati $H(\vec{y}) = F(\vec{x})$. Tablično,

	\mathcal{R}_0	\mathcal{R}_1	\mathcal{R}_k	\mathcal{R}_{k+1}	\mathcal{R}_{k+2}	\mathcal{R}_{k+l}	$\mathcal{R}_{k+l+1}..$
	0	x_1	x_k	0	0	0	0
0. $P_{G_1} \dots$	0	x_1	x_k	$G_1(\vec{x})$	0	0	?
1. $P_{G_2} \dots$	0	x_1	x_k	$G_1(\vec{x})$	$G_2(\vec{x})$	0	?
\vdots							
$(l-1). P_{G_l} \dots$	0	x_1	x_k	$G_1(\vec{x})$	$G_2(\vec{x})$	$G_l(\vec{x})$?
$l. P_H \dots$	$F(\vec{x})$	x_1	x_k	$G_1(\vec{x})$	$G_2(\vec{x})$	$G_l(\vec{x})$?

(2.8)

S druge strane, ako $\vec{x} \notin \mathcal{D}_F$, prema (2.2) to se može dogoditi na dva načina. Ako $\vec{x} \notin \bigcap_{i=1}^l \mathcal{D}_{G_i}$, postoji neki i takav da $\vec{x} \notin \mathcal{D}_{G_i}$, pa označimo s i_0 najmanji takav. Tada će izvršavanje programa Q_F doći do instrukcije s rednim brojem $i_0 - 1$, ali izvršavanje te instrukcije neće nikada stati prema propoziciji 1.31, pa ni Q_F -izračunavanje s \vec{x} neće stati. Ako pak l -torka dobivenih povratnih vrijednosti nije u \mathcal{D}_H , izvršavanje Q_F će zapeti u beskonačnoj petlji na posljednjoj instrukciji, pa opet Q_F -izračunavanje s \vec{x} neće stati. Sada prema teoremu 1.26 RAM-algoritam $Q_F^{b^k}$ također računa F , pa je $F \in \text{Comp}$. \square

2.2. Primitivna rekurzija

Iz upravo napisanog dokaza je jasno kako točno kompozicija odgovara slijednom izvršavanju naredbi. Ipak, već za vrlo jednostavne funkcije kao što je zbrajanje, trebaju nam *petlje*, odnosno način da određene naredbe izvršimo neki broj puta koji ovisi o ulaznim podacima. Rekli smo da u funkcijskom programiranju tome odgovaraju rekurzije, no opće rekurzije su vrlo komplicirane za implementaciju, a osim toga teško je ustanoviti kada točno daju totalne funkcije. Zato ćemo aksiomatski propisati samo jedan jednostavan oblik rekurzije, koji odgovara petljama čiji je broj ponavljanja poznat prije početka njihovog izvršavanja. Tek kasnije ćemo vidjeti kako se u ovom modelu mogu definirati izračunljive funkcije kao rješenja općih rekurzija.

„Broj ponavljanja je poznat” ne djeluje dovoljno precizno — zapravo, broj ponavljanja bi trebao biti *izračunljiv* nekom jednostavnijom funkcijom B istog tipa. Da bismo to omogućili, dat ćemo broj ponavljanja kao dodatni argument y našoj funkciji, te ćemo kompozicijom s B na mjestu tog argumenta dobiti željeni broj ponavljanja. Primjer te tehnike vidjet ćete u napomeni 2.51.

Da bismo funkcionalno opisali petlju, moramo opisati *inicijalizaciju* G , koja odgovara stanju prije početka izvršavanja petlje, te *tijelo* (ponekad ga zovemo i *korak*) petlje kao funkciju H koja preslikava stanje na početku jednog prolaza kroz petlju u stanje na kraju tog prolaza. Vrlo grubo rečeno, želimo modelirati funkciju $G \text{ } \mathbb{R} \text{ } H := H \circ H \circ \dots \circ H \circ G$, gdje je broj H -ova zadan kao posebni argument y .

Ipak, dozvolit ćemo još jednu stvar koja je uobičajena u petljama: tijelu petlje H kao još jedan argument prenijet ćemo „kontrolnu varijablu”, koja broji koliko puta smo već prošli kroz petlju. Imperativni jezici niže razine obično implementiraju „primitivne” petlje kontrolnom varijablom koja ide od 0 do isključivo y , recimo C :

```
for(i=0; i<y; ++i) /* ovdje možemo koristiti i */; (2.9)
```

Definicija 2.9. Neka je $k \in \mathbb{N}_+$, te neka su G^k i H^{k+2} totalne funkcije. Za funkciju F^{k+1} definiranu s

$$F(\vec{x}, 0) := G(\vec{x}), \quad (2.10)$$

$$F(\vec{x}, y + 1) := H(\vec{x}, y, F(\vec{x}, y)), \text{ za sve } y \in \mathbb{N}, \quad (2.11)$$

kažemo da je dobivena *primitivnom rekurzijom* iz funkcija G i H . Skraćeno pišemo $F := G \text{ } \mathbb{R} \text{ } H$. Smatramo da operator $\text{ } \mathbb{R} \text{ }$ ima niži prioritet od \circ .

Za skup funkcija \mathcal{F} kažemo da je *zatvoren na primitivnu rekurziju* ako za svaki $k \in \mathbb{N}_+$, za svaku totalnu k -mjesnu funkciju $G \in \mathcal{F}$, te za svaku totalnu $(k + 2)$ -mjesnu funkciju $H \in \mathcal{F}$, vrijedi $G \text{ } \mathbb{R} \text{ } H \in \mathcal{F}$. \triangleleft

Napomena 2.10. Iz Dedekindovog teorema rekurzije [18] slijedi da je jednadžbama (2.10) i (2.11) zadana jedinstvena totalna funkcija. Također, primitivna rekurzija je *definirana* samo za totalne funkcije, pa ne moramo pričati o domenama od G , H i $G \text{ } \mathbb{R} \text{ } H$. \triangleleft

Primijetimo jedan tehnički problem: budući da ne promatramo brojeve funkcije s nula ulaznih podataka, funkcija G koja zadaje početni uvjet mora biti barem jednomjesna, a onda funkcija F dobivena primitivnom rekurzijom mora biti bar dvomjesna. Naravno, htjeli bismo i jednomjesne funkcije definirati primitivnom rekurzijom: kanonski primjer je vjerojatno faktorijel,

$$0! := 1, \quad (2.12)$$

$$(n+1)! := (n+1) \cdot n!, \quad (2.13)$$

samo zasad ne možemo reći da je funkcija `factorial`¹ ($n \mapsto n!$) dobivena primitivnom rekurzijom, jer ne postoji odgovarajuća funkcija G . Recimo, za $G := C_1^1$, i odgovarajuću funkciju H^3 , definicija 2.9 bi nam dala `factorial`², što očito nije funkcija koju tražimo. (Ali nije ni toliko daleko od nje, što ćemo vidjeti kasnije.)

Definicija višemjesnih funkcija primitivnom rekurzijom sasvim lijepo funkcionira.

Primjer 2.11. $\text{add}^2 = I_1^1 \text{ } \bowtie \text{ } Sc \circ I_3^3$. Doista, zbrajanje dva broja možemo prikazati kao

$$x + 0 = x \quad \text{add}(x, 0) = I_1^1(x), \quad (2.14)$$

$$x + (y + 1) = (x + y) + 1 \quad \text{add}(x, y + 1) = (Sc \circ I_3^3)(x, y, \text{add}(x, y)). \quad (2.15)$$

Jednako tako, lako je pokazati $\text{mul}^2 = Z \text{ } \bowtie \text{ } \text{add}^2 \circ (I_1^1, I_3^3)$. (Sami napišite `pow`!) \triangleleft

Vidimo da se mnoge funkcije mogu napisati koristeći samo inicijalne funkcije, kompoziciju i primitivnu rekurziju — drugim riječima, primitivno su rekurzivne. No prije formalizacije tog pojma, dokažimo da se primitivna rekurzija može izvršavati na RAM-stroju.

Lema 2.12. *Skup Comp je zatvoren na primitivnu rekurziju.*

Dokaz. Neka je $k \in \mathbb{N}_+$, te neka su $G^k, H^{k+2} \in \text{Comp}$ totalne funkcije. One su RAM-izračunljive, pa postoje RAM-algoritmi P_G^k i P_H^{k+2} koji ih redom računaju. Želimo naći program koji računa funkciju $F^{k+1} := G \text{ } \bowtie \text{ } H$. Dakle, u registrima \mathcal{R}_1 do \mathcal{R}_k se nalazi \vec{x} , dok se u \mathcal{R}_{k+1} nalazi y , broj ponavljanja petlje odnosno broj izračunavanja funkcije H . Prvo, prije petlje moramo izračunati funkciju G na prvih k ulaznih registara. Rezultat između prolazaka kroz petlju držat ćemo u \mathcal{R}_0 , tako da završetkom petlje već bude spreman kao izlazni podatak. Petlju pišemo na standardni način koji smo već vidjeli u makroima `ZERO`, `REMOVE`, `MOVE`, i programu P_{add^3} . Kontrolnu varijablu držimo u pomoćnom registru \mathcal{R}_{k+2} — primijetimo da za to ne možemo koristiti \mathcal{R}_{k+1} jer idu u suprotnim smjerovima: svakim izvršavanjem petlje \mathcal{R}_{k+1} se dekrementira, dok se kontrolna varijabla mora inkrementirati (jer broji koliko puta smo izvršili petlju).

Primijetimo još da kontrolnu varijablu ne treba inicijalizirati: kako \mathcal{R}_{k+2} nije ulazni registar, na početku izračunavanja njegov sadržaj već jest 0. Također, inkrementiranjem

kontrolne varijable *nakon* izvođenja petlje (funkcijski makro s P_H) postizemo brojenje od nule; inkrementiranjem prije računanja H postigli bismo brojenje od 1 do uključivo y . Programski jezici koji broje od nule obično zahtijevaju inkrement poslije: recimo, standard jezika C propisuje da se u (2.9) inkrement $++i$ obavi nakon tijela petlje.

Dakle, tvrdimo da makro-program

$$Q_F := \left[\begin{array}{l} 0. P_G(\mathcal{R}_1, \mathcal{R}_2, \dots, \mathcal{R}_k) \rightarrow \mathcal{R}_0 \text{ USING } \mathcal{R}_{k+3}.. \\ 1. \text{DEC } \mathcal{R}_{k+1}, 5 \\ 2. P_H(\mathcal{R}_1, \mathcal{R}_2, \dots, \mathcal{R}_k, \mathcal{R}_{k+2}, \mathcal{R}_0) \rightarrow \mathcal{R}_0 \text{ USING } \mathcal{R}_{k+3}.. \\ 3. \text{INC } \mathcal{R}_{k+2} \\ 4. \text{GO TO } 1 \end{array} \right] \quad (2.16)$$

računa F . U tu svrhu, po definiciji 1.10, neka je $(\vec{x}, y) \in \mathbb{N}^{k+1}$ proizvoljan. Na početku Q_F -izračunavanja s (\vec{x}, y) imamo makro-konfiguraciju $(0, \vec{x}, y, 0, 0, \dots, 0, 0)$, koja izvršavanjem funkcijskog makroa rednog broja 0 prelazi u $(G(\vec{x}), \vec{x}, y, 0, ?, \dots, 1, 0)$ (jer je G totalna) — ne znamo sadržaj registara od \mathcal{R}_{k+3} nadalje, ali nam nije ni bitan. Broj u \mathcal{R}_0 je prema (2.10) jednak $F(\vec{x}, 0)$. Sada ako za sve $i \in [0..y]$, s d_i označimo oblik konfiguracije $(F(\vec{x}, i), \vec{x}, y - i, i, ?, \dots, 1, 0)$, upravo smo došli do konfiguracije oblika d_0 .

Tvrdimo da svaka konfiguracija oblika d_i za $i < y$, prelazi u konačno mnogo koraka u neku konfiguraciju oblika d_{i+1} . Doista, iz $i < y$ slijedi $y - i > 0$, pa imamo

$$\begin{aligned} d_i &= (F(\vec{x}, i), \vec{x}, y - i, i, ?, \dots, 1, 0) \rightsquigarrow^* (F(\vec{x}, i), \vec{x}, y - i - 1, i, ?, \dots, 2, 0) \rightsquigarrow^* \\ &\rightsquigarrow^* (H(\vec{x}, i, F(\vec{x}, i)), \vec{x}, y - i - 1, i, ?, \dots, 3, 0) \rightsquigarrow \\ &\rightsquigarrow (F(\vec{x}, i + 1), \vec{x}, y - i - 1, i + 1, ?, \dots, 4, 0) \rightsquigarrow \\ &\rightsquigarrow (F(\vec{x}, i + 1), \vec{x}, y - (i + 1), i + 1, ?, \dots, 1, 0) = d_{i+1}. \end{aligned} \quad (2.17)$$

Naravno, ovdje je bilo važno da je H totalna, pa računanje vrijednosti $H(\vec{x}, i, F(\vec{x}, i))$ (koja je prema (2.11) jednaka $F(\vec{x}, i + 1)$) doista stane nakon konačno mnogo koraka, reprezentiranih s \rightsquigarrow^* u (2.17). Sada odmah indukcijom po i slijedi da za svaki $i \leq y$, Q_F -izračunavanje s (\vec{x}, y) sadrži konfiguraciju oblika d_i . Specijalno, u njemu postoji konfiguracija oblika d_y , koja prelazi u završnu konfiguraciju

$$d_y = (F(\vec{x}, y), \vec{x}, y - y, y, ?, \dots, 1, 0) \rightsquigarrow (F(\vec{x}, y), \vec{x}, 0, y, ?, \dots, 5, 0), \quad (2.18)$$

oblika $(F(\vec{x}, y), \dots)$, što smo i trebali. Sada prema teoremu 1.26, RAM-algoritam $(Q_F^b)^{k+1}$ također računa F , pa je $F \in \text{Comp}$. \square

Primijetimo da smo na ovaj način posredno dobili i RAM-program za add^2 , pa time i za mul^2 , a onda i pow (pogledajte primjer 2.11). Naravno, takav RAM-program ima jako puno instrukcija i nikad ga tako ne bismo „ručno” napisali, ali to upravo pokazuje snagu funkcijske paradigme: funkcijski programi se bolje slažu u veće cjeline nego RAM-programi (to svojstvo se u programskim jezicima obično zove *composability*).

2.2.1. Primitivno rekurzivne funkcije

Već smo nekoliko puta spomenuli primitivno rekurzivne funkcije. Vrijeme je da taj pojam formalno definiramo.

Definicija 2.13. Skup *primitivno rekurzivnih* funkcija je najmanji skup funkcija koji sadrži sve inicijalne funkcije, i zatvoren je na kompoziciju i na primitivnu rekurziju. \triangleleft

Naravno, da bi definicija bila dobra, trebalo bi vidjeti da postoji *ikoji* takav skup (tada postoji i najmanji kao presjek svih takvih), no skup svih brojevnih funkcija $\text{Func} := \bigcup_{k \in \mathbb{N}_+} \text{Func}_k$ očito zadovoljava uvjete.

Definicija 2.13 je elegantna i matematički pogodna za dokazivanje, ali nije baš operativna. Za konkretne funkcije, lakše je dokazati da su primitivno rekurzivne korištenjem sljedeće karakterizacije.

Propozicija 2.14. *Neka je f brojevena funkcija. Tada je f primitivno rekurzivna ako i samo ako se može dobiti iz inicijalnih funkcija pomoću konačno mnogo primjena kompozicije i primitivne rekurzije.*

Dokaz. Označimo sa \mathcal{S} skup funkcija koje se mogu dobiti iz inicijalnih pomoću konačno mnogo primjena kompozicije i primitivne rekurzije. Ako je f inicijalna funkcija, očito je u \mathcal{S} jer ju je moguće dobiti iz inicijalnih pomoću 0 primjena kompozicije i primitivne rekurzije.

Ako su $k, l \in \mathbb{N}_+$, te $G_1^k, \dots, G_l^k, H^l \in \mathcal{S}$, tada se svaka G_i može dobiti iz inicijalnih pomoću, recimo, c_i primjena kompozicije i r_i primjena primitivne rekurzije. Također se H može dobiti iz inicijalnih, recimo pomoću c primjena kompozicije i r primjena primitivne rekurzije. Tada se $H \circ (G_1, \dots, G_l)$ može dobiti iz inicijalnih funkcija pomoću najviše $1 + c + \sum_{i=1}^l c_i$ primjena kompozicije, i najviše $r + \sum_{i=1}^l r_i$ primjena primitivne rekurzije, dakle konačno mnogo, pa je $H \circ (G_1, \dots, G_l) \in \mathcal{S}$.

Ako je $k \in \mathbb{N}_+$, te $G^k, H^{k+2} \in \mathcal{S}$ totalne funkcije, tada se one mogu dobiti iz inicijalnih pomoću recimo c_G odnosno c_H primjena kompozicije, te r_G odnosno r_H primjena primitivne rekurzije. Tada se $G \curlyeqcirc H$ može dobiti iz inicijalnih funkcija pomoću najviše $c_G + c_H$ primjena kompozicije, i najviše $r_G + r_H + 1$ primjena primitivne rekurzije, dakle konačno mnogo, pa je $G \curlyeqcirc H \in \mathcal{S}$.

Prethodna tri odlomka pokazuju da skup \mathcal{S} sadrži sve inicijalne funkcije, te da je zatvoren na kompoziciju i na primitivnu rekurziju. Kako je skup primitivno rekurzivnih funkcija najmanji takav skup, slijedi da je podskup od \mathcal{S} , odnosno svaka primitivno rekurzivna funkcija je u \mathcal{S} .

Za drugi smjer, neka je $f \in \mathcal{S}$ proizvoljna funkcija. To znači da se može dobiti iz inicijalnih, recimo pomoću c primjena kompozicije i r primjena primitivne rekurzije. Dokažimo da je f primitivno rekurzivna, jakom indukcijom po $c + r$.

Ako je $c + r = 0$, tada mora biti $c = r = 0$, drugim riječima, f mora biti inicijalna. Kako skup primitivno rekurzivnih funkcija sadrži sve inicijalne funkcije, zaključujemo da je f primitivno rekurzivna.

Pretpostavimo da za sve $t < c + r$, za svaku funkciju $g \in \mathcal{S}$ dobivenu s c' primjena kompozicije i r' primjena primitivne rekurzije tako da je $c' + r' = t$, vrijedi da je g primitivno rekurzivna.

Neka je sad $f \in \mathcal{S}$ dobivena s c primjena kompozicije i r primjena primitivne rekurzije ($c + r > 0$), iz inicijalnih funkcija. Tada $f \in \mathcal{S}$ znači da je ili $f = H \circ (G_1, \dots, G_l)$ za neke $G_1, \dots, G_l, H \in \mathcal{S}$, ili pak $f = G \text{ } \mathbb{R} \text{ } H$, za neke totalne $G, H \in \mathcal{S}$ (odgovarajućih mjesnosti). U svakom od tih slučajeva pojedine funkcije pomoću kojih je dobivena f , dobivene su iz inicijalnih funkcija sa strogo manje ukupno primjena kompozicije i primitivne rekurzije: recimo, ako je $f = G \text{ } \mathbb{R} \text{ } H$, tada je $r > 0$, te je G (i isto tako H) dobivena iz inicijalnih pomoću najviše c primjena kompozicije i najviše $r - 1$ primjena primitivne rekurzije. Po pretpostavci indukcije, G i H su primitivno rekurzivne. Kako je skup primitivno rekurzivnih funkcija zatvoren na primitivnu rekurziju, zaključujemo da je $f = G \text{ } \mathbb{R} \text{ } H$ primitivno rekurzivna. Sasvim analogno bi se dokazalo da je f oblika $H \circ (G_1, \dots, G_l)$ primitivno rekurzivna. \square

Napomena 2.15. Naravno, umjesto „*ab ovo*” od inicijalnih funkcija, možemo krenuti od nekih funkcija za koje smo već utvrdili da su primitivno rekurzivne. Zapis koji pokazuje kako se neka funkcija f može dobiti kompozicijom i primitivnom rekurzijom iz već utvrđeno primitivno rekurzivnih funkcija zovemo *simboličkom definicijom* funkcije f . Recimo, u primjeru 2.7 navedena je simbolička definicija od C_2^3 , a u primjeru 2.11, simboličke definicije od add^2 i mul^2 . \triangleleft

Simboličke definicije su koncizne i izražajne, ali nisu baš čitljive. Umjesto njih, često se pišu *točkovne* definicije, gdje funkciju definiramo „po točkama” tako da kažemo što je $f(\vec{x})$, umjesto da kažemo što je f . Većina modernih programskih jezika upravo tako definira funkcije. Lijevi stupac (2.14)–(2.15) sadrži točkovnu definiciju zbrajanja dva broja, iz koje se vidi da je ono primitivno rekurzivno. U desnom stupcu je također točkovna definicija, ali manje čitljiva jer je namještena na oblik (2.10)–(2.11).

Točkovne definicije znaju biti neprecizne, i na nekim mjestima morat ćemo pribjeći simboličkoj definiciji da bi se znalo što zapravo pokušavamo definirati. No u ogromnom broju slučajeva, posebno za kompliciranije funkcije (i relacije), pisat ćemo točkovne definicije. Važno je imati na umu da se svaka takva točkovna definicija može *pretvoriti* u simboličku — pokušajte ako ne vjerujete!

Stroga formalizacija točkovnih definicija i njihovog pretvaranja u simboličke, zahtijevala bi alate iz logike prvog reda: funkcije kompozicijski definiramo kao terme, a relacije kao formule prvog reda s ograničenim kvantifikatorima. Koristimo prirodne brojeve kao konstantske simbole, te već dokazano primitivno rekurzivne funkcije i relacije kao funkcijske odnosno relacijske simbole. To ne trebamo raditi u potpunoj općenitosti, iz

vrlo sličnog razloga iz kojeg nismo napravili ni strogi dokaz teorema 1.26 — jer nam sveukupno do univerzalnosti treba samo konačno mnogo oblika takvih definicija, pa možemo svaki od njih zasebno pretvoriti u simbolički oblik ako treba.

Primjer 2.16. Točkovna definicija

$$f(x, y, z, 0) := g(x, h(x, y, z)) \quad (2.19)$$

$$f(x, y, z, t + 1) := h(z, f(x, y, z, t), g(t, z)) \quad (2.20)$$

ekvivalentna je simboličkoj definiciji

$$f^4 := g^2 \circ (l_1^3, h^3) \circ h^3 \circ (l_3^5, l_5^5, g^2 \circ (l_4^5, l_3^5)). \quad (2.21)$$

Kompozicijom s odgovarajućim koordinatnim projekcijama možemo postići i da primitivna rekurzija ne ide po zadnjem argumentu. \triangleleft

Definiciju 2.13 možemo iskoristiti za dokazivanje da sve primitivno rekurzivne funkcije imaju neko svojstvo \wp , baš kao što smo to učinili u dokazu propozicije 2.14. Definiramo \mathcal{S} kao skup svih brojevnih funkcija sa svojstvom \wp , dokažemo da sve inicijalne funkcije imaju svojstvo \wp , te da je skup \mathcal{S} zatvoren na kompoziciju i na primitivnu rekurziju. Po svojstvu najmanjeg skupa, skup primitivno rekurzivnih funkcija je podskup bilo kojeg skupa koji ima ta svojstva, pa tako i od \mathcal{S} . Evo dva primjera takvog zaključivanja.

Propozicija 2.17. *Sve primitivno rekurzivne funkcije su totalne.*

Dokaz. Iz napomene 2.2 slijedi da je skup svih totalnih brojevnih funkcija nadskup skupa svih inicijalnih funkcija. Iz propozicije 2.6 slijedi da je taj skup zatvoren na kompoziciju. Iz napomene 2.10 slijedi da je zatvoren i na primitivnu rekurziju. Dakle tvrdnja slijedi po svojstvu najmanjeg skupa. \square

Propozicija 2.18. *Sve primitivno rekurzivne funkcije su RAM-izračunljive.*

Dokaz. Po svojstvu najmanjeg skupa, koristeći korolar 2.4, te leme 2.8 i 2.12. \square

2.2.2. Primjeri primitivno rekurzivnih funkcija i relacija

Već smo vidjeli simboličke definicije od add^2 i mul^2 , što prema propoziciji 2.14 znači da su one primitivno rekurzivne. Sada ćemo vidjeti još brojne druge primjere. Prvo generalizirajmo primjer 2.7.

Propozicija 2.19. *Za sve $n \in \mathbb{N}$ i za sve $k \in \mathbb{N}_+$, konstantna funkcija C_n^k , zadana s $C_n^k(\vec{x}) := n$, primitivno je rekurzivna.*

Dokaz. Fiksirajmo $k \in \mathbb{N}_+$, i dokažimo tvrdnju „sve C_n^k su primitivno rekurzivne”, indukcijom po n . Baza: $C_0^k = Z \circ I_1^k$ je simbolička definicija od C_0^k . Doista,

$$(Z \circ I_1^k)(\vec{x}) = Z(I_1^k(\vec{x})) = Z(x_1) = 0 = C_0^k(\vec{x}). \quad (2.22)$$

To znači da je C_0^k dobivena iz dvije inicijalne funkcije kompozicijom, pa je primitivno rekurzivna (po propoziciji 2.14, na koju se više nećemo eksplicitno pozivati).

Pretpostavka: pretpostavimo da je C_m^k primitivno rekurzivna, za neki $m \in \mathbb{N}$. Korak: tvrdimo da je $C_{m+1}^k = Sc \circ C_m^k$ simbolička definicija od C_{m+1}^k . Doista,

$$(Sc \circ C_m^k)(\vec{x}) = Sc(C_m^k(\vec{x})) = Sc(m) = m + 1 = C_{m+1}^k(\vec{x}). \quad (2.23)$$

To znači da je C_{m+1}^k dobivena iz inicijalne i (po pretpostavci indukcije) primitivno rekurzivne funkcije kompozicijom, pa je primitivno rekurzivna. Po principu matematičke indukcije, tvrdnja vrijedi za svaki $n \in \mathbb{N}$. \square

Sada napokon možemo riješiti i problem jednomjesnih funkcija definiranih „degeneriranim” rekurzijama poput one u (2.12) i (2.13). Takve funkcije neće biti direktno dobivene primitivnom rekurzijom (već kompozicijom), ali će biti primitivno rekurzivne.

Propozicija 2.20. *Neka je $a \in \mathbb{N}$, te H^2 primitivno rekurzivna funkcija.*

Tada je funkcija F^1 , zadana s

$$F(0) := a, \quad (2.24)$$

$$F(x+1) := H(x, F(x)), \quad (2.25)$$

također primitivno rekurzivna.

Dokaz. Ideja je jednostavna: dodat ćemo još jedan „dummy” argument funkciji F , koji neće raditi ništa osim što će povećavati sve mjesnosti za 1. Precizno, funkcija F^2 (različita od tražene funkcije F^1 , jer je mjesnost dio identiteta funkcije) zadana je s

$$F(x, 0) := a = C_a^1(x), \quad (2.26)$$

$$F(x, y+1) := H(y, F(x, y)), \quad (2.27)$$

dakle dobivena je primitivnom rekurzijom iz funkcija $G^1 := C_a^1$ i $H^3 := H^2 \circ (I_2^3, I_3^3)$. Te funkcije su primitivno rekurzivne pa su totalne prema propoziciji 2.17, što znači da je primitivna rekurzija dobro definirana, i $F^2 := G^1 \bowtie H^3$ je primitivno rekurzivna. (Općenito ćemo u frazi poput „funkcija dobivena primitivnom rekurzijom iz primitivno rekurzivnih funkcija je ponovo primitivno rekurzivna” prešutno koristiti propoziciju 2.17, koja nam kaže da je primitivna rekurzija takvih funkcija uopće dobro definirana.)

Sada dokažimo da za sve $n \in \mathbb{N}$ vrijedi $F^2(0, n) = F^1(n)$, matematičkom indukcijom po n . Baza: $F^2(0, 0) = a$ po (2.26), a to je jednako $F^1(0)$ po (2.24). Pretpostavka: pretpostavimo da je $F^2(0, m) = F^1(m)$ za neki $m \in \mathbb{N}$. Korak: tada je redom prema (2.27), pretpostavci indukcije i (2.25),

$$F^2(0, m+1) = H(m, F^2(0, m)) = H(m, F^1(m)) = F^1(m+1), \quad (2.28)$$

pa tvrdnja vrijedi po principu matematičke indukcije. To znači da je $F^1 = F^2 \circ (Z, l_1^1)$ simbolička definicija od F^1 , iz čega slijedi da je F^1 primitivno rekurzivna. \square

Definicija 2.21. Za takve funkcije ubuduće pišemo „simboličku definiciju” $F^1 := a \text{ } \text{ } H^2$, no treba imati na umu da je to samo pokratak za $F^1 := (C_a^1 \text{ } \text{ } H^2 \circ (l_2^3, l_3^3)) \circ (Z, l_1^1)$.

Takvo zadavanje funkcije zvat ćemo *degeneriranom primitivnom rekurzijom*. \triangleleft

Pomoću propozicije 2.20, možemo za razne funkcije dokazati primitivnu rekurzivnost.

Primjer 2.22. Jednadžbe (2.12) i (2.13) kažu da je $\text{factorial}^1 = 1 \text{ } \text{ } \text{mul}^2 \circ (\text{Sc} \circ l_1^2, l_2^2)$ simbolička definicija funkcije faktorijel, pa je ona primitivno rekurzivna. \triangleleft

Primjer 2.23. Funkcija *prethodnik* zadana je s $\text{pd}(n) := \max\{x - 1, 0\}$. Iz toga slijedi $\text{pd}(\text{Sc}(x)) = x$ za sve $x \in \mathbb{N}$, dakle pd je lijevi inverz funkcije Sc . Naravno, Sc nema desni inverz jer nije surjekcija: ne poprima vrijednost 0, pa $\text{pd}(0) = 0$ moramo posebno definirati. Te dvije jednakosti:

$$\text{pd}(0) = 0, \quad (2.29)$$

$$\text{pd}(y + 1) = y, \quad (2.30)$$

kažu da je pd dobivena degeneriranom primitivnom rekurzijom $\text{pd} = 0 \text{ } \text{ } l_1^2$, pa je primitivno rekurzivna po propoziciji 2.20. \triangleleft

Napomena 2.24. Još jedan način definiranja funkcije pd , koji ćemo često koristiti kasnije, je sljedeći: htjeli bismo definirati $\text{pd}(x)$ kao $x - 1$, no problem je $x = 0$ (ako hoćemo da funkcija bude primitivno rekurzivna, dakle totalna). Često se u takvim funkcijama onda problematična vrijednost 0 zamijeni prvom sljedećom vrijednosti 1, koja nije problematična. Ako točkom ' označimo tu transformaciju (formalno, n' je n ako je pozitivan, a 1 ako je $n = 0$), tada možemo precizno definirati $\text{pd}(x) := x' - 1$. \triangleleft

Kad imamo prethodnik kao primitivno rekurzivnu funkciju, njenom iteracijom možemo definirati neku vrst oduzimanja. Ipak, zbog $\text{pd}(0) = 0$, vrijednosti tog oduzimanja bit će „odsječene odozdo” na nuli.

Primjer 2.25. Za $x, y \in \mathbb{N}$, označimo $x \ominus y := \max\{x - y, 0\}$. Lako se vidi da je

$$x \ominus 0 = x, \quad (2.31)$$

$$x \ominus (y + 1) = \text{pd}(x \ominus y), \quad (2.32)$$

što znači da je *ograničeno oduzimanje* dobiveno primitivnom rekurzijom iz funkcija l_1^1 i $\text{pd} \circ l_3^3$, pa je primitivno rekurzivna operacija. „Operacija” nam jednostavno znači dvomjesnu totalnu brojevnju funkciju, koja se piše infiksno između operanada. Kao funkciju, ograničeno oduzimanje zovemo *sub*. Dakle, $\text{sub}^2 = l_1^1 \text{ } \text{ } \text{pd} \circ l_3^3$. Još jedan način dolaska do te simboličke definicije je da jednostavno uzmemo simboličku definiciju za add^2 (primjer 2.11), i zamijenimo u njoj Sc sa pd . \triangleleft

Pored računskih operacija (dijeljenje s ostatkom definirat ćemo kasnije, kad uvedemo još neke tehnike), brojeve možemo i *uspoređivati*, raznim dvomjesnim relacijama kao što su $<$, \geq ili $=$. Rekli smo da izračunljivost relacija promatramo kroz izračunljivost njihovih karakterističnih funkcija.

Primjer 2.26. Za početak pogledajmo jednomjesnu relaciju \mathbb{N}_+ — drugim riječima, pozitivnost. Njena karakteristična funkcija je 0 u nuli, a 1 u svim ostalim prirodnim brojevima, pa se u literaturi još zove funkcija predznaka ili *signum*. Jednostavno ju je dobiti degeneriranom primitivnom rekurzijom $\chi_{\mathbb{N}_+} = 0 \text{ } \mathbb{R} C_1^2$, iz čega zaključujemo da je primitivno rekurzivna. Drugim riječima, pozitivnost je primitivno rekurzivno svojstvo, odnosno skup \mathbb{N}_+ je primitivno rekurzivan. \triangleleft

Kada imamo pozitivnost i ograničeno oduzimanje, možemo odmah vidjeti primitivnu rekurzivnost strogog uređaja. Ostale uređajne relacije dokazat ćemo primitivno rekurzivnima kasnije, kad ustanovimo kako se radi s logičkim veznicima.

Primjer 2.27. Lako se vidi da je $x > y$ ako i samo ako je $x \ominus y$ pozitivan, iz čega je $\chi_{>} = \chi_{\mathbb{N}_+} \circ \text{sub}$, primitivno rekurzivna funkcija. Naravno, $x < y$ ako i samo ako je $y > x$, dakle $\chi_{<} = \chi_{>} \circ (l_2^2, l_1^2)$ je također primitivno rekurzivna. \triangleleft

2.3. Minimizacija

Primitivno rekurzivne funkcije su korisne i pogodne za rad, ali njihova totalnost je na neki način i nedostatak. Recimo, u funkcijskoj paradigmi još uvijek nismo dobili izračunljivost prazne funkcije \emptyset^1 , koja je bila gotovo trivijalna u RAM-paradigmi. Da bismo dobili i parcijalne (ne-totalne) funkcije, moramo uvesti novi operator, pored \circ i \mathbb{R} . Taj operator — *minimizacija* — djelovat će na *relacijama*, odnosno njihovim karakterističnim funkcijama, i tražit će najmanji prirodni broj koji zadovoljava neko svojstvo. Intuitivno, odgovarat će *neograničenim* petljama — nešto poput while-petlji u Pythonu, samo s negiranim uvjetom — koje se ne izvršavaju određen broj puta, nego dok se ne ispuni neki uvjet.

Općenite neograničene petlje su raznolike i između provjeravanja uvjeta mogu na razne načine mijenjati stanje, ali kao i za primitivnu rekurziiju, aksiomatski ćemo pretpostaviti samo izračunljivost jednostavnih petlji oblika `for(y=0; !R(y); ++y);`. Kasnije ćemo vidjeti da u tom modelu možemo pisati i općenite beskonačne petlje.

Definicija 2.28. Neka je $k \in \mathbb{N}_+$ i \mathbb{R}^{k+1} relacija. Za funkciju F^k definiranu s

$$\mathcal{D}_F := \{\vec{x} \in \mathbb{N}^k \mid \exists y \mathbb{R}(\vec{x}, y)\} =: \exists_* \mathbb{R}, \quad (2.33)$$

$$F(\vec{x}) := \min \{y \in \mathbb{N} \mid \mathbb{R}(\vec{x}, y)\}, \text{ za sve } \vec{x} \in \exists_* \mathbb{R}, \quad (2.34)$$

kažemo da je dobivena *minimizacijom* relacije R . Pišemo $F^k := \mu R^{k+1}$, ili točkovno $F(\vec{x}) := \mu y R(\vec{x}, y)$. Smatramo da μ ima viši prioritet od \circ .

Za skup funkcija \mathcal{F} kažemo da je *zatvoren na minimizaciju* ako za svaki $k \in \mathbb{N}_+$, za svaku relaciju R^{k+1} , $\chi_R \in \mathcal{F}$ povlači $\mu R \in \mathcal{F}$. \triangleleft

Domena funkcije μR^{k+1} je k -mjesna relacija koju označavamo s $\exists_* R$ i zovemo je *projekcija* relacije R . Motivaciju za taj naziv lako možemo vidjeti ako zamislimo tromjesnu relaciju R^3 grafički prikazanu kao skup točaka u trodimenzionalnom prostoru. Tada je grafički prikaz od $\exists_* R^3$ upravo dvodimenzionalna projekcija (niz os z) tog skupa na x - y ravninu. Točka (x, y) će biti element od $\exists_* R$ ako i samo ako iznad nje postoji neka točka $(x, y, z) \in R$ (može ih biti i više).

U tom prikazu lako možemo vizualizirati i funkciju μR : za svaku točku (x, y) iz projekcije, $\mu z R(x, y, z)$ je visina do koje vidimo prazan prostor, odnosno visina na kojoj vidimo prvu točku u relaciji iznad (x, y) .

Naravno, samo za $\vec{x} \in \exists_* R$ izraz $\mu y R(\vec{x}, y)$ uopće ima smisla — inače je nedefiniran. Za obrat — da je za *svaki* $\vec{x} \in \exists_* R$, izraz $\mu y R(\vec{x}, y)$ dobro definiran — zaslužna je dobra uređenost od \mathbb{N} : ako postoji neki $y \in \mathbb{N}$ takav da vrijedi $R(\vec{x}, y)$, tada postoji i najmanji takav.

Primjer 2.29. Sada je lako dobiti praznu funkciju minimizacijom prazne relacije: konkretno, za svaki $k \in \mathbb{N}_+$, $\emptyset^k = \mu \emptyset^{k+1}$. Naime, projekcija prazne relacije je opet prazna, a jedina funkcija s praznom domenom je prazna funkcija. \triangleleft

Dodavanjem minimizacije, proširili smo skup izračunljivih funkcija u ovom modelu.

Definicija 2.30. Skup *parcijalno rekurzivnih* funkcija je najmanji skup funkcija koji sadrži sve inicijalne funkcije, te je zatvoren na kompoziciju, na primitivnu rekurziju, i na minimizaciju. Skup *rekurzivnih* funkcija je presjek skupa parcijalno rekurzivnih i skupa totalnih funkcija. Za relaciju R kažemo da je *rekurzivna* ako je njena karakteristična funkcija χ_R rekurzivna. \triangleleft

Izraz „parcijalno rekurzivna relacija” je beskoristan: svaka karakteristična funkcija je po definiciji totalna, pa ako je parcijalno rekurzivna, tada je zapravo rekurzivna.

Lema 2.31. *Skup rekurzivnih funkcija zatvoren je na kompoziciju.*

Dokaz. Neka su $k, l \in \mathbb{N}_+$, te G_1^k, \dots, G_l^k i H^l rekurzivne funkcije. Tada su one po definiciji totalne i parcijalno rekurzivne. Skup parcijalno rekurzivnih funkcija je zatvoren na kompoziciju, pa je $H \circ (G_1, \dots, G_l)$ parcijalno rekurzivna, no prema propoziciji 2.6, ona je i totalna — dakle rekurzivna funkcija. \square

Lema 2.32. *Skup rekurzivnih funkcija zatvoren je na primitivnu rekurziju.*

Dokaz. Neka je $k \in \mathbb{N}_+$, te G^k i H^{k+2} rekurzivne funkcije. Jer su G i H totalne, postoji $G \circ H$, i ona je totalna po napomeni 2.10. Također, kako su G i H parcijalno rekurzivne, a skup parcijalno rekurzivnih funkcija je po definiciji zatvoren na primitivnu rekurziju, $G \circ H$ je parcijalno rekurzivna. Iz toga i totalnosti slijedi: $G \circ H$ je rekurzivna funkcija. \square

Korolar 2.33. *Svaka primitivno rekurzivna funkcija je rekurzivna.*

Dokaz. Direktno po svojstvu najmanjeg skupa: sve inicijalne funkcije su totalne po napomeni 2.2, a parcijalno su rekurzivne po definiciji, dakle rekurzivne su. Sada samo treba primijeniti leme 2.31 i 2.32. \square

Korolar 2.34. *Neka je $a \in \mathbb{N}$, te H^2 rekurzivna funkcija. Tada je funkcija $a \circ H$ također rekurzivna.*

Dokaz. Iz dokaza propozicije 2.20 vidimo $a \circ H^2 = (C_a^1 \circ H^2 \circ (I_2^3, I_3^3)) \circ (Z, I_1^1)$. Sada tvrdnja slijedi iz propozicije 2.19, korolara 2.33, te lema 2.31 i 2.32. \square

Slično kao propoziciju 2.14, mogli bismo dokazati da je funkcija parcijalno rekurzivna ako i samo ako je dobivena iz inicijalnih pomoću konačno mnogo primjena kompozicije, primitivne rekurzije (samo na totalne dobivene funkcije) i minimizacije (na relacije čije su karakteristične funkcije već dobivene). To bi bila neka generalizacija simboličke definicije za parcijalno rekurzivne funkcije. Ipak, najčešće ćemo pisati točkovne definicije parcijalno rekurzivnih funkcija — a i Kleenejev teorem o normalnoj formi, koji ćemo dokazati kasnije, reći će da nam tolika općenitost nije potrebna: dovoljno je promatrati funkcije oblika $U \circ \mu T$, gdje su U i T primitivno rekurzivne.

2.3.1. Kompajler za funkcijski jezik

Pogledajmo sada kako možemo proširiti rezultat iz propozicije 2.18 na parcijalno rekurzivne funkcije. Vidimo da je jedino što nam još nedostaje, opis kako se neograničene petlje izvršavaju na RAM-stroju.

Naravno, ideja nije ništa revolucionarno: kao „kontrolnu varijablu” koristimo upravo \mathcal{R}_0 , iz istog razloga kao kod pisanja makro-programa za primitivnu rekurziju — da eventualnim završetkom petlje bude već spremna kao izlazni podatak. U tom smislu, inicijalizirana na nulu već jest na početku izračunavanja, samo je treba inkrementirati poslije svake provjere uvjeta R .

Ili prije? Ovdje imamo mali tehnički problem: htjeli bismo da provjera bude na samom dnu programa, jer tako najbolje odgovara semantici negiranog uvjeta. Terminologijom jezika Pascal, implementiramo until-petlju, ne while-petlju. Iz sličnog razloga kao naši strojevi, Pascal je while-uvjet provjeravao na vrhu, a until-uvjet na dnu petlje. Semantika samog skoka je ista: ako je uvjet istinit, nastavljamo dalje

(ulazimo u while-petlju, ili izlazimo iz until-petlje), a ako je lažan, skačemo na drugi kraj petlje (izlazimo iz while-petlje, ili ponovo izvršavamo until-petlju) — što je baš semantika instrukcije tipa DEC.

Ali until-petlje imaju jedan veliki nedostatak: uvijek se izvrše bar jednom (kao do...while-petlja u jeziku C). Kako se u petlji mora nalaziti instrukcija INC \mathcal{R}_0 , čini se da nikako ne možemo postići da \mathcal{R}_0 na kraju bude 0, što je svakako problem: recimo za univerzalnu relaciju, $\mu y \mathbb{N}^{k+1}(\vec{x}, y) = 0$.

Rješenje problema se nalazi na drugom kraju, odnosno početku programa: program možemo početi izvršavati iz sredine! Odnosno preciznije, na sam početak programa možemo staviti instrukciju (0.GO TO 2), da preskočimo instrukciju (1.INC \mathcal{R}_0) kod prvog prolaza.

Lema 2.35. *Skup Comp je zatvoren na minimizaciju.*

Dokaz. Neka je $k \in \mathbb{N}_+$, te \mathbb{R}^{k+1} RAM-izračunljiva. To znači da postoji RAM-program P_R koji računa karakterističnu funkciju χ_R^{k+1} . Tvrdimo da makro-program

$$Q_F := \left[\begin{array}{l} 0. \text{ GO TO } 2 \\ 1. \text{ INC } \mathcal{R}_0 \\ 2. P_R(\mathcal{R}_1, \mathcal{R}_2, \dots, \mathcal{R}_k, \mathcal{R}_0) \rightarrow \mathcal{R}_{k+1} \text{ USING } \mathcal{R}_{k+2}.. \\ 3. \text{ DEC } \mathcal{R}_{k+1}, 1 \end{array} \right] \quad (2.35)$$

računa funkciju $F^k := \mu R$. U tu svrhu, po definiciji 1.10, neka je $\vec{x} \in \mathbb{N}^k$ proizvoljan. Ako $\vec{x} \notin \exists_* R$, dakle ne postoji $y \in \mathbb{N}$ takav da vrijedi $R(\vec{x}, y)$, to zapravo znači da za svaki $y \in \mathbb{N}$ vrijedi $\chi_R(\vec{x}, y) = 0$. Ta će vrijednost upravo biti izračunata kad god makro-stroj izvrši instrukciju rednog broja 2 (primijetimo da s USING $\mathcal{R}_{k+2}..$ čuvamo originalne ulazne registre), redom za sve prirodne brojeve y , i završit će u \mathcal{R}_{k+1} . Nakon toga, instrukcija s rednim brojem 3 će pokušati dekrementirati \mathcal{R}_{k+1} , ustanoviti da je njegov sadržaj 0, i postaviti PC na 1. Tada će se inkrementirati \mathcal{R}_0 , i ponovo testirati uvjet ... i tako dalje. Preciznije, imat ćemo sljedeće Q_F -izračunavanje s \vec{x} :

$$\begin{aligned} (0, \vec{x}, 0, \dots, 0, 0) &\rightsquigarrow (0, \vec{x}, 0, 0, \dots, 2, 0) \rightsquigarrow^* (0, \vec{x}, 0, ?, \dots, 3, 0) \rightsquigarrow (0, \vec{x}, 0, ?, \dots, 1, 0) \\ &\rightsquigarrow (1, \vec{x}, 0, ?, \dots, 2, 0) \rightsquigarrow^* (1, \vec{x}, 0, ?, \dots, 3, 0) \rightsquigarrow (1, \vec{x}, 0, ?, \dots, 1, 0) \rightsquigarrow \\ &\rightsquigarrow (2, \vec{x}, 0, ?, \dots, 2, 0) \rightsquigarrow^* (2, \vec{x}, 0, ?, \dots, 3, 0) \rightsquigarrow (2, \vec{x}, 0, ?, \dots, 1, 0) \rightsquigarrow \dots \end{aligned} \quad (2.36)$$

koje nikada ne stane jer nijedna konfiguracija u njemu nije završna.

Ako je pak $\vec{x} \in \exists_* R$, tada je $F(\vec{x}) \in \mathbb{N}$ definirano — označimo tu vrijednost sa y_0 . Tada za svaki $y < y_0$ i dalje vrijedi $\chi_R(\vec{x}, y) = 0$, pa će Q_F -izračunavanje s \vec{x} izgledati isto sve do trenutka kad se sadržaj registra \mathcal{R}_0 približi odozdo y_0 , kada će izgledati ovako:

$$\begin{aligned} \dots \rightsquigarrow (y_0 - 1, \vec{x}, 0, ?, \dots, 1, 0) &\rightsquigarrow (y_0, \vec{x}, 0, ?, \dots, 2, 0) \rightsquigarrow^* (y_0, \vec{x}, 1, ?, \dots, 3, 0) \rightsquigarrow \\ &\rightsquigarrow (y_0, \vec{x}, 0, ?, \dots, 4, 0) \rightsquigarrow (y_0, \vec{x}, 0, ?, \dots, 4, 0) \rightsquigarrow \dots, \end{aligned} \quad (2.37)$$

odnosno doći će do završne konfiguracije oblika (y_0, \dots) , pa će izlazni podatak biti $y_0 = F(\vec{x})$, kao što i treba. Naravno, ako je $f(\vec{x}) = 0$, tada približavanje odozdo ne postoji, već izračunavanje izgleda ovako:

$$(0, \vec{x}, 0, 0, \dots, 0, 0) \rightsquigarrow (0, \vec{x}, 0, 0, \dots, 2, 0) \rightsquigarrow^* (0, \vec{x}, 1, ?, \dots, 3, 0) \rightsquigarrow \\ \rightsquigarrow (0, \vec{x}, 0, ?, \dots, 4, 0) \rightsquigarrow (0, \vec{x}, 0, ?, \dots, 4, 0) \rightsquigarrow \dots, \quad (2.38)$$

što znači da u konačno mnogo koraka dođe do završne konfiguracije oblika $(0, \dots)$, pa je izlazni podatak 0, kao što i treba biti.

Sada prema teoremu 1.26, RAM-algoritam $(Q_F^b)^k$ računa F , pa je $F \in \text{Comp}$. \square

Napokon možemo dokazati da se svi funkcijski programi mogu simulirati odgovarajućim imperativnim programima u našoj formalizaciji.

Teorem 2.36. *Svaka parcijalno rekurzivna funkcija je RAM-izračunljiva.*

Dokaz. Po svojstvu najmanjeg skupa za parcijalno rekurzivne funkcije. Skup Comp sadrži sve inicijalne funkcije prema korolaru 2.4, te je zatvoren na kompoziciju, primitivnu rekurziju i minimizaciju prema lemapa 2.8, 2.12 i 2.35 — dakle nadskup je najmanjeg takvog skupa, koji je upravo skup svih parcijalno rekurzivnih funkcija. \square

Upravo napisani dokaz je zapravo sasvim konstruktivan: ako imamo zadanu parcijalno rekurzivnu funkciju ili relaciju, tada možemo napisati njenu simboličku definiciju u obliku stabla. Listovi tog stabla su inicijalne funkcije, a čvorovi mu predstavljaju funkcije dobivene kompozicijom, primitivnom rekurzijom ili minimizacijom iz čvorova ispod, odnosno relacije čije su karakteristične funkcije dobivene na isti način. U korijenu stabla je funkcija ili relacija koju tražimo.

Sada *postorder* obilaskom tog stabla možemo konstruirati preslikavanje (apstraktni tip podataka Mapping) *compile* koje svaku od tih funkcija odnosno relacija preslikava u RAM-program koji je računa. Kad obilazimo list L , koristimo dokaz propozicije 2.3 da bismo pogledali koji makro-program Q_L računa L , te definiramo $\text{compile}[L] := Q_L^b$. Ako se nalazimo na unutarnjem čvoru N , ovisno o tome kako je dobiven iz svoje djece koristimo dokaz leme 2.8, 2.12 ili 2.35 da utvrdimo koji oblik makro-programa — (2.7), (2.16) ili (2.35) — računa N , te u taj predložak (*template*) uvrstimo funkcijske makroe koji na odgovarajućem mjestu pozivaju $\text{compile}[D_i]$, gdje su D_i djeca čvora N . (Zbog *postorder* obilaska, D_i su već kompajlirani.) Na taj način dobijemo makro-program Q_N , te definiramo $\text{compile}[N] := Q_N^b$. Na kraju *postorder* obilaska nalazi se korijen K , i $\text{compile}[K]$ će biti RAM-program koji računa parcijalno rekurzivnu funkciju (ili relaciju) zadanu simboličkom definicijom.

Za obrat teorema 2.36 morat ćemo se više namučiti. To je tema poglavlja 3.

2.4. Tehnike za rad s (primitivno) rekurzivnim funkcijama

Odluka da ćemo koristiti marljivu evaluaciju omogućila nam je relativno jednostavan dokaz RAM-izračunljivosti tako definiranih funkcija — jer kompozicija uz marljivu evaluaciju praktički direktno odgovara slijedom izvršavanju instrukcija u programu — ali će dosta otežati dokaz u suprotnom smjeru, jer neke programske tehnike koje su jako jednostavne u RAM-modelu izračunavanja zasad ne znamo ostvariti u funkcijskom modelu s marljivom evaluacijom.

Jedna od najjednostavnijih vjerojatno je *grananje*: imamo dva RAM-programa P_{true} i P_{false} . Želimo izvršiti jedan od ta dva programa ovisno o tome je li r_j pozitivan ili nije, ali tako da ako je npr. P_{true} prazan program, P_{false} beskonačna petlja, a $r_j = 1$, čitavo izračunavanje stane. Nije veliki problem napisati makro

$$(IF \mathcal{R}_j \text{ THEN } P_{true}^* \text{ ELSE } P_{false}^*) := \left[\begin{array}{l} 0. \text{ DEC } \mathcal{R}_j, 4 \\ 1. \text{ INC } \mathcal{R}_j \\ 2. P_{true}^* \\ 3. \text{ GO TO } 5 \\ 4. P_{false}^* \end{array} \right]^{b*}, \quad (2.39)$$

ali ne možemo napisati odgovarajuću funkciju if^3 takvu da $f(\vec{x}, y) \simeq if(y, g(\vec{x}), h(\vec{x}))$ ima analognu semantiku: ako je g totalna a h prazna, bez obzira na y marljiva evaluacija ima za posljedicu da je f također prazna. Za to ćemo trebati razviti druge tehnike, no za početak pokažimo kako su beskonačne petlje (odnosno netotalne funkcije) *jedini* problem — jer nas zanima samo postojanje algoritma a ne i performanse, s totalnim izračunljivim funkcijama nešto poput if možemo napraviti relativno lako.

Ipak, prethodno nam trebaju neki pripremni rezultati. Za početak, za implementaciju $ELSE$ trebamo moći negirati uvjete i ostati u istoj „klasi” izračunljivosti.

Propozicija 2.37. *Neka je $k \in \mathbb{N}_+$, te R^k (primitivno) rekurzivna relacija. Tada je i relacija $(R^c)^k$, zadana s $R^c(\vec{x}) \iff \neg R(\vec{x})$, također (primitivno) rekurzivna.*

Primijetimo nekoliko važnih detalja u upravo iskazanoj propoziciji. Prvo, kako smo već rekli u uvodu, na k -mjesne relacije gledamo simultano kao na formule s k slobodnih varijabli, i na skupove k -torki. Zato koristimo oznaku za skupovni komplement, dok u definiciji formulom koristimo logičku negaciju. U sljedećoj propoziciji to ćemo generalizirati na dvomjesne logičke veznike odnosno skupovne operacije.

Drugo, mjesnost R^c smatramo istom kao mjesnost od R . Čak i u slučaju prazne relacije, $(\emptyset^k)^c = \mathbb{N}^k$ („univerzalni skup”), u skladu s našom odlukom da prazne relacije različitih mjesnosti gledamo kao različite relacije (komplementi su im različiti).

I treće, u iskazu propozicije pojavljuje se riječ „primitivno” u zagradama. Taj način izražavanja koristit ćemo još mnogo puta u sljedećim propozicijama. To jednostavno

znači da zapravo iskazujemo dvije propozicije: jedna kaže da je komplement primitivno rekurzivne relacije ponovo primitivno rekurzivna relacija, a druga kaže da je komplement rekurzivne relacije ponovo rekurzivna relacija. Dakle, u jednoj verziji čitamo riječ „primitivno” na svim mjestima gdje se pojavljuje u zagradama, a u drugoj je ne čitamo ni na jednom takvom mjestu.

Dokaz. Zapravo, ideja dokaza je prilično jednostavna: samo treba nekako prikazati karakterističnu funkciju χ_{R^c} pomoću χ_R . Dakle, treba nam funkcija koja 0 preslikava u 1, a 1 u 0. Naravno, $x \mapsto 1 - x$ je jedna takva, ali nama treba brojevena funkcija s \mathbb{N} u \mathbb{N} . Zato ćemo uzeti $f_0(x) := 1 \ominus x$, koja jednako djeluje na skupu $\{0, 1\}$, a sve vrijednosti su joj prirodni brojevi. Ta funkcija je primitivno rekurzivna po propoziciji 2.14: $f_0 := \text{sub} \circ (C_1^1, I_1^1)$ njena je simbolička definicija, sub je primitivno rekurzivna po primjeru 2.25, C_1^1 po propoziciji 2.19, a I_1^1 je inicijalna.

Sada točkovna jednakost $\chi_{R^c}(\vec{x}) = 1 \ominus \chi_R(\vec{x})$ simbolički glasi $\chi_{R^c} = f_0 \circ \chi_R$. Ako je χ_R primitivno rekurzivna, tada je i χ_{R^c} primitivno rekurzivna, jer je skup primitivno rekurzivnih funkcija zatvoren na kompoziciju. Ako pak samo znamo da je χ_R rekurzivna, tada zaključujemo ovako: f_0 je rekurzivna prema korolaru 2.33. Prema lemi 2.31, tada je i χ_{R^c} rekurzivna kao kompozicija dvije rekurzivne funkcije. \square

Ubuduće ćemo samo napisati simboličku definiciju tražene funkcije iz zadanih funkcija, koristeći neke pomoćne primitivno rekurzivne funkcije, te kompoziciju i eventualno primitivnu rekurziju. Podrazumijevat ćemo da na kraju dokaza uvijek imamo argumentaciju poput ove u prethodnom odlomku, tako da ako su zadane funkcije primitivno rekurzivne, tada je i tražena funkcija primitivno rekurzivna, te ako su zadane funkcije rekurzivne, tada je i tražena funkcija rekurzivna. Efektivno, imamo generalizaciju napomene 2.15, gdje polazimo od rekurzivnih funkcija umjesto od primitivno rekurzivnih.

Korolar 2.38. Brojevene relacije nestrogog uređaja \leq i \geq su primitivno rekurzivne.

Dokaz. Direktno iz primjera 2.27, propozicije 2.37, te očitih ekvivalencija

$$x \leq y \iff \neg(x > y) \quad (\leq) = (>)^c, \quad (2.40)$$

$$x \geq y \iff \neg(x < y) \quad (\geq) = (<)^c \quad (2.41)$$

(točkovno u lijevom stupcu, simbolički u desnom). \square

Propozicija 2.39. Neka je $k \in \mathbb{N}_+$, te R^k i P^k (primitivno) rekurzivne relacije iste mjesnosti. Tada su (primitivno) rekurzivne i relacije zadane logički/skupovno s

$$Q_1(\vec{x}) := R(\vec{x}) \wedge P(\vec{x}) \quad Q_1 := R \cap P, \quad (2.42)$$

$$Q_2(\vec{x}) := R(\vec{x}) \vee P(\vec{x}) \quad Q_2 := R \cup P, \quad (2.43)$$

$$Q_3(\vec{x}) := R(\vec{x}) \rightarrow P(\vec{x}) \quad Q_3 := (R \setminus P)^c, \quad (2.44)$$

$$Q_4(\vec{x}) := R(\vec{x}) \leftrightarrow P(\vec{x}) \quad Q_4 := (R \triangle P)^c. \quad (2.45)$$

Dokaz. Prvo pokažimo da skupovne i logičke definicije ekvivalentne za upravo definirane relacije. Za Q_1 i Q_2 to je upravo definicija presjeka i unije. Za Q_3 imamo

$$\vec{x} \in (R \setminus P)^c \Leftrightarrow \neg(\vec{x} \in R \wedge \vec{x} \notin P) \Leftrightarrow \vec{x} \notin R \vee \vec{x} \in P \Leftrightarrow \vec{x} \in R \rightarrow \vec{x} \in P. \quad (2.46)$$

Za Q_4 , logička definicija kaže da je $\chi_R(\vec{x}) = \chi_P(\vec{x})$. Njena negacija kaže da su ta dva broja različiti, a kako su oba iz skupa $\{0, 1\}$, mora jedan od njih biti 0 a drugi 1. To upravo znači da se \vec{x} nalazi u točno jednom od skupova R i P , dakle $\vec{x} \in R \triangle P$.

Za primitivnu rekurzivnost tih relacija, kao u dokazu propozicije 2.37, trebamo naći dvomjesne primitivno rekurzivne funkcije f_i koje će preslikavati $\chi_R(\vec{x})$ i $\chi_P(\vec{x})$ u $\chi_{Q_i}(\vec{x})$, dakle koje će na skupu $\{0, 1\}$ djelovati onako kako propisuju tablice istinitosti za pojedine logičke veznike.

Za konjunkciju odnosno presjek, to je upravo $f_1 = \text{mul}^2$ — u starijoj literaturi konjunkcija se još zna nazivati „logičko množenje”, a u programskom jeziku Pascal isti simbol $*$ služio je za množenje brojeva i za presjek skupova.

Svi se ostali logički veznici mogu ekvivalentno zapisati pomoću negacije i konjunkcije na dobro poznat način:

$$\varphi \vee \psi \Leftrightarrow \neg(\neg\varphi \wedge \neg\psi) \quad f_2(x, y) := f_0(f_0(x) \cdot f_0(y)), \quad (2.47)$$

$$\varphi \rightarrow \psi \Leftrightarrow \neg\varphi \vee \psi \quad f_3(x, y) := f_2(f_0(x), y), \quad (2.48)$$

$$\varphi \leftrightarrow \psi \Leftrightarrow (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi) \quad f_4(x, y) := f_3(x, y) \cdot f_3(y, x), \quad (2.49)$$

iz čega odmah čitamo točkovne definicije funkcija f_2 , f_3 i f_4 (desni stupac).

Sada primjenom propozicije 2.37 na Q_3 i Q_4 dobijemo da je skupovna razlika, kao i simetrična skupovna razlika, (primitivno) rekurzivnih relacija iste mjesnosti ponovo (primitivno) rekurzivna. \square

Korolar 2.40. *Jednakost (dvomjesna brojevena relacija) je primitivno rekurzivna.*

Dokaz. Direktno iz korolara 2.38, propozicije 2.39 i očite ekvivalencije („Cantor–Bernstein za konačne skupove”) $x = y \Leftrightarrow x \leq y \wedge x \geq y$, iz koje po prethodnom imamo $(=) = (\leq) \cap (\geq)$, odnosno $\chi_{=} = \chi_{\leq} \cdot \chi_{\geq} = \text{mul}^2 \circ (\chi_{\leq}^2, \chi_{\geq}^2)$. \square

2.4.1. Višestruke operacije

Sada ćemo nešto reći o višestrukim zbrojevima, umnošcima, unijama i presjecima. Primijetimo da je u većini programskih jezika npr. zbrajanje implementirano kao infiksni operator, najčešće lijevo asociiran, a jedina operacija koja je doista implementirana u procesoru je dvomjesno zbrajanje — tako da se npr. $a + b + c + d$ shvaća kao $((a + b) + c) + d$, odnosno kompajlira se kao slijed tri instrukcije zbrajanja. Mi ćemo učiniti isto, samo ćemo u skladu s funkcijskom paradigmom slijed implementirati kao kompoziciju.

Lema 2.41. Za svaki $k \in \mathbb{N}_+$, funkcije add^k i mul^k , zadane s

$$\text{add}(x_1, x_2, \dots, x_k) := x_1 + x_2 + \dots + x_k, \quad (2.50)$$

$$\text{mul}(x_1, x_2, \dots, x_k) := x_1 \cdot x_2 \cdots x_k, \quad (2.51)$$

primitivno su rekurzivne.

Dokaz. Matematičkom indukcijom po k . Za $k = 1$, vidimo da je $\text{add}^1 = \text{mul}^1 = I_1^1$, inicijalna funkcija. Za $k = 2$, tvrdnja slijedi iz primjera 2.11. Pretpostavimo sad da su za neki $l \in \mathbb{N} \setminus \{0, 1\}$, funkcije add^l i mul^l primitivno rekurzivne. Tada definiciju

$$x_1 + x_2 + \dots + x_l + x_{l+1} := (x_1 + x_2 + \dots + x_l) + x_{l+1} \quad (2.52)$$

možemo zapisati kao

$$\text{add}^{l+1}(x_1, x_2, \dots, x_l, x_{l+1}) = \text{add}^2(\text{add}^l(x_1, x_2, \dots, x_l), x_{l+1}) \quad (2.53)$$

ili simbolički

$$\text{add}^{l+1} = \text{add}^2 \circ (\text{add}^l \circ (I_1^{l+1}, I_2^{l+1}, \dots, I_l^{l+1}), I_{l+1}^{l+1}), \quad (2.54)$$

pa tvrdnja slijedi iz pretpostavke indukcije i primjera 2.11. Potpuno analogno, zamjenom add s mul , slijedi i druga tvrdnja za $l + 1$, odnosno po principu matematičke indukcije za svaki $k \in \mathbb{N}_+$. \square

Propozicija 2.42. Neka su $k, l \in \mathbb{N}_+$, te $R_1^k, R_2^k, \dots, R_l^k$ (primitivno) rekurzivne relacije iste mjesnosti. Tada su $\bigcap_{i=1}^l R_i$ i $\bigcup_{i=1}^l R_i$ također (primitivno) rekurzivne.

Dokaz. Za presjek, koristimo istu tehniku kao u dokazu propozicije 2.39 za Q_1 , samo umjesto funkcije $f_1 = \text{mul}^2$ koristimo mul^l . Konkretno, tvrdimo da je

$$\chi_{\bigcap_{i=1}^l R_i} = \chi_{R_1} \cdot \chi_{R_2} \cdots \chi_{R_l} = \text{mul}^l \circ (\chi_{R_1}, \chi_{R_2}, \dots, \chi_{R_l}) \quad (2.55)$$

— doista, ako je \vec{x} u presjeku, imamo jednakost $1 = 1 \cdot 1 \cdots 1$, a ako nije, tada po De Morganovom pravilu nije u nekoj R_i , pa na lijevoj strani stoji 0, a na desnoj je umnožak u kojem je bar jedan faktor jednak 0, te jednakost vrijedi.

Za uniju, možemo opet iskoristiti De Morganovo pravilo i propoziciju 2.37, ali možemo i upotrijebiti drugačiju tehniku, koja će nam biti korisna kasnije. Naime, u našem skupu nema negativnih brojeva, te je zbroj 0 jedino ako su svi pribrojnici 0 — odnosno, zbroj je pozitivan ako i samo ako je neki pribrojnik pozitivan. To znači da ako u širem smislu shvatimo sve pozitivne prirodne brojeve (a ne samo broj 1) kao istinite, l -struka unija jednostavno odgovara funkciji add^l — odnosno preciznije, $\chi_{\mathbb{N}_+} \circ \text{add}^l$, jer želimo sve pozitivne brojeve preslikati natrag u 1. Dakle,

$$\chi_{\bigcup_{i=1}^l R_i} = \chi_{\mathbb{N}_+} \circ \text{add}^l \circ (\chi_{R_1}, \chi_{R_2}, \dots, \chi_{R_l}), \quad (2.56)$$

što je (primitivno) rekurzivno ako su sve χ_{R_i} (primitivno) rekurzivne. \square

2.4.2. Teorem o grananju za totalne funkcije

Napokon možemo dokazati teorem o grananju za totalne funkcije, samo prethodno moramo precizno definirati pojmove.

Definicija 2.43. Neka su $k, l \in \mathbb{N}_+$, neka su $G_0^k, G_1^k, G_2^k, \dots, G_l^k$ funkcije, te neka su $R_1^k, R_2^k, \dots, R_l^k$ u parovima disjunktne relacije iste mjesnosti. Za funkciju F^k definiranu s

$$\mathcal{D}_F := \bigcup_{i=0}^l (\mathcal{D}_{G_i} \cap R_i), \quad \text{gdje je } R_0 := \left(\bigcup_{i=1}^l R_i \right)^c, \quad (2.57)$$

$$F(\vec{x}) := \begin{cases} G_1(\vec{x}), & R_1(\vec{x}) \\ G_2(\vec{x}), & R_2(\vec{x}) \\ \vdots & \vdots \\ G_l(\vec{x}), & R_l(\vec{x}) \\ G_0(\vec{x}), & \text{inače} \end{cases} \quad \text{za sve } \vec{x} \in \mathcal{D}_F, \quad (2.58)$$

kažemo da je dobivena *grananjem* iz grana $G_0, G_1, G_2, \dots, G_l$ i uvjeta R_1, R_2, \dots, R_l . Simbolički pišemo $F := \{R_1 : G_1, R_2 : G_2, \dots, R_l : G_l, G_0\}$. Ako ne navedemo G_0 , smatramo da je $G_0 := \emptyset^k$ (odnosno F nije definirana izvan unije svih uvjeta). \triangleleft

Primijetite da zahtijevamo da su svi uvjeti u parovima disjunktne, odnosno da je za svaki $\vec{x} \in \mathbb{N}^k$, najviše jedan od njih ispunjen. Tako ne moramo brinuti o redoslijedu provjeravanja uvjeta. Naravno, ako imamo fiksiran redoslijed ne nužno disjunktne uvjeta R_1, R_2, \dots, R_l , uvijek možemo napraviti nove disjunktne uvjete s istom unijom:

$$P_1 := R_1, \quad (2.59)$$

$$P_i := R_i \setminus \bigcup_{j=1}^{i-1} R_j, \quad \text{za sve } i \in [2..l], \quad (2.60)$$

koji će biti (primitivno) rekurzivni ako su R_i takvi, po propozicijama 2.39 i 2.42. U tom smislu, podrazumijevajući da u samom provjeravanju uvjeta nema beskonačnih petlji (karakteristične funkcije su totalne), grananje odgovara uobičajenom grananju poput `if/elif/else`, ili `switch/case/default` (u jeziku Python odnosno C).

Kao što smo već napomenuli, još ne znamo dokazati da je to izračunljivo ako grane G_i nisu totalne. To ćemo kasnije — zasad dokažimo da je funkcija dobivena grananjem iz *totalnih* izračunljivih funkcija i uvjeta, izračunljiva u funkcijskom modelu.

Teorem 2.44 (Teorem o grananju, (primitivno) rekurzivna verzija). *Neka su $k, l \in \mathbb{N}_+$, neka su $G_0^k, G_1^k, G_2^k, \dots, G_l^k$ (primitivno) rekurzivne funkcije, te $R_1^k, R_2^k, \dots, R_l^k$ u parovima disjunktne (primitivno) rekurzivne relacije, sve iste mjesnosti. Tada je i funkcija $F := \{R_1 : G_1, R_2 : G_2, \dots, R_l : G_l, G_0\}$ također (primitivno) rekurzivna.*

Primijetimo: ovdje ne smijemo ispustiti G_0 , jer \emptyset^k nije (primitivno) rekurzivna!

Dokaz. Prvo primijetimo da je $R_0 := (\bigcup_{i=1}^l R_i)^c$ (primitivno) rekurzivna, prema propozicijama 2.42 i 2.37. Dalje je stvar jednostavne aritmetike, jer tvrdimo da je

$$F = \chi_{R_0} \cdot G_0 + \chi_{R_1} \cdot G_1 + \chi_{R_2} \cdot G_2 + \cdots + \chi_{R_l} \cdot G_l, \quad (2.61)$$

dakle dobivena kompozicijom iz (primitivno) rekurzivnih χ_{R_i} , G_i , te mul^2 i add^{l+1} .

Doista, neka je $\vec{x} \in \mathbb{N}^k$ proizvoljan, i promotrimo funkcijsku jednakost 2.61 u \vec{x} . Ako je \vec{x} u nekoj R_i , tada je $\chi_{R_i}(\vec{x}) = 1$, pa je i -ti pribrojnik u (2.61) (brojeći od nule) jednak $G_i(\vec{x})$. Štoviše, jer su uvjeti u parovima disjunktni, $\vec{x} \notin R_j$ za sve $j \in [1..l] \setminus \{i\}$, dok iz definicije R_0 slijedi također $\vec{x} \notin R_0$, dakle svi ostali pribrojnici imaju faktor 0, pa iznose 0 i ne utječu na zbroj. Dakle ako je $\vec{x} \in R_i$, tada je $F(\vec{x}) = G_i(\vec{x})$.

S druge strane, ako \vec{x} nije ni u jednoj R_i , tada po De Morganovom pravilu nije ni u njihovoj uniji, dakle $\vec{x} \in R_0$, pa je početni pribrojnik u (2.61) jednak $G_0(\vec{x})$, a svi ostali pribrojnici, kao i u prethodnom odlomku, jednaki su 0. Dakle tada je $F(\vec{x}) = G_0(\vec{x})$, kao što i treba biti. \square

Domena funkcije dobivene grananjem zapravo je još kompliciranija od domene funkcije dobivene kompozicijom. I ovdje zato koristimo definiciju u stilu napomene 1.3,

$$F(\vec{x}) := \begin{cases} G_1(\vec{x}), & R_1(\vec{x}) \\ \vdots & \\ G_0(\vec{x}), & \text{inače,} \end{cases} \quad (2.62)$$

uz dogovor da izraz na desnoj strani ima smisla samo za one \vec{x} za koje izraz u i -tom retku ima smisla, ako uvjet u tom retku vrijedi, a za one \vec{x} za koje izraz u posljednjem retku ima smisla, ako nijedan od prethodnih uvjeta ne vrijedi. Dakle, ne zahtijevamo (kao prije) da *svaki* podizraz izraza na desnoj strani ima smisla, već samo oni koji se nalaze u „relevantnom retku” definicije.

No kao što smo već rekli, komplikacije s domenom bit će nam bitne kasnije. Zasad radimo s (primitivno) rekurzivnim funkcijama i relacijama, koje su totalne — a pod tim uvjetima i funkcija dobivena grananjem je totalna, štoviše također (primitivno) rekurzivna, dok god navedemo i funkciju G_0 za „podrazumijevani slučaj” (*default*).

Kao primjenu teorema 2.44, dokazat ćemo da *konačnom promjenom* („editiranjem”) vrijednosti ne možemo pokvariti izračunljivost funkcije.

Lema 2.45. *Svaka jednočlana brojeva relacija je primitivno rekurzivna.*

Dokaz. Neka je R jednočlana, i označimo s k njenu mjesnost, a s $\vec{c} = (c_1, \dots, c_k)$ jedini njen element. Tada po definiciji jednakosti k -torki vrijedi

$$R(\vec{x}) \iff \vec{x} \in \{\vec{c}\} \iff \vec{x} = \vec{c} \iff x_1 = c_1 \wedge \cdots \wedge x_k = c_k, \quad (2.63)$$

a svaki pojedini konjunkt ($x_i = c_i$) predstavlja primitivno rekurzivnu relaciju, karakteristične funkcije $\chi_{=} \circ (I_i^k, C_{c_i}^k)$, koja je primitivno rekurzivna po korolaru 2.40, propoziciji 2.19 i definiciji 2.13. Dakle R je primitivno rekurzivna po propoziciji 2.42. \square

Korolar 2.46. *Svaka konačna brojevena relacija je primitivno rekurzivna.*

Dokaz. Direktno po propoziciji 2.42 i lemi 2.45, jer je $\{\vec{c}_1, \vec{c}_2, \dots, \vec{c}_l\} = \bigcup_{i=1}^l \{\vec{c}_i\}$. \square

Propozicija 2.47. *Neka je $k \in \mathbb{N}_+$, neka je G^k (primitivno) rekurzivna funkcija, te neka je F^k totalna funkcija koja se podudara s G u svima osim konačno mnogo točaka. Tada je i F (primitivno) rekurzivna.*

Naglasimo, totalnost funkcije F je esencijalna. Izbacivanjem već jedne točke iz \mathcal{D}_G dobit ćemo funkciju koja nije primitivno rekurzivna, po kontrapoziciji propozicije 2.17.

Dokaz. Po pretpostavci, skup $\{\vec{x} \in \mathbb{N}^k \mid F(\vec{x}) \neq G(\vec{x})\}$ je konačan: označimo mu sve elemente (recimo, poredane leksikografski) s $\vec{c}_1, \vec{c}_2, \dots, \vec{c}_l$. Ako je taj skup prazan, tada je $F = G$ pa je (primitivno) rekurzivna po pretpostavci. Inače je

$$F(\vec{x}) = \begin{cases} F(\vec{c}_1), & \vec{x} = \vec{c}_1 \\ \vdots & \\ F(\vec{c}_l), & \vec{x} = \vec{c}_l \\ G(\vec{x}), & \text{inače} \end{cases} = \begin{cases} C_{F(\vec{c}_1)}^k(\vec{x}), & \vec{x} \in \{\vec{c}_1\} \\ \vdots & \\ C_{F(\vec{c}_l)}^k(\vec{x}), & \vec{x} \in \{\vec{c}_l\} \\ G(\vec{x}), & \text{inače} \end{cases}, \quad (2.64)$$

dakle F je dobivena grananjem iz funkcija $C_{F(\vec{c}_i)}^k$ (koje su primitivno rekurzivne po propoziciji 2.19, jer je $F(\vec{c}_i) \in \mathbb{N}$ zbog totalnosti od F), funkcije G koja je (primitivno) rekurzivna po pretpostavci propozicije, te relacija $\{\vec{c}_i\}$ koje su primitivno rekurzivne po propoziciji 2.45. Po teoremu 2.44, F je također (primitivno) rekurzivna. \square

Sada napokon možemo formalizirati intuiciju *lookup*-tablice, koja kaže da su funkcije zadane na konačnom skupu izračunljive.

Korolar 2.48. *Neka je $k \in \mathbb{N}_+$, te g^k konačna funkcija (\mathcal{D}_g je konačan skup).*

Tada postoji primitivno rekurzivna funkcija F takva da je $F|_{\mathcal{D}_g} = g$.

Dokaz. Za F uzmemo \tilde{g} , proširenje funkcije g nulom. Tada se F^k i C_0^k razlikuju samo na konačnom skupu \mathcal{D}_g , pa tvrdnja slijedi iz propozicija 2.47 i 2.19. \square

2.4.3. Ograničene sume, produkti i brojenje

Vidjeli smo da, kao što kompozicija odgovara slijednom izvršavanju naredaba u imperativnom modelu, primitivna rekurzija odgovara ograničenim petljama. Kad malo bolje pogledamo, postoji neka vrsta analogije između ta dva pojma.

Kompoziciju koristimo u slučaju *statičke* granice, najčešće direktno vezane uz mjesnost l , gdje nam je svih l argumenata zadano eksplicitnim, često različitim, funkcijama. Dakle, prvo specificiramo $l \in \mathbb{N}_+$ i l -mjesnu funkciju H , zatim specificiramo l funkcija G_1, \dots, G_l , i tek onda uvrštavamo ulazne podatke. U tom smislu, l nije ulazni

podatak, već parametar konstrukcije: za različite l (i time različite H), dobit ćemo različite kompozicije, koje se onda mogu računati s (čak istim, ako je k konstantan) ulaznim podacima \vec{x} .

S druge strane, primitivnu rekurziju koristimo u slučaju *dinamičke* granice, najčešće vezane uz broj koraka izračunavanja, gdje su nam y različitih „argumenata” (npr. konfiguracija) dobiveni iteracijom jedne te iste funkcije H počevši od početne konfiguracije dobivene funkcijom G . Dakle, prvo specificiramo k i k -mjesnu funkciju G , zatim specificiramo *jednu* $(k + 2)$ -mjesnu funkciju, u koju pored početnih ulaznih podataka \vec{x} uvrštavamo još i y , redni broj koraka (prolaza kroz petlju) koji računamo.

Na taj način, y je kao „dinamički l ”: izgubili smo mogućnost rada s parcijalnim funkcijama, ali dobili smo mogućnost da broj koraka izračunavanja prenesemo kao ulazni podatak. Slikovito, u jednom slučaju računamo $G_i(\vec{x})$, a u drugom $G(\vec{x}, i)$, pa možemo reći da smo „dignuli supskript” (ili „spustili superskript”, ako se radi o mjesnosti) na razinu ulaznog podatka. U ovoj točki napraviti ćemo nekoliko takvih „dinamizacija”, za funkcije odnosno familije funkcija koje smo već upoznali, kao i za neke koje još nismo ali se svejedno prirodno i često pojavljuju.

Primjer 2.49. Jedan jednostavan primjer smo već vidjeli: s jedne (statičke) strane, za svaki n imamo primitivno rekurzivnu funkciju C_n^k takvu da je $C_n^k(\vec{x}) = n$, ali za različite n to su različite funkcije. S druge (dinamičke) strane, imamo i primitivno rekurzivnu funkciju I_{k+1}^{k+1} takvu da je za sve $n \in \mathbb{N}$, $I_{k+1}^{k+1}(\vec{x}, n) = n$. Iako je I_{k+1}^{k+1} inicijalna funkcija, najčešće će takve dinamičke funkcije biti definirane primitivnom rekurzijom (primijetimo da je C_n^k definirana kompozicijom). I ovdje možemo „definirati”

$$I_{k+1}^{k+1}(\vec{x}, 0) = C_0^k(\vec{x}), \quad I_{k+1}^{k+1}(\vec{x}, y + 1) = Sc(I_{k+1}^{k+1}(\vec{x}, y)), \quad (2.65)$$

odnosno $I_{k+1}^{k+1} = C_0^k \circ Sc \circ I_{k+2}^{k+2}$, ali I_{k+2}^{k+2} nije ni po čemu jednostavnija od I_{k+1}^{k+1} . \triangleleft

Evo malo kompliciranijeg primjera: u lemi 2.41, dokazali smo da je za svaki k , funkcija add^k primitivno rekurzivna; ako joj damo k brojeva, ili izračunljivih funkcija, ona će ih zbrojiti (funkcije će zbrojiti na presjeku njihovih domena). Sada ćemo napraviti dinamičku varijantu: operator \sum koji za unaprijed zadanu *totalnu* funkciju G prima broj koji kaže koliko prvih njenih vrijednosti treba zbrojiti, te eventualno ostale argumente od G ako je ona mjesnosti veće od 1. Isto ćemo napraviti za množenje.

Lema 2.50. *Neka je $k \in \mathbb{N}_+$, te G^k (primitivno) rekurzivna funkcija. Tada su (primitivno) rekurzivne i funkcije F_1^k i F_2^k , zadane s*

$$F_1(\vec{x}, y) := \sum_{i < y} G(\vec{x}, i), \quad F_2(\vec{x}, y) := \prod_{i < y} G(\vec{x}, i). \quad (2.66)$$

Dokaz. Kao što smo rekli, prirodno ih je zadati primitivnom rekurzijom.

$$F_1(\vec{x}, 0) := 0 \quad F_2(\vec{x}, 0) := 1 \quad (2.67)$$

$$F_1(\vec{x}, y + 1) := F_1(\vec{x}, y) + G(\vec{x}, y) \quad F_2(\vec{x}, y + 1) := F_2(\vec{x}, y) \cdot G(\vec{x}, y) \quad (2.68)$$

Treba primijetiti da su svi ti pozivi funkcija mjesnosti k , dakle \vec{x} je duljine $k - 1$, odnosno x -eva uopće nema ako je $k = 1$. To ne smeta u napisanim jednadžbama jer nijednu funkciju ne pozivamo samo na \vec{x} , ali treba uzeti u obzir da je tim jednadžbama napisana obična primitivna rekurzija (definicija 2.9) za $k > 1$, a degenerirana primitivna rekurzija (propozicija 2.20, ili korolar 2.34) za $k = 1$. \square

Napomena 2.51. Lema 2.50 nam omogućuje da zadamo broj pribrojnika ili faktora kao argument funkcije. Naravno, tada možemo i *izračunati* broj pribrojnika odnosno faktora kao neku izračunljivu funkciju ostalih argumenata. Precizno, za $k > 1$, možemo računati i funkcije

$$F_3^{k-1}(\vec{x}) := \sum_{i < H(\vec{x})} G(\vec{x}, i) \quad i \quad F_4^{k-1}(\vec{x}) := \prod_{i < H(\vec{x})} G(\vec{x}, i), \quad (2.69)$$

gdje je H^{k-1} neka izračunljiva funkcija, jednostavnim komponiranjem funkcije F_1 odnosno F_2 s koordinatnim projekcijama i s funkcijom H .

Recimo, za $k = 3$ vrijedi $F_4^2(x, y) = \prod_{i < H(x, y)} G(x, y, i) = F_2^3(x, y, H(x, y))$. \triangleleft

U uvodnom učenju programiranja, obično se prije sumiranja nauči *brojiti* elemente koji zadovoljavaju neko svojstvo: brojač se inicijalizira na 0, te prolazimo kroz sve elemente koji dolaze u obzir, i za svaki koji zadovoljava svojstvo, inkrementiramo brojač. Zgodno je primijetiti da je, s obzirom na to da smatramo bool podskupom od \mathbb{N} , uz $false = 0$ i $true = 1$, brojenje specijalni slučaj sumiranja: to je sumiranje istinitosnih vrijednosti. Naime, if uvjet: brojač += 1 možemo jednostavno zamijeniti s brojač += uvjet, čime algoritam za brojenje postaje obični algoritam za sumiranje. Formalizirajmo to.

Lema 2.52. Neka je $k \in \mathbb{N}_+$, te R^{k+1} (primitivno) rekurzivna relacija i H^k (primitivno) rekurzivna funkcija. Tada je funkcija F^k , zadana s

$$F(\vec{x}) := \text{card}\{y \in \mathbb{N} \mid y < H(\vec{x}) \wedge R(\vec{x}, y)\}, \quad (2.70)$$

također (primitivno) rekurzivna. Skraćeno pišemo $F(\vec{x}) := (\#y < H(\vec{x}))R(\vec{x}, y)$.

Dokaz. Prema pretpostavci, karakteristična funkcija χ_R^{k+1} je (primitivno) rekurzivna. Sada tvrdnja slijedi iz napomene 2.51, jer tvrdimo da vrijedi

$$(\#y < H(\vec{x}))R(\vec{x}, y) = \sum_{y < H(\vec{x})} \chi_R(\vec{x}, y). \quad (2.71)$$

Doista, za svaki $\vec{x} \in \mathbb{N}^k$, ako skup $S := [0..H(\vec{x})]$ rastavimo u dva dijela,

$$S_1 := \{y \in S \mid R(\vec{x}, y)\}, \quad i \quad (2.72)$$

$$S_2 := \{y \in S \mid \neg R(\vec{x}, y)\}, \quad (2.73)$$

tada je $\sum_{y \in S} \chi_R(\vec{x}, y) = \sum_{y \in S_1} 1 + \sum_{y \in S_2} 0 = \text{card } S_1$. \square

2.4.4. Ograničena kvantifikacija i minimizacija

Još jedan čest obrazac (*pattern*) u uvodnim algoritmima je provjera zadovoljava li neki element zadanog konačnog skupa neko zadano svojstvo — ili dualno, zadovoljavaju li ga svi elementi tog skupa. U nekim modernim programskim jezicima to se ostvaruje kroz funkcije `any` i `all`. Na primjer, ustanoviti je li broj n složen možemo tako da ispitamo postoji li $d \in [2..n)$ (ili $d \in [2..\lfloor \sqrt{n} \rfloor]$) takav da $d \mid n$. Vidimo da je prirodna matematička formalizacija tog obrasca *ograničena kvantifikacija*, gdje univerzalno ili egzistencijalno kvantificiramo varijable u nekom uvjetu do neke granice.

Razlog zašto se takav obrazac promatra posebno je mogućnost prijevremenog izlaska iz petlje (*shortcircuit evaluation*, u ovom slučaju najčešće realizirana kroz naredbu `break`), jer ako provjeravamo postoji li element s nekim svojstvom, znamo da postoji onog trena kada ga nađemo — ne moramo provjeravati ostale elemente. Tako možemo ubrzati pretraživanje po konačnim skupovima, a i dobiti odgovor u slučaju prebrojivih skupova *ako* je taj odgovor pozitivan. O ovom drugom fenomenu reći ćemo više kasnije, kad budemo govorili o *rekurzivno prebrojivim* relacijama — a prvi fenomen nas ne zanima, jer smo rekli da se nećemo baviti performansama algoritama.

Drugim riječima, za nas se ograničena kvantifikacija lako svodi na brojenje. Prebrojivši elemente do z koji imaju traženo svojstvo, očito takvi postoje ako ih ima pozitivan broj, a znamo da su svi takvi ako ih ima upravo z .

Propozicija 2.53. *Neka je $k \in \mathbb{N}_+$, te R^{k+1} (primitivno) rekurzivna relacija i H^k (primitivno) rekurzivna funkcija. Tada su i relacije P^k i Q^k , zadane s*

$$P(\vec{x}) :\iff (\exists y < H(\vec{x}))R(\vec{x}, y), \quad (2.74)$$

$$Q(\vec{x}) :\iff (\forall y < H(\vec{x}))R(\vec{x}, y), \quad (2.75)$$

također (primitivno) rekurzivne.

Dokaz. Kao u dokazu leme 2.52, fiksirajmo \vec{x} i uvedimo oznake $S := [0..H(\vec{x}))$, te $S_1 := \{y \in S \mid R(\vec{x}, y)\} \subseteq S$. Sada vrijedi

$$P(\vec{x}) \iff S_1 \neq \emptyset \iff \text{card } S_1 > 0 \iff (\#y < H(\vec{x}))R(\vec{x}, y) \in \mathbb{N}_+, \text{ te} \quad (2.76)$$

$$Q(\vec{x}) \iff S = S_1 \iff \text{card } S_1 = \text{card } S \iff (\#y < H(\vec{x}))R(\vec{x}, y) = H(\vec{x}) \quad (2.77)$$

(za smjer $\text{card } S_1 = \text{card } S \Rightarrow S_1 = S$ koristimo rezultat iz teorije skupova da konačan skup ne može biti ekvipotentan svom pravom podskupu), pa (primitivna) rekurzivnost od P slijedi iz primjera 2.26, a (primitivna) rekurzivnost od Q iz korolara 2.40. Primijetimo samo da je (2.76) dinamizirani (2.56). \square

Napomena 2.54. U svim dosadašnjim rezultatima ove točke imali smo isključenu gornju granicu ($y < H(\vec{x})$ kao pokrata za $y \in [0..H(\vec{x}))$). Često je, posebno u zapisima

suma i produkata, uobičajeno koristiti uključene granice: recimo, $\sum_{y=0}^{H(\vec{x})} G(\vec{x}, y) = \sum_{y \leq H(\vec{x})} G(\vec{x}, y)$. Sve takve funkcije su jednako izračunljive kao i s isključenom granicom, što je jednostavna posljedica činjenice da je $y \leq H(\vec{x}) \Leftrightarrow y < 1 + H(\vec{x}) = (Sc \circ H)(\vec{x})$ — dakle, samo umjesto funkcije H u gornje rezultate trebamo uvrstiti $Sc \circ H$, koja je (primitivno) rekurzivna ako je H takva. \triangleleft

Još uvijek ne znamo ništa o izračunljivosti *neograničene* kvantifikacije, odnosno onog što smo nazvali *projekcija relacije* R . Kasnije ćemo pokazati da postoje izračunljive relacije R čije projekcije $\exists_* R$ nisu izračunljive. Kako je po definiciji $\exists_* R = \mathcal{D}_{\mu R}$, zaključujemo da **domena izračunljive funkcije ne mora biti izračunljiva!** Što se tu točno zbiva, objasniti ćemo u poglavlju 7. Zasad pokažimo da *ograničenom* minimizacijom čuvamo izračunljivost.

Definicija 2.55. Neka je $k \in \mathbb{N}_+$, R^{k+1} relacija, te H^k totalna funkcija. Za funkciju F^k definiranu s

$$F(\vec{x}) := \mu y (y < H(\vec{x}) \rightarrow R(\vec{x}, y)), \quad (2.78)$$

kažemo da je dobivena *ograničenom minimizacijom* iz relacije R , do granice H . Skraćeno pišemo $F(\vec{x}) := (\mu y < H(\vec{x})) R(\vec{x}, y)$. \triangleleft

Napomena 2.56. U definiciji smo napisali znak $:=$ umjesto \simeq , jer desna strana uvijek ima smisla: $y < H(\vec{x})$ je sigurno laž za neki y (prvi takav je upravo $H(\vec{x})$, koji postoji jer je H totalna), pa je kondicional tada istinit. Drugim riječima, $F(\vec{x})$ će biti najmanji broj $y < H(\vec{x})$ koji zadovoljava $R(\vec{x}, y)$ ako takav postoji, a inače će biti $F(\vec{x}) = H(\vec{x})$.

Dakle, ograničenom minimizacijom do totalne granice dobivamo totalnu funkciju. Štoviše, ako su H i R rekurzivne, F će biti parcijalno rekurzivna — jer je dobivena neograničenom minimizacijom relacije zadane s $P(\vec{x}, y) :\Leftrightarrow y < H(\vec{x}) \rightarrow R(\vec{x}, y)$, koja je rekurzivna zbog primjera 2.27 i propozicije 2.39. Ukratko, ograničenom minimizacijom rekurzivne relacije do rekurzivne granice dobivamo rekurzivnu funkciju. \triangleleft

Drugim riječima, ograničena minimizacija čuva rekurzivnost. Važno je da čuva i *primitivnu* rekurzivnost, što ćemo sada dokazati. Primijetimo samo da će dokaz biti teži jer ne možemo koristiti neograničenu minimizaciju u postupku.

Pokušajmo otkriti kako bismo pomoću dosadašnjih konstrukcija — ograničenog brojenja i kvantifikacije — karakterizirali ograničenu minimizaciju. Dakle, imamo $k \in \mathbb{N}_+$, $(k+1)$ -mjesnu relaciju R , k -mjesnu totalnu funkciju H , i $\vec{x} \in \mathbb{N}^k$. Gornju granicu $H(\vec{x})$ označimo s α , i uvedimo jednomjesnu relaciju $P(y) :\Leftrightarrow R(\vec{x}, y)$ (\vec{x} je ionako fiksiran). Za početak pretpostavimo da doista postoji $y < \alpha$ za koji vrijedi $P(y)$. Dakle $\alpha > 0$, i imamo niz istinitosnih vrijednosti $P(0), P(1), \dots, P(\alpha - 1)$ u kojem trebamo naći prvu vrijednost *true*.

Kad bismo imali naredbu *break*, mogli bismo jednostavno prebrojiti vrijednosti *false*, i napustiti petlju čim naiđemo na *true*. Kako možemo emulirati *break*? Najjednostavniji

način je da imamo još jednu bool varijablu (*flag*) q , inicijaliziramo je na *false*, kad naiđemo na *true* postavimo je na *true*, te uopće ne izvršavamo petlju ako je ta varijabla *true*. Gledajući kako se ta varijabla mijenja, vidimo da je njen sadržaj u y -tom prolasku upravo istinitosna vrijednost $Q(y) : \iff (\exists i \leq y) P(i)$.

Sada je lako: brojenje prvog bloka nula (do prve jedinice) za P ekvivalentno je brojenju *svih* nula za Q , jer je χ_Q rastuća: kad postane 1, ostaje 1. Dakle, samo je još potrebno negirati Q , i primijeniti operator brojenja.

Gornje razmišljanje možemo prikazati tablicom: recimo da je P svojstvo „biti prim-broj”, i da tražimo $(\mu y < 7)(y \in \mathbb{P})$, najmanji prim-broj manji od 7. Imali bismo

i	0	1	2	3	4	5	6	7
$i \in \mathbb{P} \iff P(i)$	\perp	\perp	\top	\top	\perp	\top	\perp	
$(\exists j \leq i) P(j) \iff Q(i)$	\perp	\perp	\top	\top	\top	\top	\top	
$\neg Q(i)$	\top	\top	\perp	\perp	\perp	\perp	\perp	
$(\#j < i) \neg Q(j)$	0	1	2	2	2	2	2	2

(2.79)

Propozicija 2.57. *Neka je $k \in \mathbb{N}_+$, te R^{k+1} primitivno rekurzivna relacija i H^k primitivno rekurzivna funkcija. Tada je i funkcija F^k , dobivena ograničenom minimizacijom relacije R do granice H , također primitivno rekurzivna.*

Dokaz. Tvrdimo da je za svaki $\vec{x} \in \mathbb{N}^k$, vrijednost $F(\vec{x})$ jednaka

$$(\mu y < H(\vec{x})) R(\vec{x}, y) = (\#y < H(\vec{x})) (\nexists i \leq y) R(\vec{x}, i). \quad (2.80)$$

Prvo, ako ne postoji $y < H(\vec{x})$ takav da vrijedi $R(\vec{x}, y)$, tada na lijevoj strani stoji $H(\vec{x})$ po napomeni 2.56. S druge strane, za sve $y < H(\vec{x})$, svaki $i \leq y$ je također manji od $H(\vec{x})$, pa ni za koji od njih ne vrijedi $R(\vec{x}, i)$. Dakle $(\nexists i \leq y) R(\vec{x}, i)$ je uvijek istina, te je desna strana jednaka $(\#y < H(\vec{x})) \top$, što je također jednako $H(\vec{x})$.

Ako pak postoji neki $y < H(\vec{x})$ takav da vrijedi $R(\vec{x}, y)$, najmanji takav označimo sa y_0 . Tada je vrijednost lijeve strane upravo y_0 . Kako je y_0 najmanji broj takav da vrijedi $R(\vec{x}, y_0)$, $(\nexists i \leq y) R(\vec{x}, i)$ je još uvijek istina za sve $y < y_0$. Također, za sve $y \geq y_0$, ta relacija *nije* istinita, jer uvijek možemo uzeti $i := y_0$ kao kontraprimjer. Dakle svi oni y za koje vrijedi $(\nexists i \leq y) R(\vec{x}, i)$ su upravo svi oni manji od y_0 , te na desnoj strani piše njihov broj, koji je također y_0 .

Sada samo treba primijetiti da je funkcija na desnoj strani izraza (2.80) primitivno rekurzivna, kao posljedica (redom) napomene 2.54, propozicije 2.37 i leme 2.52. \square

U iskazu ove propozicije nismo trebali stavljati riječ „primitivno” u zagrade, jer tvrdnja čitana bez riječi „primitivno” slijedi iz napomene 2.56.

Napomena 2.58. Ista napomena o uključenoj gornjoj granici, poput 2.54, vrijedi i ovdje: $\mu y \leq H(\vec{x})$ jednostavno shvaćamo kao $\mu y < Sc(H(\vec{x}))$, a kompozicijom s inicijalnom funkcijom Sc sigurno nismo pokvarili izračunljivost granice H . \triangleleft

Poglavlje 3.

Univerzalna izračunljivost

3.1. Kodiranje

Napokon smo napravili dovoljno alata da možemo i prilično komplicirane funkcije dokazati primitivno rekurzivnima. Sljedeći veliki zalogaj koji ćemo uzeti je kodiranje, specijalno kodiranje konačnih nizova prirodnih brojeva (skraćeno, kodiranje \mathbb{N}^*).

Zašto baš to kodiranje? Dva su razloga. Prvo, trebat će nam za opis rada RAM-stroja, tako da možemo raditi s konfiguracijama i izračunavanjima kao s ulaznim podacima. Recimo, imat ćemo funkciju U koja će primiti izračunavanje koje je stalo, i vratiti izlazni podatak (rezultat) tog izračunavanja. Uбудuće ćemo često tako neformalno govoriti: „funkcija prima izračunavanje”, ili „funkcija vraća RAM-program”, misleći pritom na kanonsko kodiranje izračunavanja odnosno RAM-programa. Znamo da su RAM-programi konačni nizovi instrukcija, a izračunavanja koja stanu se također mogu pamtit i samo do završne konfiguracije kao konačni nizovi konfiguracija. Same pak konfiguracije također se mogu pamtit i kao konačni nizovi — vrijednost programskog brojača i sadržaj samo relevantnih registara. Vidjet ćemo da se mnogi složeni objekti koje ćemo željeti kodirati mogu prikazati kao konačni nizovi jednostavnijih objekata, te ako već imamo kodiranje tih jednostavnijih objekata, kodiranje \mathbb{N}^* dat će nam odmah mogućnost kodiranja složenih objekata.

Drugi razlog je jednostavnost specifikacije. Kodiranja su zapravo opisana algoritmima, čiji izlazni podaci su prirodni brojevi, a ulazni podaci su *nešto drugo* osim prirodnih brojeva. Općenito može biti komplicirano specificirati takav algoritam, jer on praktički po definiciji ne može biti formalni algoritam (recimo, RAM-algoritam) — ulazni podaci mu nisu prirodni brojevi, a da bismo ih prikazali kao prirodne brojeve, trebamo upravo kodiranje koje pokušavamo implementirati! Kodiranje \mathbb{N}^* pruža jednostavan izlaz iz tog začaranog kruga, jer imamo formalizaciju algoritama koji primaju konačne nizove prirodnih brojeva: za fiksnu duljinu niza (k -torke) to su jednostavno k -mjesni algoritmi, a za proizvoljnu duljinu niza bit će to familija algoritama Code^k , $k \in \mathbb{N}_+$ (i još jedna konstanta kao kod praznog niza, koja igra ulogu Code^0). Ipak, prvo pokušajmo preciznije definirati općenita kodiranja.

Definicija 3.1. Neka je \mathcal{K} neki skup (koji ne sadrži prirodne brojeve nego neku drugu vrstu objekata). *Kodiranje skupa* \mathcal{K} je izračunljiva (totalna) injekcija $\mathbb{N}\mathcal{K}: \mathcal{K} \rightarrow \mathbb{N}$, kojoj je slika (skup svih kodova) $\mathcal{I}_{\mathbb{N}\mathcal{K}}$ rekurzivan skup, a parcijalni inverz (lijevi inverz s obzirom na kompoziciju) $\mathbb{N}\mathcal{K}^{-1}: \mathbb{N} \rightarrow \mathcal{K}$, s domenom $\mathcal{D}_{\mathbb{N}\mathcal{K}^{-1}} = \mathcal{I}_{\mathbb{N}\mathcal{K}}$, je također izračunljiva funkcija. \triangleleft

Napominjemo da $\mathbb{N}\mathcal{K}$ i $\mathbb{N}\mathcal{K}^{-1}$ jesu izračunljive, dakle imamo (neformalne) algoritme za njih — ali to nisu brojevne funkcije, pa te algoritme nemamo u formalnom smislu (recimo, nema smisla reći „ $\mathbb{N}\mathcal{K} \in \text{Comp}$ ”). Ipak, možemo biti precizni u pogledu izračunljivosti skupa $\mathcal{I}_{\mathbb{N}\mathcal{K}}$: kako je to obični podskup od \mathbb{N} , dakle jednomjesna relacija, zahtijevamo da njena karakteristična funkcija bude rekurzivna.

Pokušajmo još malo preciznije odrediti što mislimo pod izračunljivošću funkcija $\mathbb{N}\mathcal{K}$ i $\mathbb{N}\mathcal{K}^{-1}$. Zamislimo da imamo funkciju $g: \mathcal{K} \rightarrow \mathcal{K}$, za koju želimo utvrditi je li izračunljiva. Tada možemo na $\mathcal{I}_{\mathbb{N}\mathcal{K}}$ definirati tzv. *prateću funkciju* (*tracking function*) $\mathbb{N}g := \mathbb{N}\mathcal{K} \circ g \circ \mathbb{N}\mathcal{K}^{-1}$, koja uzme kod c , iz njega odredi jedinstveni $\kappa \in \mathcal{K}$ takav da je $\mathbb{N}\mathcal{K}(\kappa) = c$, primijeni g na κ , te rezultat (ako je definiran, odnosno ako je $\kappa \in \mathcal{D}_g$) kodira natrag funkcijom $\mathbb{N}\mathcal{K}$. Ključno je primijetiti da je *prateća funkcija* uvijek brojeva. Ako je ona izračunljiva u nekom smislu (recimo parcijalno rekurzivna), te je kodiranje relativno kanonsko, prirodno je smatrati funkciju g izračunljivom u tom istom smislu. Primijetimo samo da $\mathbb{N}\mathcal{K} \circ g \circ \mathbb{N}\mathcal{K}^{-1}$ ne možemo smatrati simboličkom definicijom *prateće funkcije*, jer to nije kompozicija brojevnihih funkcija. Moramo nekako drugačije, samo koristeći prirodne brojeve, karakterizirati *prateću funkciju*. Ako to uspijemo za dovoljno intuitivno izračunljivih funkcija g , to je argument za tvrdnju da imamo izračunljivo kodiranje.

Slično možemo činiti za funkcije iz \mathcal{K} u \mathbb{N} (samo ih komponiramo s $\mathbb{N}\mathcal{K}^{-1}$ zdesna), za funkcije iz \mathbb{N} u \mathcal{K} (samo ih komponiramo s $\mathbb{N}\mathcal{K}$ slijeva — primijetite da su karakteristične funkcije specijalni slučaj *pratećih funkcija*, za $\mathbb{N}\text{bool}(\text{false}) := 0, \mathbb{N}\text{bool}(\text{true}) := 1$), te čak za razne „višemjesne” funkcije iz skupova poput $\mathcal{K}^2 \times \mathcal{L} \times \mathbb{N}$ (gdje je \mathcal{L} neki drugi skup čije kodiranje $\mathbb{N}\mathcal{L}$ već imamo): ulaze dekodiramo (ulaze koji već jesu prirodni brojevi ostavimo nepromijenjene), primijenimo funkciju, te kodiramo rezultat ako je definiran. Puna implementacija tog principa odvela bi nas u *objektno programiranje*, gdje je \mathcal{K} *klasa*, čije razne *metode* kodiramo na opisani način. Često među tim metodama postoji jedna istaknuta surjekcija na \mathcal{K} (ili više njih čije slike čine partciju od \mathcal{K}) koju onda zovemo *konstruktor*, te koordinatne funkcije njenog inverza (*getters*, ili jednostavno *komponente*) pomoću kojih možemo napraviti sve ostale metode.

Napomena 3.2. U još jednoj stvari budimo precizni: **neprebrojive skupove ne možemo kodirati!** Doista, ako postoji kodiranje kao injekcija s \mathcal{K} u \mathbb{N} , tada mora biti $\text{card } \mathcal{K} \leq \text{card } \mathbb{N} = \aleph_0$. Ovo je izuzetno važno, jer pruža opravdanje za intuiciju da samo konačni i prebrojivi skupovi imaju *totalnu reprezentaciju*: sve njihove elemente možemo reprezentirati u (po volji velikom) računalu.

Neegzaktosti tipa float, i raznih drugih tipova koji bi trebali reprezentirati realne brojeve, nisu samo tehnički nedostaci pojedinog standarda (kao što je IEEE 754): one su fundamentalna posljedica činjenice da \mathbb{R} kao neprebrojiv skup nema totalnu reprezentaciju. Kako god pokušali [3], ne možemo u računalu reprezentirati proizvoljan realan broj, i to nema veze s ograničenom veličinom naših računala. \triangleleft

3.1.1. Kodiranje konačnih nizova

Sada se možemo pozabaviti samom implementacijom kodiranja \mathbb{N}^* . Kako bismo najlakše kodirali $\vec{x} = (x_1, x_2, \dots, x_k)$ kao jedan prirodni broj, iz kojeg se kasnije mogu izvući pojedini brojevi x_i ? U teoriji skupova, za dokaz da je \mathbb{N}^2 prebrojiv, promatra se injekcija $p(x, y) := 2^x \cdot 3^y$, te se iskoristi osnovni teorem aritmetike (rastav na prim-faktore) da bi se iz $p(x, y)$ natrag dobili x i y . To se očito lako može proširiti na proizvoljnu mjesnost (jer prim-brojeva ima beskonačno mnogo — samo ih uzmemo dovoljno redom po veličini, potenciramo odgovarajućim eksponentima, i pomnožimo), ali to nije kodiranje skupa \mathbb{N}^* . Naime, takvo preslikavanje nije injekcija, jer se recimo $(1, 2, 0, 0)$ i $(1, 2)$ preslikaju u isti broj $2^1 \cdot 3^2 \cdot 5^0 \cdot 7^0 = 2^1 \cdot 3^2 = 18$.

Ipak, mala modifikacija dat će nam kodiranje. Zapravo, jedini problem su nule u konačnom nizu — opisano preslikavanje *jest* kodiranje skupa \mathbb{N}_+^* konačnih nizova *pozitivnih* prirodnih brojeva. Sada je još samo preostalo komponirati ga s izračunljivom bijekcijom Sc između \mathbb{N} i \mathbb{N}_+ , i dobili smo traženo kodiranje.

Definicija 3.3. Za $i \in \mathbb{N}$, sa p_i označimo i -ti po redu prim-broj (počevši od $p_0 = 2$). Definiramo $\langle \rangle := 1$, te za svaki $k \in \mathbb{N}_+$ definiramo funkciju

$$\text{Code}^k(x_1, x_2, \dots, x_k) := \langle x_1, x_2, \dots, x_k \rangle := 2^{x_1+1} \cdot 3^{x_2+1} \cdots p_{k-1}^{x_k+1}. \quad (3.1)$$

Time je definirana funkcija $\langle \dots \rangle: \mathbb{N}^* \rightarrow \mathbb{N}$. \triangleleft

To doduše nije brojeva funkcija jer nema fiksnu mjesnost, ali ipak možemo reći da je izračunljiva u istom smislu u kojem su višestruko zbrajanje i množenje izračunljive operacije po lemi 2.41.

Propozicija 3.4. Za svaki $k \in \mathbb{N}_+$, funkcija Code^k je primitivno rekurzivna.

Dokaz. Jednostavno, svaki faktor u produktu (3.1) možemo prikazati kao kompoziciju potenciranja, konstante, sljedbenika i koordinatne projekcije. Tada vanjska kompozicija s mul^k daje simboličku definiciju

$$\text{Code}^k = \text{mul}^k \circ (\text{pow} \circ (C_2^k, Sc \circ I_1^k), \text{pow} \circ (C_3^k, Sc \circ I_2^k), \dots, \text{pow} \circ (C_{p_{k-1}}^k, Sc \circ I_k^k)), \quad (3.2)$$

iz koje po propozicijama 2.19 i 2.14, primjeru 2.11, te lemi 2.41 slijedi tvrdnja. \square

Ovaj put smo napisali simboličku definiciju da bi bilo jasno da nigdje nismo trebali izračunljivost funkcije $n \mapsto p_n$. To svakako vrijedi, i trebat će nam kasnije, ali za pojedinu funkciju Code^k , dovoljno je da znamo da postoji bar k prim-brojeva, te da su konstante s tim vrijednostima izračunljive.

Propozicija 3.5. *Preslikavanje $\langle \dots \rangle: \mathbb{N}^* \rightarrow \mathbb{N}$ je injekcija.*

Dokaz. Pretpostavimo da su $\vec{x}^k, \vec{y}^l \in \mathbb{N}^*$ takvi da je $\langle \vec{x} \rangle = \langle \vec{y} \rangle$. Ako je $k > l$, tada je $k-1 \in \mathbb{N}$, te je $\langle \vec{x} \rangle$ djeljiv s p_{k-1} , dok $\langle \vec{y} \rangle$ to nije (konkretno, $p_{k-1} \notin \{p_0, p_1, \dots, p_{l-1}\}$, pa ne dijeli njihov produkt), kontradikcija. Analogno ne može biti $k < l$. Dakle je $k = l$. Za svaki $i \in [1..k]$, tvrdimo da ne može biti $x_i > y_i$. Kada bi bilo, tada bi kod bio djeljiv s $p_{i-1}^{y_i}$, te bi nakon dijeljenja lijeva strana bila djeljiva s p_{i-1} , a desna ne bi. Analogno se vidi da ne može biti $x_i < y_i$, dakle mora biti $x_i = y_i$. Kako je i bio proizvoljan, zaključujemo $\vec{x} = \vec{y}$. \square

Sljedeće bismo trebali pokazati da je $\text{Seq} := \mathcal{I}_{\langle \dots \rangle}$ rekurzivan skup, te da na tom skupu imamo izračunljivu funkciju koja nam na neki način daje originalni \vec{x} iz kojeg je pojedini kod dobiven. Za taj dokaz potrebno nam je ponešto teorije brojeva, no prije toga samo precizirajmo u kakvom obliku tražimo naš inverz.

Recimo odmah da tražimo primitivno rekurzivne, dakle totalne funkcije. Njihovo djelovanje će nas zanimati samo na skupu Seq : bit će parcijalno specificirano, iako naravno, kad damo algoritme, moći će se izračunati u svakom prirodnom broju.

Da bismo odredili konačni niz iz kojeg je dobiven kod c , svakako moramo prvo odrediti njegovu duljinu. Dakle, tražimo funkciju lh takvu da za svaki $c \in \text{Seq}$, $lh(c)$ bude duljina konačnog niza čiji je c kod. Takvih funkcija ima neprebrojivo mnogo jer je Seq^c beskonačan (recimo, sadrži sve neparne brojeve osim 1) a na njemu lh može djelovati proizvoljno — pa sigurno postoje i neizračunljive takve funkcije. Ipak, dokazat ćemo da postoji takva funkcija lh koja je primitivno rekurzivna.

Kad smo odredili $lh(c) =: k$, na prvi pogled imamo tipičnu situaciju algoritma s više izlaza: od jednog broja c trebamo dobiti k njih, i u skladu s napomenom 1.1, to bismo trebali reprezentirati pomoću koordinatnih funkcija $\text{part}_1, \text{part}_2, \dots, \text{part}_k$. Ipak, to nema puno smisla jer broj takvih funkcija ovisi o c (po funkciji lh), a osim toga, ponekad će nam trebati i dinamički određeni indeksi. Recimo, kad budemo pisali funkciju U , koja kôd izračunavanja koje stane preslikava u njegov rezultat, bit će potrebno odabrati *zadnju* konfiguraciju u konačnom nizu. A čak i da imamo izračunljive funkcije part_i za sve i , iz toga ne slijedi da je preslikavanje $c \mapsto \text{part}_{lh(c)}(c)$ izračunljivo (pokušajte napisati simboličku definiciju i vidjet ćete u čemu je problem).

Srećom, ideja dinamizacije i ovdje pomaže: zapravo ćemo imati *dvomjesnu* funkciju part^2 , tako da $\text{part}(c, i)$ (skraćena oznaka $c[i]$) bude ono što smo bili nazvali $\text{part}_{i+1}(c)$ — pomaknuli smo indekse za 1 jer je ideja shvatiti konačne nizove kao polja (*arrays*), a u većini modernih programskih jezika indeksiranje polja kreće od nule. Bitno nam

je da to vrijedi samo za $c \in \text{Seq}$ i $i \in [0..lh(c)]$ — za ostale uređene parove mora biti nekako definirana jer je primitivno rekurzivna, ali nije nam bitno kako točno. Zapravo će se ispostaviti da za $c \in \text{Seq}$ i $i \geq lh(c)$ vrijedi $c[i] = 0$, što će biti korisno u jednom trenutku, ali naglasit ćemo to kad nam bude trebalo.

Napomena 3.6. Ako za fiksni $k \in \mathbb{N}_+$ promatramo funkciju Code^k u jednom smjeru i funkcije $\text{part}_1, \dots, \text{part}_k$ u drugom, zapravo modeliramo ono što jezik C zove struktura (`struct`) s k članova. Razlog zašto C ima i polja i strukture kao zasebne tipove podataka leži u tome da strukture mogu sadržavati podatke različitih tipova. Kako mi sve kodiramo prirodnim brojevima, to nam neće bitno trebati. \triangleleft

Kad smo već kod programskog jezika C , vjerojatno znate da se polja u njemu najčešće prenose tako da se u jednom argumentu prenese pokazivač (što u našem slučaju odgovara kodu), a u drugom, zasebnom argumentu njegova duljina. Tada bismo mogli kodirati i bez sljedbenika u eksponentu, jer bi duljina do koje gledamo „memoriju” bila posebno zadana. Ipak, većina modernijih programskih jezika drži duljinu zajedno s pojedinim elementima spremnika (npr. Python ima ugrađenu funkciju `len`), te je zato i mi kodiramo tako da ju je moguće odrediti iz koda. Nama će ideja kodiranja bez sljedbenika biti bitna u jednom drugačijem slučaju, kada je duljina nespecificirana — ali o tome kasnije.

Funkcije Code^k su dovoljno dobre ako imamo fiksnu mjesnost, te možemo argumente zadati posebno. Što dobijemo ako primijenimo ideju dinamizacije na tu familiju funkcija? Očito, argumenti su tada zadani nekom funkcijom G , te posebni argument kaže koliko ih ima (usporedite s točkom 2.4.3). Operator koji dinamički kodira zadani broj vrijednosti neke funkcije zovemo operatorom *povijesti*, jer često argument koji kaže koliko vrijednosti treba kodirati zapravo predstavlja neku vrstu vremena, odnosno broji korake u nekom postupku (npr. P-izračunavanju s \vec{x}).

Definicija 3.7. Neka je $k \in \mathbb{N}_+$, i G^k totalna funkcija. Za funkciju F^k zadanu s

$$F(\vec{x}, y) := \langle G(\vec{x}, 0), G(\vec{x}, 1), \dots, G(\vec{x}, y-1) \rangle \quad (3.3)$$

(početak je $F(\vec{x}, 0) := \langle \rangle = 1$) kažemo da je *povijest* funkcije G , i pišemo $F := \overline{G}$. \triangleleft

Primjer 3.8. $\overline{\text{Code}}(0, 2) = \langle \langle 0, 0 \rangle, \langle 0, 1 \rangle \rangle = \langle 6, 18 \rangle = 2^7 \cdot 3^{19} = 148\,769\,467\,776$. Također,

$$\begin{aligned} \overline{\text{mul}}(2, 1, 3) &= \langle \text{mul}(2, 1, 0), \text{mul}(2, 1, 1), \text{mul}(2, 1, 2) \rangle = \langle 2 \cdot 1 \cdot 0, 2 \cdot 1 \cdot 1, 2 \cdot 1 \cdot 2 \rangle = \\ &= \langle 0, 2, 4 \rangle = 2^{0+1} \cdot 3^{2+1} \cdot 5^{4+1} = 2 \cdot 27 \cdot 3125 = 168\,750. \end{aligned} \quad (3.4)$$

Naravno, k može biti i 1; jedan važan slučaj je $G := Z$. Recimo,

$$\overline{Z}(5) = \langle Z(0), Z(1), Z(2), Z(3), Z(4) \rangle = \langle 0, 0, 0, 0, 0 \rangle = 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 = 2310. \quad (3.5)$$

Dakle, funkcija \overline{Z} je u uskoj vezi s funkcijom tzv. *primorijel*, koja je definirana slično kao faktorijel, osim što množi samo prim-brojeve. \triangleleft

Sada bismo htjeli dokazati rezultat analogan onom u lemi 2.50, samo za operator povijesti. Kao što smo već rekli, to će zahtijevati ponešto teorije brojeva.

3.1.2. Potrebni rezultati iz teorije brojeva

Dokazujemo niz rezultata o primitivnoj rekurzivnosti, navodeći uglavnom samo točkovne definicije (obilno koristeći programiranje s ograničenim petljama: sume, produkte, brojenje i minimizaciju), te obrazlažući ukratko manje poznate rezultate iz teorije brojeva koje ovdje koristimo.

Sjetimo se napomene 2.24: često brojevnoteorijske funkcije nisu definirane u nuli, te ćemo je zamijeniti jedinicom, pišući n' za n ako je pozitivan, a 1 ako je $n = 0$.

Propozicija 3.9. *Cjelobrojno dijeljenje, zadano s $x // y := \lfloor \frac{x}{y} \rfloor$, kao i ostatak cjelobrojnog dijeljenja $(x \bmod y)$, primitivno su rekurzivne operacije.*

Prije dokaza samo primijetimo da je nula problematična u operaciji $//$, ali nije u operaciji \bmod : uobičajeno se u modernoj teoriji brojeva definira $x \bmod 0 := x$.

Dokaz. Cjelobrojno dijeljenje zapravo je ponovljeno oduzimanje, onoliko puta koliko se može. Dakle, za zadane x i y , tražimo najveći t takav da je $t \cdot y \leq x$. Relacija $t \cdot y \leq x$ je svakako primitivno rekurzivna, ali nemamo operator maksimizacije. S razlogom ga općenito nemamo: ako malo razmislimo, vidjet ćemo da ne postoji općeniti algoritam kojim bismo odredili najveći broj s nekim svojstvom, čak ni ako znamo da taj broj postoji, jer nikad ne znamo jesmo li tražili dovoljno dugo — u svakom trenutku iznad najvećeg broja koji smo ispitali postoji još beskonačno mnogo kandidata, svaki od kojih može biti traženi broj.

Srećom, naša relacija je *padajuća* po t : jednom kad $t \cdot y$ postane veće od x , ostatak će veće od x za sve veće t . Kod takvih relacija, „prva laž” i „zadnja istina” su susjedne, te jednostavno možemo naći najmanji t za koji je $t \cdot y > x$, i uzeti njegov prethodnik.

Ipak, to je neograničena minimizacija, a mi bismo htjeli primitivno rekurzivnu funkciju — možemo li je nekako ograničiti? Svakako: $y' \geq 1$ znači $t \cdot y' \geq t$ za svaki t , te specijalno $(x + 1) \cdot y' \geq x + 1 > x$. Dakle, dovoljno je tražiti t do $x + 1$ — odnosno do x uključivo, jer znamo da će ograničena minimizacija tada dati $x + 1$ ako ne pronađe nijedan broj s traženim svojstvom u zadanom rasponu. Sve u svemu, tvrdimo da vrijedi

$$x // y = \text{pd}((\mu t \leq x)(t \cdot y > x)). \quad (3.6)$$

Za $y > 0$, samo trebamo formalizirati navedeni argument: označimo $z := x // y$. Tada je $zy \leq x$ (i očito $ty \leq zy \leq x$ za sve $t \leq z$) a $(z + 1)y > x$, dakle $z + 1$ je najmanji t koji pomnožen s y daje broj veći od x . S druge strane je $z \leq x // 1 = x$, pa je $z + 1 \leq x + 1$. Ako je $z + 1 \leq x$, tada će ga ograničena minimizacija u (3.6) naći, a ako je $z + 1 = x + 1$, tada ga neće naći do uključivo x (isključivo $x + 1$), pa će po definiciji vratiti upravo

$x + 1$. U svakom slučaju vrijednost te ograničene minimizacije bit će $z + 1$, pa će njen prethodnik biti $z = x // y$, što smo trebali.

Za $y = 0$, samo trebamo izračunati lijevu i desnu stranu. Na lijevoj je $x // 0 = \lfloor \frac{x}{1} \rfloor = x$, a na desnoj je prethodnik minimizacije po $t \leq x$, relacije $t \cdot 0 = 0 > x$. Ta relacija je očito uvijek lažna (nema negativnih brojeva), pa minimizacija daje $x + 1$, a njen prethodnik onda daje upravo x , kao što i treba.

Sada je jednostavno dobiti ostatak: tvrdimo da je

$$x \bmod y = x \ominus x // y \cdot y \quad (3.7)$$

(prioritet operacija je: $//$, pa \cdot , pa \ominus). Za $y > 0$, direktno iz teorema o dijeljenju s ostatkom dobijemo da je ostatak $x - z \cdot y$ (sa z smo označili količnik), a iz provedenog razmatranja se vidi da je $z \cdot y \leq x$, pa se oduzimanje može ograničiti nulom. Za $y = 0$, na lijevoj strani piše x , a na desnoj također $x \ominus x // 0 \cdot 0 = x \ominus x \cdot 0 = x \ominus 0 = x$. \square

Korolar 3.10. *Djeljivost je primitivno rekurzivna relacija.*

Dokaz. Svatko tko je ikad provjeravao djeljivost u programiranju zna kako se to radi:

$$y \mid x \iff x \bmod y = 0. \quad (3.8)$$

Zaista, za $y > 0$, postojanje broja z takvog da je $z \cdot y = x$ (definicija djeljivosti) zapravo znači da je $z = x // y$, pa je $x \bmod y = x \ominus z \cdot y = x \ominus x = 0$. U drugom smjeru, ako je $x \ominus x // y \cdot y = 0$, iz toga slijedi (kao i prije, vrijedi $x // y \cdot y \leq x$) $x = x // y \cdot y$, pa postoji $z := x // y$ takav da vrijedi $x = z \cdot y$, odnosno $y \mid x$.

Za $y = 0$, na desnoj strani stoji $x \bmod 0 = x = 0$, a to stoji i na lijevoj strani, jer je jedino 0 djeljiva nulom — „postoji z takav da je $0 \cdot z = x$ ” zapravo znači da je $x = 0$. \square

Korolar 3.11. *Skup \mathbb{P} svih prim-brojeva je primitivno rekurzivan.*

Dokaz. Direktno pomoću korolara 3.10 i ograničenog brojenja (lema 2.52): prim-brojevi imaju točno dva prirodna djelitelja, a za pozitivne x , svaki djelitelj od x je manji ili jednak x :

$$x \in \mathbb{P} \iff (\#d \leq x)(d \mid x) = 2. \quad (3.9)$$

Za nulu će tom metodom ispasti da ima jedan djelitelj (samu sebe), a zapravo ih ima beskonačno mnogo, ali bitno je samo da ih nema točno 2, odnosno $0 \notin \mathbb{P}$. \square

U „stvarnom životu”, provjera je li $x \in \mathbb{P}$ odvija se drugačije: nulu, jedinicu i dvojku, te sve ostale parne brojeve odvojimo kao specijalne slučajeve, a onda provjeravamo samo neparne kandidate za djelitelje od 3 do uključivo $\lfloor \sqrt{x} \rfloor$. Ipak, to su sve samo praktične optimizacije, koje bitno ubrzavaju algoritam — ostavljajući ga doduše u istoj klasi složenosti. Zanimljivo je napomenuti da je početkom ovog stoljeća pronađen *polinomni* algoritam za provjeru je li zadani broj prim-broj, ali je dosta kompliciraniji. Kako mi ovdje nismo opterećeni performansama, tražimo samo najelegantniji zapis algoritma, a to je bez sumnje (3.9).

Propozicija 3.12. *Niz $(p_i)_{i \in \mathbb{N}}$ (strogo rastući niz sa slikom \mathbb{P} , tzv. enumeracija skupa \mathbb{P}) je primitivno rekurzivan.*

Dokaz. Zapravo trebamo algoritam za funkciju **prime**¹, koja svakom broju n pridružuje p_n , n -ti prim-broj po veličini. Očito, treba nam primitivna rekurzija — i to degenerirana jer definiramo funkciju mjesnosti 1. Za inicijalizaciju stavimo samo vrijednost $p_0 = 2$, a u koraku trebamo funkciju **nextprime**² koja prima n i p_n , i mora vratiti p_{n+1} . Je li ta funkcija izračunljiva?

Kako uopće znamo da je ta funkcija *totalna*, odnosno da možemo pomoću nje graditi primitivnu rekurziju? Prim-brojeva ima beskonačno mnogo: za svaki p_n postoji $q \in \mathbb{P}$ takav da je $q > p_n$. Štoviše, p_{n+1} je prvi takav q , pa ga možemo naći minimizacijom:

$$\text{nextprime}(n, p) := \mu q (q \in \mathbb{P} \wedge q > p). \quad (3.10)$$

Upravo napisani izraz zapravo kaže da je **nextprime** *rekurzivna* funkcija (parcijalno je rekurzivna jer je dobivena minimizacijom konjunkcije dvije rekurzivne relacije, a totalna je zbog beskonačnosti skupa \mathbb{P}), pa je i **prime** rekurzivna po korolaru 2.34.

Za primitivnu rekurzivnost, moramo nekako ograničiti minimizaciju, odnosno moramo „isprogramirati” dokaz da je \mathbb{P} beskonačan. Uzmimo Euklidov dokaz: ako imamo $p_0, p_1, \dots, p_n \in \mathbb{P}$, novi prim-broj možemo dobiti tako da potražimo prim-djelitelj broja $m := p_0 \cdot p_1 \cdots p_n + 1 > 1$. Aha! Znamo da je djelitelje pozitivnog broja dovoljno tražiti do samog tog broja, dakle samo trebamo primitivno rekurzivno izračunati m iz n i p_n . Tu bismo mogli upotrijebiti primorijel iz primjera 3.8 — konkretno, $m = \text{Sc}(\bar{Z}(\text{Sc}(n)))$ — ali nažalost još ne znamo da je funkcija \bar{Z} primitivno rekurzivna, jer nismo dokazali da povijest čuva primitivnu rekurzivnost. I to s razlogom: pokazat će se da za tu lemu treba rezultat koji upravo dokazujemo.

Kako se izvući? Spas leži u tome da ne trebamo pomoću n i p_n izračunati baš m , nego samo neki broj od kojeg je m manji. Naime, ionako nećemo tražiti njegove prim-djelitelje, nego će nam on samo poslužiti kao gornja granica za traženje sljedećeg prim-broja nakon p_n . Sada je lako: umjesto primorijela, možemo upotrijebiti faktorijel, za koji znamo da je primitivno rekurzivan (primjer 2.22), a očito je $p_n! \geq p_0 \cdot p_1 \cdots p_n$ (jer je $p_n!$ umnožak svih brojeva na desnoj strani, i eventualno još nekih, koji su veći ili jednaki 1).

$$\text{nextprime}(n, p) = (\mu q (\leq p! + 1)(q \in \mathbb{P} \wedge q > p)) \quad (3.11)$$

Dakle **nextprime** je dobivena ograničenom minimizacijom primitivno rekurzivne relacije do primitivno rekurzivne (uključive, napomena 2.58) granice, te je primitivno rekurzivna. Sada je i **prime** = $2 \text{ PR } \text{nextprime}$ primitivno rekurzivna po propoziciji 2.20. \square

3.1.3. Rastav na prim-faktore

Lema 3.13. *Za $(n, i) \in \mathbb{N}^2$, označimo s $\text{ex}(n, i)$ eksponent prim-broja p_i u rastavu broja n na prim-faktore. Funkcija ex^2 je primitivno rekurzivna.*

Nula nema rastav na prim-faktore, pa je moramo zamijeniti jedinicom — koja *ima* jedinstven rastav na prim-faktore: prazan produkt, gdje su svi eksponenti jednaki 0.

Dokaz. Opet, tražimo najveći broj t takav da $p_i^t \mid n'$. Kao i u dokazu propozicije 3.9, ta relacija je padajuća po t : jednom kad prestane biti istina, ne može ponovo postati istina ni za koji veći t . Dakle, isti trik (prethodnik najmanjeg elementa komplementa) prolazi. Također, kako je *prime* rastuća funkcija, imamo $n' \geq p_i^t \geq p_0^t = 2^t > t$ (Cantorov osnovni teorem za konačne skupove), pa je t dovoljno tražiti do n' isključivo, ili do n jer su za $n = 0$ ionako svi eksponenti 0. Sve u svemu, vrijedi

$$\text{ex}(n, i) := \text{pd}((\mu t < n)(\text{pow}(\text{prime}(i), t) \nmid n)), \quad (3.12)$$

pa je *ex* primitivno rekurzivna. \square

Pomoću funkcije *ex* napokon možemo dekodirati proizvoljni kod konačnog niza.

Propozicija 3.14. *Postoje primitivno rekurzivne funkcije lh^1 i part^2 , takve da za svaki $\vec{x} = (x_1, x_2, \dots, x_k) \in \mathbb{N}^*$, uz oznaku $c := \langle \vec{x} \rangle$, te pokratu $c[i] := \text{part}(c, i)$, vrijedi:*

1. $\text{lh}(c) = k$;
2. za sve $i < k$, $c[i] = x_{i+1}$;
3. za sve $i \geq k$, $c[i] = 0$.

Dokaz. Tvrdimo da funkcije zadane s

$$\text{lh}(c) := (\#i < c)(\text{prime}(i) \mid c), \quad (3.13)$$

$$c[i] := \text{part}(c, i) := \text{pd}(\text{ex}(c, i)), \quad (3.14)$$

zadovoljavaju sve uvjete. Prije svega, primitivno su rekurzivne: *lh* je dobivena ograničenim brojenjem primitivno rekurzivne relacije do primitivno rekurzivne granice, a *part* je jednostavno kompozicija *pd* o *ex*.

Što se tiče tvrdnje 1, prema definiciji *lh* broji prim-djelitelje od c (kako je *prime* strogo rastuća i $0 < p_0$, vrijedi $i < p_i \leq c$, pa je dovoljno brojiti do c), a kako je $c = \langle \vec{x} \rangle = 2^{x_1+1} \cdot 3^{x_2+1} \cdots p_{k-1}^{x_{k-1}+1}$, te su svi napisani eksponenti pozitivni, c ima točno k prim-djelitelja.

Za tvrdnju 2, neka je $i < k$ proizvoljan. Tada je eksponent od p_i u rastavu $c > 0$ na prim-faktore upravo $\text{ex}(c, i) = x_{i+1} + 1$, te je $c[i] = \text{pd}(x_{i+1} + 1) = x_{i+1}$.

Za tvrdnju 3, neka je $i \geq k$ proizvoljan. Lako je vidjeti da su jedini prim-djelitelji od c upravo p_0, p_1, \dots, p_{k-1} , dakle p_i nije među njima. Dakle $(\mu t < c)(p_i^t \mid c) = 1$, pa je $\text{ex}(c, i) = \text{pd}(1) = 0$, te je i $c[i] = \text{pd}(0) = 0$. \square

Primijetimo da je funkcija *part* parcijalno specificirana samo po c : po i je totalno specificirana, odnosno jednom kad znamo da je $c \in \text{Seq}$, sve vrijednosti $c[i]$ su propisane. Još nam jedino nedostaje dokaz da je *Seq* primitivno rekurzivna relacija, no to će slijediti uskoro.

3.2. Funkcije definirane pomoću operatora povijesti

Lema 3.15 (Lema o povijesti). *Neka je $k \in \mathbb{N}_+$, i G^k totalna funkcija.*

Tada je G (primitivno) rekurzivna ako i samo ako je \overline{G} (primitivno) rekurzivna.

Za razliku od leme 2.50, ovdje se tvrde dva smjera. (Zapravo, i u lemi 2.50 vrijedi obrat za F_1 , dok za F_2 ne vrijedi. Zgodna je vježba pokušati to dokazati.)

Dokaz. Za jedan smjer (zapisati \overline{G} primitivno rekurzivno pomoću G), treba uvrstiti (3.1) u (3.3):

$$\overline{G}(\vec{x}, y) = \prod_{i < y} \text{pow}(\text{prime}(i), \text{Sc}(G(\vec{x}, i))). \quad (3.15)$$

Ako i -ti faktor u (3.15) označimo s $H(\vec{x}, i)$, tada je $\text{pow} \circ (\text{prime} \circ l_{k+1}^{k+1}, \text{Sc} \circ G)$ simbolička definicija od H^{k+1} kompozicijom iz funkcija za koje je već dokazano da su primitivno rekurzivne, te funkcije G koja je (primitivno) rekurzivna — pa je H (primitivno) rekurzivna. Sada je \overline{G} (primitivno) rekurzivna po lemi 2.50.

Za drugi smjer, pretpostavimo da nam je zadana \overline{G} , i želimo iz nje dobiti G . Pažljivim gledanjem (3.3) vidimo da $\overline{G}(\vec{x}, y)$ kodira prvih y vrijednosti oblika $G(\vec{x}, i)$, među kojima *nije* $G(\vec{x}, y)$. (To je čest početnički *bug* u C-programiranju: polje deklarirano s `int a[7]`; *ne* sadrži element `a[7]`.) Ako želimo dobiti taj broj, moramo zapravo izračunati \overline{G} u nekom broju *većem* od y kao zadnjem argumentu. Dovoljno je uzeti $\overline{G}(\vec{x}, y + 1)$: tvrdimo

$$G(\vec{x}, y) = \overline{G}(\vec{x}, \text{Sc}(y))[y], \quad (3.16)$$

odakle slijedi (primitivna) rekurzivnost od G , iz (primitivne) rekurzivnosti od \overline{G} . Doista, $c := \overline{G}(\vec{x}, y + 1)$ jest kod konačnog niza duljine $y + 1$, pa prema propoziciji 3.14(2) za svaki $i < y + 1$ vrijedi $c[i] = G(\vec{x}, i)$. Specijalno za $i := y$ imamo jednakost (3.16). \square

Napokon možemo dokazati primitivnu rekurzivnost slike kodiranja.

Korolar 3.16. *Relacija $\text{Seq}^1 := \mathcal{I}_{\langle \dots \rangle}$ je primitivno rekurzivna.*

Dokaz. Formalnije,

$$\text{Seq}(c) \iff (\exists \vec{x} \in \mathbb{N}^*)(c = \langle \vec{x} \rangle), \quad (3.17)$$

i način za odrediti \vec{x} je kanonski: pokušamo dekodirati c . (To funkcionira i općenito, ali razlog zašto se rekurzivnost slike navodi kao zasebno svojstvo funkcije kodiranja je u tome što ga u slučaju kodiranja nekih drugih objekata možemo provjeriti formalnije nego na ovaj način, koji općenito koristi neformalne algoritme.) Dakle, tvrdimo

$$\text{Seq}(c) \iff c = \overline{\text{part}}(c, \text{lh}(c)); \quad (3.18)$$

smjer (\Leftarrow) slijedi iz činjenice da je *svaka* vrijednost funkcije oblika \overline{G} kod konačnog niza.

Za smjer (\Rightarrow), pretpostavimo da je $c = \langle \vec{x} \rangle$ za neki $\vec{x} = (x_1, x_2, \dots, x_k) \in \mathbb{N}^k \subset \mathbb{N}^*$. Tada je prema propoziciji 3.14,

$$\begin{aligned} \overline{\text{part}}(c, \text{lh}(c)) &= \langle c[0], c[1], \dots, c[\text{lh}(c) - 1] \rangle = \\ &= \langle x_{0+1}, x_{1+1}, \dots, x_{\text{lh}(c)-1+1} \rangle = \langle x_1, x_2, \dots, x_k \rangle = \langle \vec{x} \rangle = c. \end{aligned} \quad (3.19)$$

Za $c = \langle \rangle = 1$, također vrijedi $\overline{\text{part}}(1, \text{lh}(1)) = \overline{\text{part}}(1, 0) = \langle \rangle = 1 = c$.

Sada primitivna rekurzivnost slijedi na uobičajeni način: prema propoziciji 3.14 funkcija part je primitivno rekurzivna, prema lemi 3.15 je tada i $\overline{\text{part}}$ primitivno rekurzivna, a onda je $\chi_{\text{Seq}} = \chi_{=} \circ (I_1^1, \overline{\text{part}} \circ (I_1^1, \text{lh}))$ simbolička definicija karakteristične funkcije od Seq , iz koje se vidi da je ona primitivno rekurzivna. \square

Time smo u potpunosti opisali kodiranje skupa \mathbb{N}^* , koje ćemo kasnije koristiti na brojnim mjestima. Što možemo njime? Rekli smo da nam kodiranje pruža mogućnost rada s kodiranim skupom kao *klasom*, gdje konstruktori i komponente pružaju mogućnost pisanja svih ostalih metoda. Striktno, ulogu konstruktora za \mathbb{N}^* igra familija funkcija Code^k , $k \in \mathbb{N}_+$, dinamizirana kroz operator povijesti, a ulogu komponenta funkcije lh i part . Ideju da sad sve metode klase \mathbb{N}^* možemo napisati pomoću tih funkcija, možemo shvatiti kao da je \mathbb{N}^* *apstraktni tip podataka*, čija konkretna implementacija (umnožak prim-brojeva potenciranih sljedbenicima elemenata niza) nam nije bitna, dok god koristimo $\langle \dots \rangle$ odnosno $\overline{\dots}$, te lh i part prema njihovoj specifikaciji. Pogledajmo jedan jednostavni primjer.

Primjer 3.17. *Konkatenacija* je preslikavanje $\text{concat}: \mathbb{N}^* \times \mathbb{N}^* \rightarrow \mathbb{N}^*$, zadano s

$$\text{concat}((x_1, x_2, \dots, x_k), (y_1, y_2, \dots, y_l)) := (x_1, x_2, \dots, x_k, y_1, y_2, \dots, y_l). \quad (3.20)$$

Pomalo neprecizno, ali sasvim jasno, pišemo $\text{concat}(\vec{x}^k, \vec{y}^l) := (\vec{x}, \vec{y})$. I ubuduće ćemo smatrati da su takvi konstrukti „spljošteni” na jednu razinu — duljina od (\vec{x}, \vec{y}) nije 2, već $k + l$. To smo zapravo već koristili svaki put kad smo napisali npr. $f(\vec{x}, y)$. \triangleleft

Prateća funkcija Nconcat definirana je samo na $\text{Seq} \times \text{Seq}$, ali u svrhu primitivne rekurzivnosti, to se shvaća kao parcijalna specifikacija. Također, njeno primitivno rekurzivno proširenje se često piše infiksno kao operacija.

Lema 3.18. *Postoji primitivno rekurzivna operacija $*$ takva da je za sve $\vec{x}, \vec{y} \in \mathbb{N}^*$,*

$$\langle \vec{x} \rangle * \langle \vec{y} \rangle = \langle \vec{x}, \vec{y} \rangle. \quad (3.21)$$

Dokaz. Zapravo, operator povijesti nam daje mogućnost pisanja „točkovne definicije” željenog konačnog niza: samo kažemo koju duljinu želimo, i zadamo funkciju koja propisuje elemente. Konkretno, ovdje želimo duljinu koja je zbroj duljina od \vec{x} i od

\vec{y} , te (ako sa x i y označimo operande od $*$) prvih $k := \text{lh}(x)$ elemenata trebaju biti dobiveni pomoću `part` iz x , a preostali elementi iz y (s indeksima pomaknutima za k).

Preciznije, definirajmo funkciju G^3 točkovno po slučajevima:

$$G(x, y, i) := \begin{cases} x[i], & i < \text{lh}(x) \\ y[i \ominus \text{lh}(x)], & \text{inače} \end{cases}, \quad (3.22)$$

i pomoću nje

$$x * y := \overline{G}(x, y, \text{lh}(x) + \text{lh}(y)). \quad (3.23)$$

Sada sigurno vrijedi da je $x * y \in \text{Seq}$ (jer je vrijednost funkcije dobivene poviješću), vrijedi $t := \text{lh}(x * y) = \text{lh}(x) + \text{lh}(y)$, te za svaki $i < t$ vrijedi $(x * y)[i] = G(x, y, i)$. Kako kod konačnog niza $\text{concat}(\vec{x}, \vec{y})$ ima sva ta svojstva, kodiranje je injekcija, a konačan niz je jednoznačno zadan svojom duljinom i elementima, zaključujemo da je $x * y$ upravo $\langle \text{concat}(\vec{x}, \vec{y}) \rangle = \langle \vec{x}, \vec{y} \rangle$.

Jednadžbe (3.22) i (3.23) mogu poslužiti kao točkovna definicija od G i $*$ za sve x i y , ne samo za one iz Seq — i iz toga slijedi primitivna rekurzivnost funkcije G po teoremu 2.44 (primitivno rekurzivna verzija), a onda i funkcije \overline{G} po lemi 3.15, pa tako i operacije $*$ dobivene iz nje kompozicijom. \square

Zanimljivo je da smo koristili samo javno sučelje kodiranja konačnih nizova — isti dokaz funkcionira za liste u Pythonu, koje sigurno imaju drugačiju implementaciju:

```
>>> x, y = [2, 5, 8], [0, 3]
>>> [x[i] if i < len(x) else y[i - len(x)] for i in range(len(x) + len(y))]
[2, 5, 8, 0, 3]
```

3.2.1. Primitivna rekurzija kroz prostor i vrijeme

Primitivna rekurzija pruža nam jednostavan način pisanja ograničenih petlji, koje prate jedan podatak. Što ako ih želimo pratiti više, recimo l njih? Rekli smo da ćemo više izlaznih podataka reprezentirati kroz više nezavisnih algoritama, i ponekad se to doista može rastaviti: recimo ako tražimo kumulativni zbroj i umnožak nekog niza, možemo prvo naći zbroj pa onda umnožak, zasebnim primitivnim rekurzijama. No kod kompliciranijih rekurzija, algoritmi više nisu lako odvojivi, jer je zamislivo da sljedeće vrijednosti svih l podataka ovise o prethodnim vrijednostima svih njih.

Važno mjesto gdje se to pojavljuje je simulacija kompliciranih (npr. univerzalnih) modela izračunljivosti, primitivnom rekurzijom kroz vrijeme. Recimo, u RAM-stroju moramo pratiti registre i programski brojač. Stanje registara u koraku $n + 1$ ovisi o stanju registara u koraku n , ali ovisi i o tome koja se instrukcija izvršava, što (za fiksni program) ovisi o vrijednosti programskog brojača. Vrijednost pak programskog brojača u koraku $n + 1$ definirana je po slučajevima, i najčešće je jednostavno sljedbenik

vrijednosti programskog brojača u koraku n , ali kod izvršavanja instrukcije tipa DEC ovisi i o stanju registra na koji ta instrukcija djeluje, u koraku n .

Takva ovisnost zove se *simultana* primitivna rekurzija, jer moramo simultano računati svih l vrijednosti — ali kodiranje \mathbb{N}^* (zapravo samo \mathbb{N}^l) omogućuje nam da to implementiramo pomoću obične primitivne rekurzije. Ideja je jednostavna: umjesto l vrijednosti a_1, a_2, \dots, a_l , pratimo jednu vrijednost $\langle \vec{a} \rangle$. Slikovito, pratimo jedan `struct` s l članova.

Propozicija 3.19. *Neka su $k, l \in \mathbb{N}_+$, te neka su $G_1^k, G_2^k, \dots, G_l^k, H_1^{k+l+1}, H_2^{k+l+1}, \dots, H_l^{k+l+1}$ (primitivno) rekurzivne funkcije. Tada su i $F_1^k, F_2^k, \dots, F_l^k$ zadane s*

$$F_i(\vec{x}, 0) := G_i(\vec{x}), \quad (3.24)$$

$$F_i(\vec{x}, y + 1) := H_i(\vec{x}, y, F_1(\vec{x}, y), F_2(\vec{x}, y), \dots, F_l(\vec{x}, y)), \quad (3.25)$$

za sve $i \in [1..l]$, (primitivno) rekurzivne.

Dokaz. Kao što smo već rekli, cilj nam je primitivnom rekurzijom prvo dobiti funkciju $F := \text{Code}^l \circ (F_1, F_2, \dots, F_l)$, iz koje ćemo onda lako dobiti svaki F_i kompozicijom s funkcijom `part`. Inicijalizacija je jednostavna: iz (3.24) imamo

$$F(\vec{x}, 0) = \langle F_1(\vec{x}, 0), \dots, F_l(\vec{x}, 0) \rangle = \langle G_1(\vec{x}), \dots, G_l(\vec{x}) \rangle =: G(\vec{x}), \quad (3.26)$$

te je $G = \text{Code}^l \circ (G_1, \dots, G_l)$ (primitivno) rekurzivna kao kompozicija takvih. Za korak petlje je definicija dulja, ali zapravo sasvim jednostavna: iz (3.25) je

$$\begin{aligned} F(\vec{x}, y + 1) &= \langle F_1(\vec{x}, y + 1), \dots, F_l(\vec{x}, y + 1) \rangle = \\ &= \langle H_1(\vec{x}, y, F_1(\vec{x}, y), \dots, F_l(\vec{x}, y)), \dots, H_l(\vec{x}, y, F_1(\vec{x}, y), \dots, F_l(\vec{x}, y)) \rangle = \\ &= \langle H_1(\vec{x}, y, F(\vec{x}, y)[0], \dots, F(\vec{x}, y)[l - 1]), \dots, H_l(\vec{x}, y, F(\vec{x}, y)[0], \dots, F(\vec{x}, y)[l - 1]) \rangle \\ &=: H(\vec{x}, y, F(\vec{x}, y)), \end{aligned} \quad (3.27)$$

te je H (primitivno) rekurzivna kao kompozicija funkcija $\text{Code}^l, H_i, \text{part}$, konstanti C_0^{k+1} do C_{l-1}^{k+1} , i koordinatnih projekcija.

Jednakosti (3.26) i (3.27) kažu nam da je $F = G \circ H$, dakle funkcija F je dobivena primitivnom rekurzijom iz (primitivno) rekurzivnih funkcija, pa je i sama (primitivno) rekurzivna.

A onda je za svaki $i \in [1..l]$, funkcija F_i zadana s $F_i(\vec{x}, y) = F(\vec{x}, y)[i - 1]$, simbolički $F_i = \text{part} \circ (F, C_{i-1}^{k+1})$. To znači da su sve F_i dobivene kompozicijom iz (primitivno) rekurzivnih funkcija F, part i konstanti, te su i one sve (primitivno) rekurzivne. \square

Napomena 3.20. Gdje se u točkovnoj definiciji pojavljuje sintaksno isti izraz više puta, uvodit ćemo pokrate koje će nam omogućiti da kompliciranije izraze zapišemo lakše i

preglednije. Recimo, (3.27) bismo mogli zapisati kao

$$H(\vec{x}, y, z) := \langle H_1(\vec{d}), H_2(\vec{d}), \dots, H_l(\vec{d}) \rangle, \quad (3.28)$$

$$\text{uz pokratu } \vec{d} := (\vec{x}, y, z[0], z[1], \dots, z[l-1]). \quad (3.29)$$

Treba napomenuti da je to samo kraći *zapis* za (3.27), ne uvođenje pomoćnih funkcija — jer tada bi \vec{d} kao funkcija trebala imati više izlaznih podataka, te primiti \vec{x} , y i z kao argumente, što bi uništilo dobar dio kratkoće zapisa.

Analogija u programskom jeziku C je korištenje preprocesora (`#define`). Na neki način, uvodimo „makroe” u funkcijski jezik, ali ih nećemo formalizirati jer nam neće biti potrebni tako često, nećemo uopće koristiti makroe s parametrima (koje C preprocesor podržava), a „grafičko” uvrštavanje izraza na određena mjesta u većem izrazu nije pretjerano zahtjevna operacija — samo smanjuje preglednost, koja je zapravo jedina motivacija za uvođenje pokrata. \triangleleft

Dokazali smo da je moguće u primitivnoj rekurziji simultano graditi l funkcija, tako da svaka sljedeća vrijednost ovisi „prostorno” o prethodnim vrijednostima različitih funkcija. Što dobijemo ako pokušamo dinamizirati taj l ? Dobit ćemo funkciju koja ovisi o *povijesti* neke druge funkcije, no svakako je najzanimljiviji slučaj kad ovisi „vremenski” o povijesti same sebe — kad je definirana *rekurzijom s poviješću*, koja može koristiti ne samo neposredno prethodnu vrijednost, nego sve ranije.

Matematički, ako obična primitivna rekurzija odgovara običnom principu matematičke indukcije — gdje u dokazu $\wp(n+1)$ smijemo koristiti $\wp(n)$, ali još moramo zasebno dokazati bazu $\wp(0)$ — tada rekurzija s poviješću odgovara principu *jake* indukcije — gdje u dokazu $\wp(n)$ smijemo koristiti $\wp(m)$ za sve $m < n$, te ne trebamo odvajati bazu kao zasebni slučaj: za $n = 0$ ionako nema pretpostavki $\wp(m)$ koje bismo mogli koristiti.

Propozicija 3.21. *Neka je $k \in \mathbb{N}_+$, te G^k (primitivno) rekurzivna funkcija. Tada je i funkcija F^k , zadana s*

$$F(\vec{x}, y) := G(\vec{x}, \bar{F}(\vec{x}, y)), \quad (3.30)$$

također (primitivno) rekurzivna.

Primijetimo da po Dedekindovom teoremu rekurzije (pogledajte [18, str. 60] za detalje: $\bar{F}(\vec{x}, y)$ ovdje kodira $\varphi|_y$) za svaku totalnu funkciju G^k postoji jedinstvena (totalna) funkcija F^k koja zadovoljava jednadžbu (3.30). Zato u njoj možemo pisati simbol $:=$, odnosno reći da je F *definirana* rekurzijom s poviješću.

Dokaz. Ideja je slična kao u dokazu propozicije 3.19, samo umjesto kodiranja fiksne mjesnosti `Code`¹ imamo dinamički operator povijesti. Dakle, trebamo dobiti \bar{F} primitivnom rekurzijom (degeneriranom u slučaju $k = 1$). Inicijalizacija: svaka povijest počinje kodom praznog niza,

$$\bar{F}(\vec{x}, 0) = \langle \rangle = 1. \quad (3.31)$$

Za korak, moramo izraziti $\bar{F}(\vec{x}, y + 1)$ pomoću \vec{x} , y i $z := \bar{F}(\vec{x}, y)$ — primijetimo da nam ovdje „kontrolna varijabla” zapravo i ne treba, jer y uvijek možemo dobiti kao $\text{lh}(z)$. Kao što smo, primjerice, operator \sum mogli dobiti iteriranjem operacije $+$ (2.68) na početnoj vrijednosti 0 (2.67), tako operator $\overline{\cdot}$ možemo dobiti iteriranjem operacije $*$ na početnoj vrijednosti $\langle \rangle$. Dakle, vrijedi

$$\bar{F}(\vec{x}, y + 1) = \bar{F}(\vec{x}, y) * \langle F(\vec{x}, y) \rangle = \bar{F}(\vec{x}, y) * \langle G(\vec{x}, \bar{F}(\vec{x}, y)) \rangle, \quad (3.32)$$

odnosno

$$H(\vec{x}, y, z) := z * \text{Code}^1(G(\vec{x}, z)). \quad (3.33)$$

Sada je funkcija H (primitivno) rekurzivna prema lemi 3.18 i propoziciji 3.4, pa je i $\bar{F} = C_1^{k-1} \text{ } \text{ } H$ (primitivno) rekurzivna jer je dobivena primitivnom rekurzijom iz (primitivno) rekurzivnih funkcija (za $k = 1$ to je degenerirana primitivna rekurzija $\bar{F} = 1 \text{ } H$, pa je \bar{F} (primitivno) rekurzivna po propoziciji 2.20 odnosno po korolaru 2.34). Prema lemi 3.15, tada je i F (primitivno) rekurzivna. \square

3.2.2. Primjeri korištenja rekurzije s poviješću

Primjer 3.22. Vjerojatno najpoznatija funkcija definirana rekurzijom s poviješću je Fibonaccijev niz:

$$\text{Fib}(n) := n, \text{ za } n < 2; \quad (3.34)$$

$$\text{Fib}(n) := \text{Fib}(n - 1) + \text{Fib}(n - 2), \text{ inače.} \quad (3.35)$$

Dokažimo da je Fib^1 primitivno rekurzivna. Po propoziciji 3.21, dovoljno je naći primitivno rekurzivnu funkciju G koja prima povijest $p := \bar{\text{Fib}}(n)$ (kod prvih n vrijednosti Fibonaccijevog niza), te vraća sljedeću vrijednost $\text{Fib}(n)$. Kao što smo rekli, n uvijek možemo dobiti kao $\text{lh}(p)$. Pomoću njega možemo i napisati pomoćnu funkciju za indeksiranje „s kraja” (koja je i inače prilično korisna; moderni programski jezici često dozvoljavaju indeksiranje s kraja pomoću negativnih indeksa, ali mi nemamo negativne brojeve pa ćemo upotrijebiti drugu funkciju):

$$\text{rpart}(c, i) := \begin{cases} c[\text{lh}(c) \ominus \text{Sc}(i)], & i < \text{lh}(c) \\ 0, & \text{inače} \end{cases}. \quad (3.36)$$

Sada nije teško napisati točkovnu definiciju funkcije G :

$$G(p) := \begin{cases} \text{lh}(p), & \text{lh}(p) < 2 \\ \text{rpart}(p, 0) + \text{rpart}(p, 1), & \text{inače} \end{cases}. \quad (3.37)$$

Prema teoremu o grananju (primitivno rekurzivna verzija), funkcija rpart , pa onda i funkcija G , je primitivno rekurzivna, a tada je i Fib primitivno rekurzivna jer je dobivena rekurzijom s poviješću iz G . \triangleleft

Iako smo rekursiju s poviješću uveli koristeći funkcije, prelaskom na karakteristične funkcije možemo analogni rezultat dobiti za relacije. Ugrubo, ako pri utvrđivanju vrijedi li $R(\vec{x}, n)$ koristimo samo istinitosti $R(\vec{x}, m)$ za $m < n$, i to na neki način koji čuva (primitivnu) rekursivnost, tada je i R (primitivno) rekursivna. Evo jednog važnog primjera, koji će također poslužiti kao uvod u sljedeću točku, pokazivanjem da se mogu kodirati razni objekti, ne samo konačni nizovi prirodnih brojeva.

Primjer 3.23. Kodiramo formule logike sudova, tako da propozicijsku varijablu P_i kodiramo kao $\langle 0, i \rangle$, negaciju $\neg\varphi$ kao $\langle 1, u \rangle$ gdje je u kod od φ , te $(\varphi \rightarrow \psi)$ kao $\langle 2, u, v \rangle$ gdje je u kod od φ , a v kod od ψ . Ostali veznici se mogu dobiti pomoću negacije i kondicionala na dobro poznat način — vidjeti [17]. Recimo, kod varijable P_0 je $\langle 0, 0 \rangle = 2^1 \cdot 3^1 = 6$, a kod formule $(P_0 \rightarrow P_0)$ je

$$\langle 2, 6, 6 \rangle = 2^3 \cdot 3^7 \cdot 5^7 = 1\,366\,875\,000. \quad (3.38)$$

Ovakva vrsta kodiranja, gdje se tip zapisuje na početku kao element nekog početnog komada od \mathbb{N} (tzv. *enum*), a nakon njega ostali podaci ili kodovi (koji odgovaraju *pokazivačima* na podatke kod rekursivno definiranih struktura), česta je u računarstvu. U imperativnim jezicima (Pascal, Ada, ...) obično se koristi pojam *variant record*, a u funkcijskima (Haskell, Scala, ...) pojam *algebraic data type*. Recimo, u Haskellu bi deklaracija tog tipa izgledala ovako:

$$\text{data PF} = \text{PropVar Integer} \mid \text{Not PF} \mid \text{Implies PF PF} \quad (3.39)$$

i reprezentacija elementa takvog tipa u računalnoj memoriji bila bi vrlo slična kodiranju koje smo mi napravili. Još jedan primjer, jednostavniji jer nije zadan rekursivno pa ne treba „pokazivače”, vidjet ćemo na početku sljedeće točke.

Može se vidjeti da je to kodiranje injekcija, jer je kompozicija dvije injekcije: prva se dobije tako da „zamijenimo šiljate zagrade oblina”, pa dobijemo elemente od \mathbb{N}^* , a druga je kodiranje \mathbb{N}^* . Ova druga je injekcija prema propoziciji 3.5, ali zašto je prva injekcija? Ako su dvije formule različitih tipova (recimo, jedna je propozicijska varijabla, a druga negacija), preslikavaju se u konačne nizove s različitim prvim elementom. No ako su istog tipa, zapravo trebamo provesti neku indukciju po složenosti formule da bismo dokazali injektivnost. (Pokušajte — to je dobra vježba.) Uostalom, i sama definicija je rekursivna po izgradnji formula: recimo, u kodiranju $\neg\varphi$ pretpostavljamo da već imamo kod od φ .

Pokušajmo sada dokazati da je slika tog kodiranja (nazovimo je PF) primitivno rekursivna. Karakterističnu funkciju te slike, χ_{PF} , definirat ćemo rekursijom s poviješću. Kao i prije, trebamo naći primitivno rekursivnu funkciju G koja prima povijest $\overline{\chi_{\text{PF}}}(n)$, i vraća je li n kod formule logike sudova. Kako vraća 0 ili 1 (bool), funkciju G možemo dobiti kao karakterističnu funkciju neke relacije. Precizno, $G = \chi_R$, gdje R relacija čija

parcijalna specifikacija glasi: za $p := \overline{\chi_{PF}}(n)$,

$$R(p) \iff „n := lh(p) \text{ je kod neke formule logike sudova}”. \quad (3.40)$$

Dakle, pretpostavimo da imamo n , i razmislimo kako bismo odlučili je li kod neke formule. Očito, to će vrijediti ako i samo ako je ili kod propozicijske varijable, ili kod negacije, ili kod kondicionala. Prvi disjunkt možemo napisati kao $\exists i (n = \langle 0, i \rangle)$, ali to nije dovoljno dobro jer je kvantifikacija neograničena. Možemo li je ograničiti? Svakako. Lako je vidjeti da je svaki element konačnog niza manji od koda tog niza:

$$x_i < x_i + 1 < 2^{x_i+1} = p_0^{x_i+1} \leq p_i^{x_i+1} \leq (\dots) \cdot p_i^{x_i+1} \cdot (\dots) = \langle \dots, x_i, \dots \rangle. \quad (3.41)$$

Dakle, ako postoji takav i , on je sigurno manji od n , te prvi disjunkt možemo napisati u obliku ograničene kvantifikacije $(\exists i < n)(n = \langle 0, i \rangle)$, što je primitivno rekurzivno.

Za drugi disjunkt, opet možemo napisati $\exists u (n = \langle 1, u \rangle \wedge PF(u))$, i kao i prije možemo ograničiti kvantifikaciju na $(\exists u < n)$, ali što ćemo s rekurzivnim $PF(u)$? Izvući ćemo ga iz povijesti p , u kojoj su zapisane sve vrijednosti karakteristične funkcije χ_{PF} na brojevima manjim od n . Dakle, drugi disjunkt je

$$(\exists u < n)(n = \langle 1, u \rangle \wedge p[u] = 1), \quad (3.42)$$

što je primitivno rekurzivno. Analogno bismo dobili i treći disjunkt (s dvije ograničene kvantifikacije), primitivno rekurzivan dvostrukom primjenom propozicije 2.53 (i brojnih drugih rezultata koji pokazuju da je kvantificirana relacija primitivno rekurzivna).

Tada je R primitivno rekurzivna kao disjunkcija tri primitivno rekurzivne relacije (propozicija 2.42), što znači da je njena karakteristična funkcija $G = \chi_R$ primitivno rekurzivna. I za kraj, prema propoziciji 3.21, χ_{PF} je tada primitivno rekurzivna, dakle PF^1 je primitivno rekurzivna relacija. \triangleleft

3.3. Kodiranje RAM-modela izračunljivosti

Napokon imamo dovoljno alata da možemo proći kroz točku 1.3, i sve bitno u njoj kodirati prirodnim brojevima. Tako ćemo dobiti mogućnost simulacije rada RAM-stroja primitivno rekurzivnim funkcijama, a time i parcijalnu rekurzivnost RAM-izračunljivih funkcija. Krenimo redom: prvo su na redu RAM-instrukcije.

Dok još nije dio RAM-programa, instrukcija ima samo tip (INC , DEC ili $GO TO$), te ovisno o tipu, adresu registra na koji djeluje, i/ili odredište (na koje zasad nema nikakvih uvjeta). Posljednje dvoje već jesu prirodni brojevi, a tip instrukcije možemo kodirati na način koji je standardan za konačne skupove: fiksiramo neki poredak. Konkretno, uzet ćemo kodiranje koje preslikava $INC \mapsto 0$, $DEC \mapsto 1$, te $GO TO \mapsto 2$, točno onako kao što bi bio efekt, u programskom jeziku C, naredbe

$$\text{enum ins_type } \{ INC, DEC, GOTO \}; \quad (3.43)$$

te smo time dobili injekciju s $\mathcal{I}ns$ u \mathbb{N}^* , čije kodiranje iskoristimo. Primijetimo da smo sličnu stvar već napravili manje formalno u dokazu leme 1.6 — disjunktifikacija unije skupova A i B u obliku $\{0\} \times A \cup \{1\} \times B$, koju poznajemo iz teorije skupova, upravo odgovara ovakvom kodiranju.

Definicija 3.24. Za proizvoljnu RAM-instrukciju I , definiramo *kod instrukcije* $\lceil I \rceil$, jednadžbama:

$$\lceil INC \mathcal{R}_j \rceil := \langle 0, j \rangle = \text{codeINC}(j) = 6 \cdot 3^j, \quad (3.44)$$

$$\lceil DEC \mathcal{R}_j, l \rceil := \langle 1, j, l \rangle = \text{codeDEC}(j, l) = 60 \cdot 3^j \cdot 5^l, \quad (3.45)$$

$$\lceil GO TO l \rceil := \langle 2, l \rangle = \text{codeGOTO}(l) = 24 \cdot 3^l, \quad (3.46)$$

za sve $j, l \in \mathbb{N}$. ◁

U desnom stupcu napisani su konstruktori kao aritmetički izrazi, iz kojih se vidi kako ih možemo primitivno rekurzivno izračunati iz j i/ili l . Da je $\lceil \dots \rceil: \mathcal{I}ns \rightarrow \mathbb{N}$ injekcija, već smo objasnili. Da mu je slika $Ins := \mathcal{I}ns_{\lceil \dots \rceil}$ primitivno rekurzivna, možemo vidjeti kao u primjeru 3.23. Komponente j odnosno l , te tip, definiramo kroz dvije funkcije i tri relacije.

Lema 3.25. *Skupovi $InsINC$, $InsDEC$ i $InsGOTO$, kodova instrukcija pojedinog tipa, kao i skup svih kodova instrukcija Ins , primitivno su rekurzivni.*

Dokaz. Vidimo

$$InsINC(i) \iff i = \text{codeINC}(i[1]), \quad (3.47)$$

$$InsDEC(i) \iff i = \text{codeDEC}(i[1], i[2]), \quad (3.48)$$

$$InsGOTO(i) \iff i = \text{codeGOTO}(i[1]). \quad (3.49)$$

Dokažimo samo ekvivalenciju (3.47), ostale su sasvim analogne. Za smjer (\Rightarrow), ako je i kod instrukcije tipa INC , recimo $i = \lceil INC \mathcal{R}_j \rceil$, tada po definiciji 3.24 vrijedi $i = \text{codeINC}(j) = \langle 0, j \rangle$, pa je $j = \langle 0, j \rangle[1] = i[1]$.

Za smjer (\Leftarrow), očito je $i = \text{codeINC}(i[1]) = \lceil INC \mathcal{R}_{i[1]} \rceil$ kod instrukcije tipa INC .

Sada primitivna rekurzivnost skupa $Ins = InsINC \cup InsDEC \cup InsGOTO$ slijedi direktno iz propozicije 2.42. □

Lema 3.26. *Definiramo $regn(t)$ kao adresu registra na koji djeluje instrukcija koda t , ako takva postoji i djeluje na neki registar, a 0 inače. Analogno definiramo $dest(t)$ za odredište. Funkcije $regn^1$ i $dest^1$ su primitivno rekurzivne.*

Dokaz. Iz dokaza leme 3.25 odmah se vidi da ako je t kod instrukcije tipa INC , adresa njenog registra je $t[1]$. Ako je tipa DEC , adresa registra je i dalje $t[1]$, a odredište je

$t[2]$. Ako je tipa $GO\ TO$, odredište je $t[1]$. Koristeći tu činjenicu i teorem o grananju (primitivno rekurzivnu verziju), odmah pišemo točkovne definicije:

$$\text{regn}(t) = \begin{cases} t[1], & \text{InsINC}(t) \vee \text{InsDEC}(t) \\ 0, & \text{inače} \end{cases}, \quad (3.50)$$

$$\text{dest}(t) = \begin{cases} t[2], & \text{InsDEC}(t) \\ t[1], & \text{InsGOTO}(t) \\ 0, & \text{inače} \end{cases}. \quad (3.51)$$

Naravno, napisane jednakosti su samo „obrnuto” napisana definicija 3.24. \square

Sad kada imamo kodiranje instrukcija, lako je dobiti i kodiranje programa. Naime, programi su konačni nizovi instrukcija, te ih tako možemo i kodirati.

Definicija 3.27. Za proizvoljni RAM-program $P := \left[t. I_t \right]_{t < n}$ definiramo *kod programa* P kao $\ulcorner P \urcorner := \langle \ulcorner I_0 \urcorner, \ulcorner I_1 \urcorner, \dots, \ulcorner I_{n-1} \urcorner \rangle$. \triangleleft

To je također kodiranje, iako bismo za konstruktore trebali napraviti dinamičke *generatore koda* (jer većina programskih konstrukcija koje smo napravili nemaju fiksnu, statičku duljinu), koji primaju izračunljivu funkciju koja za svaki $i < n$ daje kod instrukcije s rednim brojem i . To se može napraviti sasvim općenito, i vidjeti da su sve konstrukcije programa koje smo dosad sreli (i koje ćemo još sresti u nastavku) primitivno rekurzivne, ali dva su razloga zašto to nećemo raditi.

Prvo, makroi kompliciraju stvar: većinu zanimljivih programa (recimo, one za računanje kompozicije, primitivne rekurzije i minimizacije) nismo napisali kao RAM-programe, nego kao makro-programe. Mogli bismo kodirati makroe kao zaseban tip instrukcija (prirodno se nameće $\ulcorner P^* \urcorner := \langle 3, \ulcorner P \urcorner \rangle$ kao logičan nastavak definicije 3.24) i onda dobiti spljoštenje kao primitivno rekurzivnu funkciju *flat*¹ na kodovima, ali ... postoji i drugi razlog, a taj je da je takvo razmišljanje u potpunoj općenitosti nepotrebno. Kad dokažemo univerzalnost našeg modela, postat će jasno da možemo jednu malu i jednostavnu transformaciju programa — specijalizaciju — napraviti kao primitivno rekurzivnu funkciju, a sve ostale transformacije napraviti pomoću nje.

Ipak, možemo vidjeti jedan jednostavni primjer.

Primjer 3.28. Pri samom početku, u dokazu teorema 1.15, napisali smo RAM-programe P_n (1.10) koji računaju konstantne funkcije. Za svaki n tako možemo dobiti kod tog programa, i to preslikavanje $n \mapsto \ulcorner P_n \urcorner$ je primitivno rekurzivno. Doista, svaka instrukcija u tim programima je $INC\ \mathcal{R}_0$, s kodom $\text{codeINC}(0) = 6$, pa je

$$\ulcorner P_n \urcorner = \langle 6, 6, \dots, 6 \rangle \text{ (n šestica)} = \langle C_6(0), C_6(1), \dots, C_6(n-1) \rangle = \overline{C_6}(n). \quad (3.52)$$

$\overline{C_6}$ je primitivno rekurzivna po propozicijama 2.19 i 3.15. (Kuriozitet: ista se stvar može dobiti i potenciranjem primorijela, $\ulcorner P_n \urcorner = (\overline{Z}(n))^7$; vidite li zašto?) \triangleleft

Primjer 3.29. Evo i jednog statičkog primjera: u primjeru 1.24 naveden je RAM-program Q^b (1.21). Njegov kod je

$$\begin{aligned} \lceil Q^b \rceil &= \langle \lceil \text{DEC } \mathcal{R}_1, 2 \rceil, \lceil \text{GO TO } 0 \rceil, \lceil \text{DEC } \mathcal{R}_2, 2 \rceil, \lceil \text{DEC } \mathcal{R}_1, 6 \rceil, \lceil \text{INC } \mathcal{R}_0 \rceil, \dots, \lceil \text{GO TO } 9 \rceil \rangle \\ &= \langle \langle 1, 1, 2 \rangle, \langle 2, 0 \rangle, \langle 1, 2, 2 \rangle, \langle 1, 1, 6 \rangle, \langle 0, 0 \rangle, \langle 2, 3 \rangle, \langle 1, 2, 9 \rangle, \langle 0, 0 \rangle, \langle 2, 6 \rangle, \langle 1, 3, 12 \rangle, \langle 0, 0 \rangle, \langle 2, 9 \rangle \rangle = \\ &= \langle 4\,500, 24, 13\,500, 2\,812\,500, 6, 648, 1\,054\,687\,500, 6, \dots, 6, 472\,392 \rangle = \\ &= 2^{4501} \cdot 3^{25} \cdot 5^{13501} \dots 31^7 \cdot 37^{472393} =: e_0. \quad (3.53) \end{aligned}$$

Taj kod ćemo koristiti u kasnijim primjerima. \triangleleft

Injektivnost preslikavanja $\lceil \dots \rceil$ slijedi direktno iz injektivnosti od $\lceil \dots \rceil$ i $\langle \dots \rangle$: dva različita programa se razlikuju ili po duljini (pa njihovi kodovi imaju različit *lh*), ili u nekoj instrukciji, recimo onoj na rednom broju i (pa njihovi kodovi imaju različit *part* na mjestu i , jer je kodiranje instrukcija injektivno).

Lema 3.30. *Slika kodiranja RAM-programa, $\text{Prog}^1 := \mathcal{J}_{\lceil \dots \rceil} = \{\lceil P \rceil \mid P \in \text{Prog}\}$, primitivno je rekurzivna.*

Dokaz. Samo treba reći da se radi o konačnom nizu instrukcija, čija odredišta (ako postoje) su manja ili jednaka duljini programa:

$$\text{Prog}(e) \iff \text{Seq}(e) \wedge (\forall i < \text{lh}(e)) (\text{Ins}(e[i]) \wedge \text{dest}(e[i]) \leq \text{lh}(e)). \quad (3.54)$$

Primijetimo samo da koristimo činjenicu da je $\text{dest}(t) = 0$ ako instrukcija koda t nema odredište, te je $0 \leq \text{lh}(e)$ uvijek. Marljiva evaluacija od \wedge (odnosno *mul*²) znači da će se $\text{lh}(e)$ uvijek izračunati, čak i kad e nije kod konačnog niza, ali rezultat će sigurno biti *false* (odnosno 0), a nećemo zapeti u beskonačnoj petlji jer je *lh* totalna. Izračunavanje će biti dulje jer nemamo *shortcircuiting*, ali izračunljivost se neće promijeniti. \square

3.3.1. Kodiranje stanja registara i RAM-konfiguracija

Konfiguraciju RAM-stroja smo definirali kao jednu funkciju, ali zapravo je jasno da trebamo kodirati dvije odvojene stvari: stanje registara, i vrijednost programskog brojača. Vrijednost programskog brojača već jest prirodan broj, pa ga ne kodiramo (odnosno kodiramo ga identitetom). Sa stanjem registara imamo više posla.

Na prvi pogled, skup svih mogućih stanja registara ne možemo uopće kodirati jer je neprebrojiv: imamo prebrojivo mnogo registara, svaki može sadržavati jednu od prebrojivo mnogo vrijednosti, a $\aleph_0^{\aleph_0} = \mathfrak{c} > \aleph_0$. Ipak, definicija 1.8 kaže da promatramo samo one konfiguracije koje su s konačnim nosačem — i to je doista dovoljno. Naime, početna konfiguracija (bilo kojeg RAM-stroja s bilo kojim ulazom) je 0 na svim registrima osim najviše k ulaznih, te se u svakom koraku izračunavanja broj elemenata

nosača može povećati najviše za 1 — izvršavanjem instrukcije $\text{INC } \mathcal{R}_j$ na nekom registru čija je vrijednost prije bila $r_j = 0$.

Iz teorije skupova znamo da nizova prirodnih brojeva s konačnim nosačem ima prebrojivo mnogo, no kako ih kodirati? Jedna elegantna metoda koristi istu ideju kao za konačne nizove, samo bez sljedbenika eksponenata. Dakle, za proizvoljan niz s konačnim nosačem $(r_0, r_1, r_2, \dots, 0, 0, 0, \dots)$ definiramo kod s

$$\|r_0, r_1, r_2, \dots, 0, 0, 0, \dots\| := \prod_{i \in \mathbb{N}} p_i^{r_i}. \quad (3.55)$$

Zbog konačnosti nosača samo konačno mnogo eksponenata je pozitivno (svi ostali su 0), pa samo konačno mnogo prim-brojeva sudjeluje u produktu.

Važno je da takvo kodiranje stanja registara ne ovisi o konkretnom RAM-stroju odnosno programu. Mogli bismo npr. proći kroz program i naći mu širinu m_P , pa kodirati stanje kao konačan niz registara do \mathcal{R}_{m_P} — ali to bi ovisilo o programu: dodavanje potpuno irelevantnih instrukcija koje se možda uopće ne mogu izvršiti (*unreachable code*) bi moglo promijeniti kod stanja registara za isto izračunavanje (isti niz konfiguracija), te kodiranje ne bi bilo funkcija.

Treba nam jedan bitan konstruktor, a to je početna konfiguracija RAM-stroja s ulazom \vec{x} (prenesenim preko koda, jer želimo imati jednu funkciju za sve mjesnosti).

Lema 3.31. *Postoji primitivno rekurzivna funkcija start^1 , s parcijalnom specifikacijom: za svaki $\vec{x} \in \mathbb{N}^+$ (neprazni konačni niz), $\text{start}(\langle \vec{x} \rangle) = \|0, \vec{x}, 0, 0, 0, \dots\|$.*

Dokaz. Samo treba napisati definiciju:

$$\text{start}(x) := \prod_{i=0}^{\text{lh}(x)-1} p_{i+1}^{x[i]} = \prod_{i < \text{lh}(x)} \text{pow}(\text{prime}(\text{Sc}(i)), \text{part}(x, i)). \quad (3.56)$$

Primitivna rekurzivnost slijedi iz propozicija 3.12 i 3.14, napomene 2.51 i primjera 2.11. Prim-brojevi su pomaknuti jer adrese ulaznih registara počinju od 1. \square

Napomenimo da funkcija start nije injekcija: recimo, $\text{start}(\langle 2, 1 \rangle) = \text{start}(\langle 2, 1, 0, 0 \rangle) = \|0, 2, 1, 0, 0, \dots\|$, odnosno $\text{start}(72) = \text{start}(2520) = 45$. Ali preslikavanje $\|\cdot\|$ jest injekcija (kao što kodiranje i treba biti), po osnovnom teoremu aritmetike. Također po osnovnom teoremu aritmetike, slika mu je $\mathcal{J}_{\|\cdot\|} = \mathbb{N}_+$, što smo dokazali primitivno rekurzivnim još davno (primjer 2.26).

Što se komponenata tiče, lh nema smisla, a za indeksiranje služi funkcija ex iz leme 3.13. Doista, $\text{ex}(\|r_0, r_1, \dots, 0, 0, \dots\|, i) = r_i$, jer je to upravo eksponent od p_i u tom kodu. Koristit ćemo je u očitavanju rezultata (sadržaj registra \mathcal{R}_0 u završnoj konfiguraciji).

$$\text{result}(c) := \text{ex}(c, 0) \quad (3.57)$$

Za samo izvršavanje instrukcija na registrima, koristit ćemo množenje odnosno dijeljenje s p_j za inkrement odnosno dekrement registra \mathcal{R}_j . Jasno je da time povećavamo odnosno smanjujemo eksponent odgovarajućeg prim-broja za 1. Konkretno,

$$\|r_0, r_1, \dots, r_{j-1}, r_j + 1, r_{j+1}, r_{j+2}, \dots, 0, 0, \dots\| = \|r_0, r_1, \dots, 0, 0, \dots\| \cdot p_j, \quad (3.58)$$

$$\|r_0, r_1, \dots, r_{j-1}, r_j - 1, r_{j+1}, r_{j+2}, \dots, 0, 0, \dots\| = \|r_0, r_1, \dots, 0, 0, \dots\| // p_j \quad (3.59)$$

(jasno, (3.59) vrijedi samo ako je $r_j > 0$).

Sada možemo kodirati dokaz leme 1.9: za preslikavanje `nextconf` koje konfiguraciju RAM-stroja preslikava u „sljedeću” konfiguraciju (u koju ova prelazi), konstruirat ćemo `Nnextconf` kao izračunljivu funkciju. Ona ima dva izlaza, pa ćemo je reprezentirati kroz dvije primitivno rekurzivne funkcije — koje će primiti trenutnu instrukciju $I_{c(PC)}$ kao da ona uvijek postoji, a završnim konfiguracijama ćemo se baviti kasnije.

Lema 3.32. *Za proizvoljnu RAM-konfiguraciju c , označimo kod stanja njenih registara s $c(\mathcal{R}_*) := \|c(\mathcal{R}_0), c(\mathcal{R}_1), \dots\|$. Tada postoje primitivno rekurzivne funkcije `NextReg`² i `NextCount`³ takve da za RAM-instrukciju I , i za RAM-konfiguracije c i d takve da c nije završna, te $c \rightsquigarrow d$ po instrukciji I , vrijedi*

$$\text{NextReg}(\ulcorner I \urcorner, c(\mathcal{R}_*)) = d(\mathcal{R}_*), \quad i \quad (3.60)$$

$$\text{NextCount}(\ulcorner I \urcorner, c(\mathcal{R}_*), c(PC)) = d(PC). \quad (3.61)$$

Dokaz. Označimo argumente tih funkcija redom s $i := \ulcorner I \urcorner \in \text{Ins}$, $r := c(\mathcal{R}_*) \in \mathbb{N}_+$, te $p := c(PC) \in \mathbb{N}$ (naravno, ovaj treći samo za `NextCount`).

`NextReg` treba pomnožiti ili podijeliti (ako je djeljiv) r s p_j , ako I djeluje na \mathcal{R}_j — ili ga ostaviti na miru, u suprotnom. `NextCount` treba inkrementirati p ako je I promijenila neki registar, ili ga postaviti na njeno odredište ako nije.

$$\text{NextReg}(i, r) := \begin{cases} r \cdot p_j, & \text{Up} \\ r // p_j, & \text{Down} \\ r, & \text{inače} \end{cases} \quad (3.62)$$

$$\text{NextCount}(i, r, p) := \begin{cases} Sc(p), & \text{Up} \vee \text{Down} \\ dest(i), & \text{inače} \end{cases} \quad (3.63)$$

$$\text{uz pokrate } p_j := \text{prime}(\text{regn}(i)) \quad (3.64)$$

$$\text{Up} :\iff \text{InsINC}(i) \quad (3.65)$$

$$\text{Down} :\iff \text{InsDEC}(i) \wedge p_j \mid r \quad (3.66)$$

Uvjeti `Up` i `Down` su disjunktni jer su već `InsINC` i `InsDEC` disjunktni ($i[0]$ ne može istovremeno biti 0 i 1). Također, uvjeti su primitivno rekurzivni, kao i pojedine grane funkcija `NextReg` i `NextCount`, pa su one primitivno rekurzivne po teoremu 2.44.

Dokažimo da **NextReg** i **NextCount** doista zadovoljavaju željenu parcijalnu specifikaciju. U tu svrhu, neka su I i c , odnosno i , r i p kao na početku dokaza. Tvrdimo da **NextReg**(i, r) i **NextCount**(i, r, p) upravo kodiraju konfiguraciju u koju c prelazi po I .

Ako je I tipa **INC**, recimo $I = (\text{INC } \mathcal{R}_j)$, tada vrijedi $\text{Up } i \text{ } p_j = \text{prime}(j) = p_j$, pa je $r' = r \cdot p_j$, što prema (3.58) kodira upravo stanje registara nakon izvršavanja I .

Ako je I tipa **GO TO**, recimo $I = (\text{GO TO } l)$, opet ne vrijedi ni **Up** ni **Down**, te je $r' = r$, a $p' = \text{dest}(i) = \text{dest}(\ulcorner \text{GO TO } l \urcorner) = l$, kao što i treba biti.

Ako je pak I tipa **DEC**, recimo $I = (\text{DEC } \mathcal{R}_j, l)$, tada je opet $p_j = p_j$, te **Up** ne vrijedi, a $\text{Down} \Leftrightarrow p_j \mid r = c(\mathcal{R}_*) \Leftrightarrow c(\mathcal{R}_j) > 0$. Ako je to istina, $p' = p + 1$ i $r' = r // p_j$, a ako nije, $p' = \text{dest}(\ulcorner \text{DEC } \mathcal{R}_j, l \urcorner) = l$ i $r' = r$, kao što i treba biti po semantici instrukcije tipa **DEC**. \square

3.3.2. Kodiranje RAM-izračunavanja

Sada raspoložemo svime potrebnim da bismo kodirali postupak izračunavanja kao niz konfiguracija. Konkretno, neka je P^k RAM-algoritam, te $\vec{x} \in \mathbb{N}^k$ ulaz za njega. Htjeli bismo konstruirati izračunljivu funkciju koja prima $k \in \mathbb{N}_+$, $\vec{x} \in \mathbb{N}^k$ i $P \in \text{Prog}$, te vraća niz $(c_n)_{n \in \mathbb{N}}$ koji predstavlja P -izračunavanje s \vec{x} .

Izrazimo taj zadatak preko brojevnihi funkcija. Umjesto \vec{x} i P očito možemo prenijeti njihove kodove $x := \langle \vec{x} \rangle \in \text{Seq}' := \text{Seq} \setminus \{\langle \rangle\}$ i $e := \ulcorner P \urcorner \in \text{Prog}$. Tada ne treba prenositi k jer ga uvijek možemo odrediti kao $\text{lh}(x)$. No kako vratiti niz? Skup kodova stanja registara je $\mathbb{J}_{\parallel \dots \parallel} = \mathbb{N}_+$, a skup „kodova” vrijednosti programskog brojača je jednostavno \mathbb{N} (zapravo $[0.. \text{lh}(e)]$, ali hoćemo imati jednu kodomenu za sve $e \in \text{Prog}$), dakle skup kodova konfiguracija je $\mathbb{N}_+ \times \mathbb{N}$, a skup kojem pripadaju izračunavanja je onda skup nizova $(\mathbb{N}_+ \times \mathbb{N})^{\mathbb{N}}$. Preslikavanje koje konstruiramo tada je element skupa

$$((\mathbb{N}_+ \times \mathbb{N})^{\mathbb{N}})^{\text{Seq}' \times \text{Prog}} \cong (\mathbb{N}_+ \times \mathbb{N})^{\mathbb{D}} \cong \mathbb{N}_+^{\mathbb{D}} \times \mathbb{N}^{\mathbb{D}}, \quad (3.67)$$

gdje smo označili $\mathbb{D} := \text{Seq}' \times \text{Prog} \times \mathbb{N}$ — dakle zapravo želimo dvije tromjesne funkcije s ulaznim podacima $(x, e, n) \in \mathbb{D}$, gdje je n broj koraka izračunavanja koje smo napravili. Ukratko, nizove kodiramo „točkovno” tako da zapravo kodiramo njihove članove, dodavši indeks u nizu kao još jedan ulazni podatak, a algoritam s dva izlazna podatka (kod stanja registara i vrijednost programskog brojača) već standardno shvaćamo kao dva algoritma s istim ulaznim podacima. Sada je još samo potrebno proširiti te funkcije do totalnih tako da budu primitivno rekurzivne.

Lema 3.33. *Postoje primitivno rekurzivne funkcije Reg^3 i Count^3 , čija parcijalna specifikacija glasi: za svaki RAM-program P , za svaki neprazni konačni niz \vec{x} , za svaki prirodni broj n , $\text{Reg}(\langle \vec{x} \rangle, \ulcorner P \urcorner, n)$ je kod stanja registara, a $\text{Count}(\langle \vec{x} \rangle, \ulcorner P \urcorner, n)$ je vrijednost programskog brojača, nakon n koraka P -izračunavanja s \vec{x} .*

Preciznije, ako je P-izračunavanje s \vec{x} niz $(c_n)_{n \in \mathbb{N}}$, tada je $\text{Reg}(\langle \vec{x} \rangle, \ulcorner P \urcorner, n) = c_n(\mathcal{R}_*)$, a $\text{Count}(\langle \vec{x} \rangle, \ulcorner P \urcorner, n) = c_n(\text{PC})$.

Dokaz. To što smo razdvojili reprezentaciju izračunavanja na dvije funkcije ne znači da ih možemo računati odvojenim algoritmima. Lako je vidjeti da, prema lemi 1.9, sljedeća konfiguracija ovisi samo o neposredno prethodnoj (izračunavanje je *memoryless* — kao Markovljev lanac, samo što je determinističko), ali o oba njena dijela — i o registrima i o brojaču. To znači da je prirodni način definiranja tih funkcija *simultana rekurzija*.

Za nju nam prvo trebaju inicijalizacije, odnosno vrijednosti funkcija u $n = 0$. To je lako: nakon 0 koraka stanje registara je početna konfiguracija s ulazom \vec{x} , čiji se kod može dobiti iz koda $\langle \vec{x} \rangle$ funkcijom **start** iz leme 3.31:

$$\text{Reg}(\langle \vec{x} \rangle, e, 0) = \|0, x_1, x_2, \dots, x_k, 0, 0, \dots\| = \text{start}(\langle \vec{x} \rangle), \quad (3.68)$$

pa za općenitu točkovnu definiciju inicijalizacijske funkcije za **Reg** (gdje je $x \in \mathbb{N}$ proizvoljan) uzmimo

$$\text{Reg}(x, e, 0) := G_1(x, e) := \text{start}(x). \quad (3.69)$$

Inicijalizacija za **Count** je još jednostavnija: $G_2 := C_0^2$, jer je početna vrijednost programskog brojača uvijek 0.

Prelazimo na korak računanja. Za njega nam trebaju funkcije H_1^5 i H_2^5 , koje primaju x i e , broj već napravljenih koraka n , te stare vrijednosti koda stanja registara r i programskog brojača p — a vraćaju nove vrijednosti r' i p' redom. Njih dobijemo pomoću **NextReg** i **NextCount**, s tim da sad moramo voditi računa i o završnim konfiguracijama, i ostaviti ih fiksima. **NextReg** to već čini, jer ako t nije kod instrukcije, (3.62) kaže da je $\text{NextReg}(t, r) = r$ — no za H_2 ćemo morati granati.

$$H_1(x, e, n, r, p) := \text{NextReg}(e[p], r), \quad (3.70)$$

$$H_2(x, e, n, r, p) := \begin{cases} \text{NextCount}(e[p], r, p), & p < \text{lh}(e) \\ p, & \text{inače} \end{cases}. \quad (3.71)$$

Funkcije H_1 i H_2 su primitivno rekurzivne po teoremu o grananju 2.44, propoziciji 3.14, lemi 3.32 i primjeru 2.27. Sada je jasno da su **Reg** i **Count**, definirane s

$$\text{Reg}(x, e, 0) := G_1(x, e) = \text{start}(x), \quad (3.72)$$

$$\text{Count}(x, e, 0) := G_2(x, e) = 0, \quad (3.73)$$

$$\text{Reg}(x, e, n+1) := H_1(x, e, n, \text{Reg}(x, e, n), \text{Count}(x, e, n)), \quad (3.74)$$

$$\text{Count}(x, e, n+1) := H_2(x, e, n, \text{Reg}(x, e, n), \text{Count}(x, e, n)), \quad (3.75)$$

dobivene simultanom primitivnom rekurzijom iz primitivno rekurzivnih funkcija G_1 , G_2 , H_1 i H_2 , te su primitivno rekurzivne po propoziciji 3.19.

Dokažimo da doista ispunjavaju navedenu parcijalnu specifikaciju. U tu svrhu, neka je \vec{x} neprazni konačni niz, P RAM-program, te x i e njihovi kodovi redom. Tvrdnju dokazujemo indukcijom po n , broju koraka. Za $n = 0$, $\text{Count}(x, e, n) = 0$, što je po definiciji vrijednost programskog brojača nakon 0 koraka izračunavanja. Također, iz leme 3.31 slijedi da je $\text{Reg}(x, e, 0)$ kod stanja registara na početku izračunavanja.

Pretpostavimo da je parcijalna specifikacija zadovoljena nakon n koraka, i pogledajmo što se događa u $(n+1)$. koraku. Ako je konfiguracija c_n (nakon n koraka) završna, tada je po pretpostavci indukcije $\text{Count}(x, e, n) = c_n(\text{PC}) = \text{lh}(e)$, te ne vrijedi uvjet grananja u definiciji H_2 , pozvanoj iz (3.75). Također je prema propoziciji 3.14 $e[\text{lh}(e)] = 0 \notin \text{Ins}$, što znači da funkcija H_1 vraća nepromijenjen kod stanja registara (ne vrijedi ni Up ni Down u (3.62)). Iz toga je $\text{Reg}(x, e, n+1) = \text{Reg}(x, e, n)$ i $\text{Count}(x, e, n+1) = \text{Count}(x, e, n)$, što i treba biti jer je $c_{n+1} = c_n$.

Ako pak c_n nije završna, tada je $p := c_n(\text{PC}) < \text{lh}(e)$, te je $i := e[p]$ kod instrukcije koja se izvršava u $(n+1)$. koraku. Tada je po pretpostavci indukcije i lemi 3.32,

$$\begin{aligned} \text{Reg}(x, e, n+1) &= H_1(x, e, n, \text{Reg}(x, e, n), p) = \\ &= \text{NextReg}(e[p], \text{Reg}(x, e, n)) = \text{NextReg}(i, c_n(\mathcal{R}_*)) = c_{n+1}(\mathcal{R}_*), \end{aligned} \quad (3.76)$$

i analogno $\text{Count}(x, e, n+1) = c_{n+1}(\text{PC})$, čime je proveden korak indukcije. \square

3.3.3. Prepoznavanje završne konfiguracije i čitanje rezultata

Pomoću funkcije Count lako je detektirati završnu konfiguraciju.

$$\text{Final}'(x, e, n) :\iff \text{Count}(x, e, n) = \text{lh}(e) \quad (3.77)$$

Zbog $\chi_{\text{Final}'}^3 = \chi_{=} \circ (\text{Count}, \text{lh} \circ \text{l}_2^3)$, to je primitivno rekurzivna relacija, te za sve $P \in \text{Prog}$, $\vec{x} \in \mathbb{N}^+$ i $n \in \mathbb{N}$, $\text{Final}'(\langle \vec{x} \rangle, \ulcorner P \urcorner, n)$ upravo znači da je konfiguracija nakon n koraka P -izračunavanja s \vec{x} , završna.

Ipak, za proizvoljne $(x, e, n) \in \mathbb{N}^3$, može se dogoditi da vrijedi $\text{Final}'(x, e, n)$, iako e uopće nije kod RAM-programa, ili x nije kod nepraznog konačnog niza.

Primjer 3.34. Recimo, za $x = 100$ i $e = 10^{217}$, prvo bismo odredili $\text{start}(100)$. Broj 100 ima dva prim-djelitelja (2 i 5), pa je $\text{lh}(100) = 2$. Iz toga $\text{start}(100) = 3^{100[0]} \cdot 5^{100[1]} = \|0, 1, 0, 0, \dots\| = 3 = \text{Reg}(100, 10^{217}, 0)$. Dakle, \mathcal{R}_1 kreće od 1, a svi ostali registri od nule. $\text{Count}(100, 10^{217}, 0) = 0$ jer PC uvijek kreće od nule.

Sada odredimo „trenutnu instrukciju”: $e[0] = 216 = 2^3 \cdot 3^3 = \langle 2, 2 \rangle = \ulcorner \text{GO TO } 2 \urcorner \in \text{InsGOTO}$, te je $\text{Count}(100, 10^{217}, 1) = \text{dest}(216) = 2 = \text{lh}(e)$. Dakle, $\text{Final}'(100, 10^{217}, 1)$ vrijedi — no nema smisla reći da to reprezentira zaustavljanje nekog izračunavanja, jer ne postoje $P \in \text{Prog}$ i $\vec{x} \in \mathbb{N}^+$ takvi da bi to bilo P -izračunavanje s \vec{x} . \triangleleft

Primjer 3.34 pokazuje da trebamo biti oprezni s parcijalnim specifikacijama. Dok su za primitivno rekurzivne funkcije one ponekad „nužno zlo” ili bar „manje zlo”, za primitivno rekurzivne *relacije* je mnogo prirodnije isključiti sve nespecificirane točke (efektivno, promatrati presjek s parcijalnom specifikacijom). To ćemo učiniti ovdje.

$$\text{Final}(x, e, n) : \Longleftrightarrow \text{Seq}'(x) \wedge \text{Prog}(e) \wedge \text{Final}'(x, e, n) \quad (3.78)$$

Lema 3.35. *Relacija Final^3 je primitivno rekurzivna, te za proizvoljne $(x, e, n) \in \mathbb{N}^3$, vrijedi $\text{Final}(x, e, n)$ ako i samo ako je x kod nekog nepraznog konačnog niza \vec{x} prirodnih brojeva, e je kod nekog RAM-programa P , te P -izračunavanje s \vec{x} stane nakon najviše n koraka.*

Dokaz. Za primitivnu rekurzivnost: jednočlana relacija $\{\langle \rangle\} = \{1\}$ je primitivno rekurzivna po lemi 2.45. Tada su po propoziciji 2.39, $\text{Seq}' := \text{Seq} \setminus \{1\}$, a onda i Final , primitivno rekurzivne.

Za specifikaciju, smjer slijeva nadesno, raspetljavanjem definicije od $\text{Final}(x, e, n)$ vidimo da mora vrijediti:

- $\text{Seq}(x)$, dakle $x = \langle \vec{x} \rangle$ za neki $\vec{x} \in \mathbb{N}^*$, jer je $\text{Seq} = \mathcal{I}_{\langle \dots \rangle}$;
- $x \neq 1 = \langle \rangle$, dakle \vec{x} je neprazan, jer je $\langle \dots \rangle$ injekcija;
- $\text{Prog}(e)$, dakle $e = \lceil P \rceil$ za neki RAM-program P , jer je $\text{Prog} = \mathcal{I}_{\lceil \dots \rceil}$;
- $\text{Count}(x, e, n) = \text{lh}(e)$, što prema lemi 3.33 (koju možemo primijeniti zbog prethodne tri stavke) znači da je n -ta konfiguracija u P -izračunavanju s \vec{x} završna — iz čega slijedi da ono stane, te je prvi indeks završne konfiguracije u izračunavanju manji ili jednak n .

Analogno, zaključivanjem u suprotnom smjeru, dobijemo i drugi smjer tvrdnje. \square

To je u redu ako već imamo (kandidat za) n — ali možemo li *izračunati* n ? Naravno, to je upravo minimizacija: $\text{step} := \mu \text{Final}$ je parcijalno rekurzivna, te je $\text{step}(\langle \vec{x} \rangle, \lceil P \rceil) = \mu n \text{Final}(\langle \vec{x} \rangle, \lceil P \rceil, n)$ upravo broj koraka P -izračunavanja s \vec{x} , do dolaska u završnu konfiguraciju. To je definirano samo za izračunavanja koja stanu: $\text{HALT} := \mathcal{D}_{\text{step}} = \exists_* \text{Final}$ je dvomjesna relacija takva da $\text{HALT}(\langle \vec{x} \rangle, \lceil P \rceil)$ vrijedi ako i samo ako P -izračunavanje s \vec{x} stane.

Napomena 3.36. Štoviše, zahvaljujući oprezu sa specifikacijom prije, sad možemo biti precizniji: kad god e nije kod nekog programa, ili x nije kod nepraznog konačnog niza, ne vrijedi $\text{Final}(x, e, n)$, ni za koji n . Dakle, tada $(x, e) \notin \text{HALT}$, te izraz $\text{step}(x, e)$ nema smisla. \triangleleft

Primijetimo da HALT nismo napisali u „izračunljivom fontu”, jer nemamo algoritam za računanje χ_{HALT} . To što je HALT domena parcijalno rekurzivne funkcije, odnosno projekcija primitivno rekurzivne relacije, nije dovoljno: ako je $(x, e) \in \text{HALT}$, to ćemo doznati čim nađemo broj koraka n — ali ako $(x, e) \notin \text{HALT}$, to nećemo nikada doznati tim pristupom. Pokazat ćemo, štoviše, da **nijedan pristup ne radi za sve parove** (x, e) , iako u nekim slučajevima (poput $e \notin \text{Prog}$) možemo utvrditi $\neg \text{HALT}(x, e)$. Precizno, pokazat ćemo da skup HALT nije rekurzivan — što će biti jedan od prvih primjera nepostojanja algoritma za neki problem. No to će morati još malo pričekati.

Pozabavimo se sada čitanjem rezultata izračunavanja (izlaznog podatka algoritma, odnosno vrijednosti funkcije koja se računa na ulaznim podacima). Za $\text{HALT}(x, e)$, izraz $\text{Reg}(x, e, \text{step}(x, e))$ ima smisla, i tada predstavlja stanje registara u završnoj konfiguraciji izračunavanja. Za $\neg \text{HALT}(x, e)$, što uključuje i $\neg \text{Prog}(e)$ i $\neg \text{Seq}'(x)$, taj izraz je nedefiniran. Još treba dokomponirati primitivno rekurzivnu funkciju result iz (3.57), i dobiti

$$\text{univ}(x, e) := \text{result}(\text{Reg}(x, e, \text{step}(x, e))), \quad (3.79)$$

univerzalnu funkciju koja preslikava RAM-program i ulaz za njega u rezultat izračunavanja, ako i samo ako je taj rezultat definiran.

Lema 3.37. *Funkcija univ^2 je parcijalno rekurzivna, i za sve $x, e \in \mathbb{N}$ vrijedi:*

1. *Ako je x kod nekog nepraznog konačnog niza \vec{x} , ako je e kod nekog RAM-programa P , te ako P -izračunavanje s \vec{x} stane, tada je $\text{univ}(x, e)$ rezultat tog izračunavanja.*
2. *U svim ostalim slučajevima, $(x, e) \notin \mathcal{D}_{\text{univ}}$.*

Dokaz. Parcijalna rekurzivnost slijedi direktno iz (3.79), iz primitivne rekurzivnosti funkcija $\text{result} = \text{ex} \circ (I_1^1, Z)$ i Reg , te parcijalne rekurzivnosti funkcije step .

Za tvrdnju 1, pretpostavke znače da postoji konfiguracija u P -izračunavanju s \vec{x} koja je završna. Prema lemi 3.35, to znači $\exists n \text{ Final}(x, e, n)$, odnosno $(x, e) \in \exists_* \text{Final} = \mathcal{D}_{\text{step}}$, pa postoji $n_0 := \text{step}(x, e)$, i $\text{Reg}(x, e, n_0)$ je kod stanja registara u završnoj konfiguraciji c_{n_0} tog izračunavanja.

$$z := \text{Reg}(x, e, n_0) = c_{n_0}(\mathcal{R}_*) = \|c_{n_0}(\mathcal{R}_0), c_{n_0}(\mathcal{R}_1), \dots\| \quad (3.80)$$

Sada je $\text{univ}(x, e) = \text{result}(z) = \text{ex}(\|c_{n_0}(\mathcal{R}_0), \dots\|, 0) = c_{n_0}(\mathcal{R}_0)$, sadržaj registra \mathcal{R}_0 u završnoj konfiguraciji, što smo i trebali.

Za tvrdnju 2, ako neki od uvjeta nije zadovoljen, prema lemi 3.35 (drugi smjer) ni za koji n ne vrijedi $\text{Final}(x, e, n)$, te zato $(x, e) \notin \mathcal{D}_{\text{step}}$. Po definiciji domene kompozicije (marljiva evaluacija), (x, e) ne može biti niti u domeni od univ . \square

3.4. Kleenejev teorem o normalnoj formi

Funkcija `univ` je univerzalna, jer može simulirati bilo koji RAM-stroj, odnosno računati bilo koju RAM-izračunljivu funkciju, pa time (po teoremu 2.36) i svaku parcijalno rekurzivnu funkciju. Na neki način, `univ` predstavlja funkcijski *interpreter* za RAM-stroj, nasuprot imperativnom *kompajleru* za simboličke definicije izgrađenom u točki 2.3.1. No iz raznih razloga, uglavnom tehničkih i povijesnih, zapravo se univerzalna funkcija uvodi na malo drugačiji način.

Htjeli bismo napisati što jednostavniju „simboličku definiciju” univerzalne funkcije. Definicija (3.79) ima nezgodno svojstvo da dvaput koristi x i e , odnosno napisana je kao kompozicija dvije funkcije, svaka od kojih ovisi o ulaznim podacima. To je očito prekomplicirano: vanjska funkcija ne mora primati x i e , ako joj unutarnja funkcija jednostavno pošalje završnu konfiguraciju, ili nešto iz čega se završna konfiguracija može odrediti. Razlog zašto ih trenutno prima je što joj unutarnja funkcija pošalje samo broj koraka, koji je očito sam po sebi nedovoljan za određivanje završne konfiguracije. No koristeći kodiranje \mathbb{N}^* , poslana vrijednost može biti proizvoljno komplicirana.

Drugo, očito od osnovnih operatora (\circ , \mathbb{R} i μ) moramo koristiti minimizaciju jer `univ` nije totalna — ali htjeli bismo je koristiti što „kasnije”, tako da što veći dio stabla koje predstavlja njenu simboličku definiciju bude primitivno rekurzivan. Nije teško pokazati, tehnikama svođenja iz poglavlja 5, da se ne može postići da minimizacija bude u korijenu (ne postoji rekurzivna relacija R^3 takva da je $\text{univ} = \mu R$) — dakle moramo nakon μ primijeniti još neki operator. To ne može biti \mathbb{R} jer bismo time opet dobili totalnu funkciju, dakle mora biti kompozicija. Najjednostavnija „normalna forma” koja zadovoljava te uvjete je $\text{univ}^2 = U^1 \circ \mu \hat{T}^3$ za primitivno rekurzivne U i \hat{T} (U je funkcija, \hat{T} je relacija).

Najjednostavniji način da funkciji U pošaljemo dovoljno podataka o izračunavanju je da joj pošaljemo *čitavo* izračunavanje. Možemo li ga kodirati prirodnim brojem? Ako ne stane, teško: može se ponavljati ciklički, ali se može i ponašati vrlo komplicirano. Ali izračunavanja koja ne stanu ionako nas ne zanimaju, jer ne želimo da $U \circ \mu \hat{T}$ bude definirano u tom slučaju.

Dakle, promotrimo izračunavanje koje stane. Vidjeli smo da ono mora biti oblika $(c_0, c_1, \dots, c_{n_0}, c_{n_0}, c_{n_0}, \dots)$, gdje je c_{n_0} završna, a nijedna c_i za $i < n_0$ nije završna. Za potrebe kodiranja, dovoljno je gledati samo konačan niz $(c_0, c_1, \dots, c_{n_0})$ duljine $n_0 + 1$, jer se iz njega beskonačnim ponavljanjem zadnjeg elementa može dobiti i čitavo izračunavanje. Za kodirati pojedinu c_i , trebali bismo poslati i $\text{Reg}(x, e, i)$ i $\text{Count}(x, e, i)$, no s obzirom na to da nam je u završnoj konfiguraciji po definiciji poznata ova druga vrijednost ($c_{n_0}(\text{PC}) = \text{lh}(e)$) i zapravo nam treba sadržaj registra \mathcal{R}_0 , slat ćemo samo vrijednosti funkcije Reg . Drugim riječima, treba nam povijest $\overline{\text{Reg}}(x, e, n_0 + 1)$, što je izračunljivo jednom kad imamo $n_0 := \text{step}(x, e)$.

Definicija 3.38. Neka je P^k RAM-algoritam, te $\vec{x} \in \mathbb{N}^k$ takav da P -izračunavanje s \vec{x} stane. Kod tog izračunavanja definiramo kao povijest kodova stanja registara, do uključivo prvog indeksa završne konfiguracije u tom izračunavanju. Za izračunavanja koja ne stanu kod nije definiran. \triangleleft

Propozicija 1.11, restringirana samo na izračunavanja koja stanu (jer jedino takva znamo kodirati), može se iskazati kao: tromjesna relacija

$$\text{Trace}(\vec{x}, P, (c_n)_n) : \Longleftrightarrow \text{„}(c_n)_n \text{ je } P\text{-izračunavanje s } \vec{x}, \text{ koje stane} \text{”}, \quad (3.81)$$

ima funkcijsko svojstvo. Tada će i $\hat{T}^3 := \mathbb{N}\text{Trace}$ imati funkcijsko svojstvo. Cilj nam je dokazati da je Trace izračunljiva, odnosno da je \hat{T} primitivno rekurzivna.

U skladu s napomenom u primjeru 3.17, Trace ćemo shvatiti kao $(k+2)$ -mjesnu relaciju, tako da svaki x_i shvatimo kao zasebni argument. To vodi na promatranje familije brojevnih relacija T_k^{k+2} , $k \in \mathbb{N}_+$. A jednom kad dobijemo primitivnu rekurzivnost od \hat{T} , dobit ćemo i primitivnu rekurzivnost svih T_k , jer vrijedi

$$T_k(\vec{x}, e, y) \Longleftrightarrow \hat{T}(\langle \vec{x} \rangle, e, y), \quad (3.82)$$

dakle T_k je dobivena iz \hat{T} kompozicijom s Code^k i koordinatnim projekcijama.

Primjer 3.39. U primjeru 1.21 je naveden primjer makro-programa Q i Q -izračunavanja s $(2, 4)$, koje stane. Kako Q^b -izračunavanje s $(2, 4)$ prolazi kroz ista stanja registara, možemo iz (1.18) izračunati kod tog izračunavanja (iz definicije 1.19 trebamo zanemariti prijelaze tipa 4, jer oni ne odgovaraju nikakvim prijelazima RAM-stroja, kao i one tipa 1, jer smo već stigli do završne konfiguracije):

$$\begin{aligned} c_0 &:= \langle 5625, 1875, 1875, 625, 625, 625, 125, 125, 25, 50, 50, 10, 20, 20, 4, 8, 8, 8, 8 \rangle = \\ &= 2^{5626} \cdot 3^{1876} \cdot 5^{1876} \cdot 7^{626} \cdot 11^{626} \cdot 13^{626} \cdot 17^{126} \dots 61^9 \cdot 67^9. \end{aligned} \quad (3.83)$$

Recimo, $c_0[6] = \text{part}(c_0, 6) = \text{pd}(\text{ex}(c_0, 6)) = \text{pd}(126) = 125 = 5^3 = \|0, 0, 3, 0, \dots\|$, jer nakon 6 koraka Q^b -izračunavanja s $(2, 4)$ u \mathcal{R}_2 bude broj 3, a u svim ostalim registrima broj 0.

Drugi način za iskazati to isto je $\text{Reg}(\langle 2, 4 \rangle, \lceil Q^b \rceil, 6) = \|0, 0, 3, 0, \dots\|$. Ako izračunamo $\langle 2, 4 \rangle = 2^3 \cdot 3^5 = 1944$ i upotrijebimo $e_0 := \lceil Q^b \rceil$ iz primjera 3.29, možemo napisati i $\text{Reg}(1944, e_0, 6) = 5^3$. Vidimo da je c_0 povijest funkcije Reg , konkretno $\hat{T}(1944, e_0, c_0)$ znači $c_0 = \overline{\text{Reg}}(1944, e_0, 19)$, gdje je $\text{pd}(19) = 18 = \text{step}(1944, e_0)$. \triangleleft

Funkcija step jest izračunljiva, ali budući da nam treba *relacija* za minimizaciju, koristit ćemo njen graf $\text{Step}^3 := \mathcal{G}_{\text{step}}$ kao relaciju iz koje možemo dobiti njene vrijednosti.

Napomena 3.40. Primijetimo da ne možemo napisati $n = \text{step}(x, e)$ kao točkovnu definiciju relacije Step — to bi simbolički glasilo $\chi_{\text{Step}} = \chi_{=} \circ (l_3^3, \text{step} \circ (l_1^3, l_2^3))$, što

jednostavno nije istina jer te dvije funkcije imaju različite domene: lijeva je totalna, a desna je definirana samo na $\text{HALT} \times \mathbb{N}$. To je samo jedan od problema koje imamo s parcijalnim funkcijama, koji su osnovni razlog zašto se držimo primitivno rekurzivnih funkcija dok god možemo: vidjet ćemo kasnije da općenito graf (baš kao ni domena) izračunljive funkcije ne mora biti izračunljiv. \triangleleft

Ipak, za totalne funkcije takav rezultat vrijedi, i to već sada možemo dokazati. Prvo strogo definirajmo graf, i dokažimo jednu tehničku lemu.

Definicija 3.41. Neka je $k \in \mathbb{N}_+$ i f^k funkcija. *Graf* funkcije f je relacija \mathcal{G}_f^{k+1} zadana s

$$\mathcal{G}_f(\vec{x}, y) :\iff \vec{x} \in \mathcal{D}_f \wedge y = f(\vec{x}). \quad (3.84)$$

Zapravo, uvjet $\vec{x} \in \mathcal{D}_f$ ne treba pisati jer slijedi iz $y = f(\vec{x})$, ali je naveden zbog jasnoće (pogledajte problem koji smo maloprije imali sa $\mathcal{G}_{\text{step}}$). \triangleleft

Napomena 3.42. Iz elementarne matematike znamo da je relacija R graf neke funkcije ako i samo ako ima *funkcijsko svojstvo*: $x R y_1 \wedge x R y_2 \Rightarrow y_1 = y_2$. Riječima, „svaka vertikala siječe graf u najviše jednoj točki”. \triangleleft

Lema 3.43. Za svaki $k \in \mathbb{N}_+$, za svaku funkciju f^k , vrijede sljedeće jednakosti:

$$\exists_* \mathcal{G}_f = \mathcal{D}_f, \quad (3.85)$$

$$\mu \mathcal{G}_f = f. \quad (3.86)$$

Dokaz. Za (3.85), iz $\vec{x} \in \exists_* \mathcal{G}_f$ slijedi da postoji $y \in \mathbb{N}$ takav da je $\vec{x} \in \mathcal{D}_f$ i $y = f(\vec{x})$. Specijalno to znači $\vec{x} \in \mathcal{D}_f$. U drugom smjeru, $\vec{x} \in \mathcal{D}_f$ znači da postoji $f(\vec{x}) \in \mathbb{N}$, a onda vrijedi i $\mathcal{G}_f(\vec{x}, f(\vec{x}))$, pa je $\vec{x} \in \exists_* \mathcal{G}_f$.

Dokažimo sada (3.86). Iz (3.85) slijedi da te dvije funkcije imaju istu domenu (2.33), trebamo samo vidjeti da se podudaraju u svim točkama te domene. U tu svrhu, neka je $\vec{x} \in \mathcal{D}_f$. Tada postoji $f(\vec{x}) =: y_0 \in \mathbb{N}$, i vrijedi $\mathcal{G}_f(\vec{x}, y_0)$. Štoviše, zbog funkcijskog svojstva, ni za koji drugi $y \neq y_0$ ne vrijedi $\mathcal{G}_f(\vec{x}, y)$, dakle skup $\{y \in \mathbb{N} \mid \mathcal{G}_f(\vec{x}, y)\}$ je jednočlan skup $\{y_0\}$, pa mu je najmanji element $\mu y \mathcal{G}_f(\vec{x}, y) = y_0 = f(\vec{x})$. \square

Teorem 3.44 (Teorem o grafu za totalne funkcije). Neka je $k \in \mathbb{N}_+$, te f^k totalna funkcija. Tada je \mathcal{G}_f rekurzivan ako i samo ako je f rekurzivna.

Dokaz. Za smjer (\Rightarrow): po pretpostavci, $\chi_{\mathcal{G}_f}$ je rekurzivna, dakle parcijalno rekurzivna. Skup parcijalno rekurzivnih funkcija je zatvoren na minimizaciju, pa je $\mu \mathcal{G}_f$ također parcijalno rekurzivna — no ta funkcija je jednaka f po (3.86). Dakle, f je parcijalno rekurzivna, a po pretpostavci teorema je totalna, pa je rekurzivna.

Za smjer (\Leftarrow): kako je f totalna, $\vec{x}^k \in \mathcal{D}_f = \mathbb{N}^k$ uvijek vrijedi, pa (3.84) postaje $\mathcal{G}_f(\vec{x}, y) \iff y = f(\vec{x})$, odnosno $\chi_{\mathcal{G}_f}$ je dobivena kompozicijom iz $\chi_=$, f i koordinatnih projekcija. $\chi_=$ i koordinatne projekcije su rekurzivne po korolarima 2.40 i 2.33, a f je rekurzivna po pretpostavci, pa je $\chi_{\mathcal{G}_f}$ rekurzivna po lemi 2.31. \square

Napomenimo samo da u teoremu 3.44 ne možemo staviti riječ „primitivno” u zagrade, kao što smo činili u mnogim rezultatima do sada: postoje rekurzivne funkcije čiji grafovi su primitivno rekurzivni, ali one same nisu primitivno rekurzivne. *Ackermannova* funkcija, uvedena u [16, dodatak], primjer je takve funkcije.

3.4.1. Univerzalne funkcije comp_k

Kako step nije totalna, ne možemo direktno primijeniti teorem 3.44, ali možemo neke druge rezultate. Konkretno, prema (3.86), imamo $\text{step} = \mu \text{Step}$, no step je već definirana minimizacijom primitivno rekurzivne relacije Final . Relacije Step i Final nisu jednake jer jedna ima funkcijsko svojstvo a druga nema (čim vrijedi $\text{Final}(x, e, n_0)$, vrijedi i $\text{Final}(x, e, n)$ za sve $n > n_0$), ali možemo li dobiti jednu pomoću druge? Svakako, $\text{Final}(x, e, n) \Leftrightarrow (\exists m \leq n) \text{Step}(x, e, m)$ znači da je Final dobivena ograničenom egzistencijalnom kvantifikacijom iz Step , ali nama treba drugi smjer.

Relacija $\text{Step}(x, e, m) \Leftrightarrow m = \mu n \text{Final}(x, e, n)$, koju dobijemo direktnim čitanjem definicije $\text{Step} = \mathcal{G}_{\text{step}}$, čini se kao dobar početak: jedino što joj nedostaje je totalnost ove minimizacije na desnoj strani. No taj problem smo već imali nekoliko puta u točki 3.1.2, i uvijek smo ga uspješno rješavali ograničavanjem minimizacije. Postoji li gornja granica za n do koje je dovoljno provjeravati vrijedi li $\text{Final}(x, e, n)$, da bismo znali je li m najmanji takav n ? Naravno — to je upravo $m + 1$! Odnosno, dovoljno je provjeravati do uključivo m .

Lema 3.45. *Relacija $\text{Step}^3 := \mathcal{G}_{\text{step}^2}$ je primitivno rekurzivna.*

Dokaz. Kao što smo upravo rekli, cilj nam je dokazati

$$\text{Step}(x, e, m) \iff m = (\mu n \leq m) \text{Final}(x, e, n) \quad (3.87)$$

— iz toga će onda slijediti primitivna rekurzivnost prema (redom) lemi 3.35, propoziciji 2.57, napomeni 2.58 i korolaru 2.40.

Pretpostavimo da vrijedi $\text{Step}(x, e, m)$. Tada vrijedi $(x, e) \in \mathcal{D}_{\text{step}} = \text{HALT}$, i $m = \text{step}(x, e)$. Dakle vrijedi $\text{Final}(x, e, m)$, i ni za koji $n < m$ ne vrijedi $\text{Final}(x, e, n)$, te je $\mu n (n \leq m \rightarrow \text{Final}(x, e, n)) = \mu n \text{Final}(x, e, n) = \text{step}(x, e) = m$.

Ako pak ne vrijedi $\text{Step}(x, e, m)$, tada negiranjem (3.84) vidimo da ili ne vrijedi $\text{HALT}(x, e)$, ili pak postoji $s := \text{step}(x, e)$, ali je različit (veći ili manji) od m . Tvrdimo da ni u kojem od tih slučajeva broj $t := (\mu n \leq m) \text{Final}(x, e, n)$ nije jednak m .

Ako $\neg \text{HALT}(x, e)$, tada ne postoji n takav da vrijedi $\text{Final}(x, e, n)$, pa je

$$t = \mu n (n \leq m \rightarrow \perp) = \mu n \neg(n \leq m) = \mu n (n > m) = m + 1 \neq m. \quad (3.88)$$

Ako je $s < m$, tada je $t = s$, pa je opet $t \neq m$. Ako je $s > m$, tada (po definiciji funkcije step) za svaki $n < s$ — pa specijalno za svaki $n \leq m$ — vrijedi $\neg \text{Final}(x, e, n)$, te je opet $t = m + 1 \neq m$. \square

Sada se napokon možemo pozabaviti relacijama \hat{T} i T_k , $k \in \mathbb{N}_+$. Prisjetimo se, one su dobivene kodiranjem argumenata relacije Trace, prva kodirajući ulazne podatke \vec{x} kao element od \mathbb{N}^* , a druga gledajući ih zasebno.

Propozicija 3.46. *Za svaki $k \in \mathbb{N}_+$, relacija T_k , zadana s*

$$T_k(\vec{x}^k, e, y) : \Longleftrightarrow (\exists P \in \text{Prog})(e = \ulcorner P \urcorner \wedge „y je kod P-izračunavanja s \vec{x} ”), \quad (3.89)$$

primitivno je rekurzivna.

Dokaz. Kao što smo već rekli, prvo ćemo dokazati primitivnu rekurzivnost relacije \hat{T} , zadane s

$$\hat{T}(x, e, y) : \Longleftrightarrow (\exists \vec{x} \in \mathbb{N}^+)(x = \langle \vec{x} \rangle \wedge T_{lh(x)}(\vec{x}, e, y)). \quad (3.90)$$

Tvrdimo da je njena točkovna definicija

$$\hat{T}(x, e, y) \Longleftrightarrow \text{Step}(x, e, n) \wedge y = \overline{\text{Reg}}(x, e, \text{Sc}(n)), \quad (3.91)$$

$$\text{uz pokratu } n := \text{pd}(\text{lh}(y)). \quad (3.92)$$

U jednom smjeru, pretpostavimo da vrijedi $\hat{T}(x, e, y)$. Tada prema (3.90) i (3.89) postoje $\vec{x} \in \mathbb{N}^+$ (njegovu duljinu označimo s k) i $P \in \text{Prog}$ takvi da je x kod od \vec{x} , e je kod od P , a y je kod P -izračunavanja s \vec{x} . Po definiciji 3.38, to znači da P -izračunavanje s \vec{x} stane (inače kod ne bi bio definiran) — odnosno vrijedi $\text{HALT}(x, e)$, pa postoji $n_0 := \text{step}(x, e)$ — i y je upravo povijest stanja registara prvih $n_0 + 1$ konfiguracija u tom izračunavanju, od c_0 do uključivo c_{n_0} .

Iz $n_0 = \text{step}(x, e)$ slijedi $\text{Step}(x, e, n_0)$, povijest stanja registara opisana je funkcijom $\overline{\text{Reg}}$, a upravo smo vidjeli da je $\text{lh}(y) = \text{Sc}(n_0)$. Dakle, $n_0 = \text{pd}(\text{lh}(y))$, što se upravo tvrdi u točkovnoj definiciji.

U drugom smjeru, pretpostavimo da vrijedi točkovna definicija. Tada iz $(x, e, n) \in \text{Step} = \mathcal{G}_{\text{step}}$ slijedi $(x, e) \in \mathcal{D}_{\text{step}} = \text{HALT}$ i $n := \text{pd}(\text{lh}(y)) = \text{step}(x, e) = (\mu \text{Final})(x, e)$. Specijalno vrijedi $\text{Final}(x, e, n)$, pa je (po lemi 3.35) x kod nekog nepraznog konačnog niza \vec{x} , e je kod nekog RAM-programa P , te P -izračunavanje s \vec{x} stane nakon najviše n koraka — zapravo u ovom slučaju nakon točno n koraka, jer je $n = \text{step}(x, e)$.

Također vrijedi $y = \overline{\text{Reg}}(x, e, \text{Sc}(n))$, dakle $\text{lh}(y) = \text{Sc}(n) > 0$. To znači da je y povijest stanja registara prvih $n + 1$ konfiguracija, što je upravo kod tog izračunavanja.

Primijetimo još samo da nismo mogli zapisati jednostavnije $y = \text{Reg}(x, e, \text{lh}(y))$, jer bi to zadovoljavao i kod praznog niza: $1 = \overline{G}(\vec{x}, \text{lh}(1))$ bez obzira na specifikaciju funkcije G i vrijednosti argumenata \vec{x} . Na neki način, 0 je problematična kao $m = \text{lh}(y)$, pa smo umjesto m napisali $\text{Sc}(\text{pd}(m)) = m$ u duhu napomene 2.24.

Sada za proizvoljni k , primitivna rekurzivnost T_k slijedi iz (3.82). \square

Korolar 3.47. *Za svaki $k \in \mathbb{N}_+$, relacija T_k ima funkcijsko svojstvo, te je njena projekcija $\exists_* T_k = \{(\vec{x}, e) \in \mathbb{N}^{k+1} \mid \text{HALT}(\langle \vec{x} \rangle, e)\} =: \text{Halt}_k$.*

Dokaz. Direktno iz definicije: za proizvoljne \vec{x} i e , postoji najviše jedan RAM-program P s kodom e ($\lceil \cdot \rceil$ je injekcija), pa onda postoji jedinstveno P -izračunavanje s \vec{x} (propozicija 1.11), pa ako ono stane, postoji jedinstven njegov kod. Ako P ne postoji, ili ako izračunavanje ne stane, ne postoji nijedan y takav da vrijedi $T_k(\vec{x}, e, y)$. Dakle, uvijek postoji *najviše* jedan takav y , te postoji *točno* jedan takav y ako i samo ako je $(\vec{x}, e) \in \text{Halt}_k$ — a to je upravo tvrdnja koju smo željeli dokazati. \square

Korolar 3.47 prema napomeni 3.42 kaže da je za svaki pozitivni k , relacija T_k graf neke funkcije. Štoviše, po lemi 3.43, ta funkcija je upravo μT_k , njena domena je Halt_k , i ona preslikava svaki $(\vec{x}^k, \lceil P \rceil) \in \text{Halt}_k$ (svaki element od Halt_k mora biti tog oblika, po napomeni 3.36) u kod P -izračunavanja s \vec{x} . Sada, da bismo dobili *rezultat* tog izračunavanja, samo treba dokomponirati slijeva funkciju zadanu s

$$U(y) := \text{result}(\text{rpart}(y, 0)) = \text{ex}(y[\text{pd}(\text{lh}(y))], 0), \quad (3.93)$$

doslovno, „sadržaj izlaznog registra zadnje konfiguracije kodirane s y ”, te ćemo dobiti

$$\text{comp}_k(\vec{x}, e) := U(\mu y T_k(\vec{x}, e, y)). \quad (3.94)$$

Propozicija 3.48. *Funkcija U je primitivno rekurzivna. Za svaki $k \in \mathbb{N}_+$, funkcija comp_k je parcijalno rekurzivna, s domenom $\mathcal{D}_{\text{comp}_k} = \text{Halt}_k$, te vrijedi*

$$\text{univ}(\langle \vec{x} \rangle, e) \simeq \text{comp}_k(\vec{x}, e). \quad (3.95)$$

Dokaz. Prvo, $U = \text{result} \circ \text{rpart} \circ (I_1^1, Z)$ je simbolička definicija od U : result je točkovno definirana u (3.57), a rpart u (3.36), pomoću primitivno rekurzivnih funkcija (ex , part , pd i lh), pa su result i rpart — a onda i U — primitivno rekurzivne.

Sada je $\text{comp}_k = U \circ \mu T_k$ parcijalno rekurzivna jer je dobivena kompozicijom i minimizacijom iz primitivno rekurzivnih U i T_k . Prema korolaru 3.47 domena joj je Halt_k , baš kao i domena funkcije $(\vec{x}, e) \mapsto \text{univ}(\langle \vec{x} \rangle, e)$ (po definiciji domene kompozicije, uzevši u obzir da je Code^k totalna). Još treba samo vidjeti da se te dvije funkcije podudaraju na toj domeni. Za svaki $(\vec{x}, e) \in \text{Halt}_k$ vrijedi (uz oznake $x := \langle \vec{x} \rangle$ i $n := \text{step}(x, e)$, dakle vrijedi $\text{Step}(x, e, n)$):

$$\begin{aligned} \text{univ}(x, e) &= \text{result}(\text{Reg}(x, e, n)) = \text{result}(\overline{\text{Reg}}(x, e, \text{Sc}(n))[n]) = \\ &= \text{result}(\text{rpart}(\overline{\text{Reg}}(x, e, \text{Sc}(n)), 0)) = U(\overline{\text{Reg}}(x, e, \text{Sc}(n))) = \\ &= U(\mu y (y = \overline{\text{Reg}}(x, e, \text{Sc}(n)))) = U(\mu y (y = \overline{\text{Reg}}(x, e, \text{Sc}(n)) \wedge \text{Step}(x, e, n))) = \\ &= U(\mu y \hat{T}(x, e, y)) = U(\mu y T_k(\vec{x}, e, y)) = \text{comp}_k(\vec{x}, e), \end{aligned} \quad (3.96)$$

iz čega slijedi tražena parcijalna jednakost. \square

Korolar 3.49. *Za svaki $k \in \mathbb{N}_+$, za sve $(\vec{x}, e) \in \mathbb{N}^{k+1}$ vrijedi:*

1. Ako je e kod nekog RAM-programa P , te ako P -izračunavanje s \vec{x} stane, tada je $\text{comp}_k(\vec{x}, e)$ rezultat tog izračunavanja.
2. U ostalim slučajevima ($e \notin \text{Prog}$, ili $e = \ulcorner P \urcorner$ ali P -izračunavanje s \vec{x} ne stane), izraz $\text{comp}_k(\vec{x}, e)$ nema smisla.

Dokaz. Ovo je zapravo lema 3.37, iskazana na malo drugačiji način koristeći upravo dokazanu parcijalnu jednakost (3.95) — i malo pojednostavljena jer znamo da vrijedi $\langle \vec{x}^k \rangle \in \text{Seq}'$ za $k \in \mathbb{N}_+$, pa to ne treba pisati u uvjete. \square

3.4.2. Indeksi izračunljivih funkcija

Sve bitno što smo dosad napravili u ovoj točki može se iskazati u jednom teoremu.

Teorem 3.50 (Kleenejev teorem o normalnoj formi). *Postoji primitivno rekurzivna funkcija U , takva da za svaki $k \in \mathbb{N}_+$ postoji primitivno rekurzivna relacija T_k , takva da za svaku parcijalno rekurzivnu funkciju f mjesnosti k postoji prirodni broj e , takav da za svaki $\vec{x} \in \mathbb{N}^k$ vrijede sljedeće dvije tvrdnje:*

$$\vec{x} \in \mathcal{D}_f \iff \exists y T_k(\vec{x}, e, y), \quad (3.97)$$

$$f(\vec{x}) \simeq U(\mu y T_k(\vec{x}, e, y)). \quad (3.98)$$

Dokaz. Funkcija U je definirana s (3.93), i dokazano je da je primitivno rekurzivna u propoziciji 3.48. Za svaki $k \in \mathbb{N}_+$, relacija T_k je definirana, i dokazano je da je primitivno rekurzivna, u propoziciji 3.46. Neka je sada f proizvoljna parcijalno rekurzivna funkcija. Prema teoremu 2.36, postoji RAM-program koji računa f . Uzmimo $e := \ulcorner P \urcorner$, i neka je $\vec{x} \in \mathbb{N}^k$ proizvoljan.

Prema definiciji 1.10, ako je $\vec{x} \in \mathcal{D}_f$, tada P -izračunavanje s \vec{x} stane, pa postoji njegov kod y_0 . Tada po definiciji (3.89) vrijedi $T_k(\vec{x}, e, y_0)$, pa postoji y (konkretno, y_0) takav da vrijedi $T_k(\vec{x}, e, y)$. Ako pak $\vec{x} \notin \mathcal{D}_f$, tada opet po definiciji 1.10 P -izračunavanje ne stane, te ne postoji njegov kod, odnosno ne postoji y takav da vrijedi $T_k(\vec{x}, e, y)$. Time je dokazano (3.97), odnosno dokazano je da u (3.98) lijeva i desna strana imaju smisla za iste \vec{x} (jer lijeva ima smisla za $\vec{x} \in \mathcal{D}_f$, a desna za $(\vec{x}, e) \in \exists_* T_k$).

Za takve \vec{x} , kao što smo vidjeli, postoji kod P -izračunavanja s \vec{x} , koji smo označili s y_0 i vidjeli da vrijedi $T_k(\vec{x}, e, y_0)$. Ali T_k ima funkcijsko svojstvo, dakle y_0 je *jedini*, pa onda i najmanji, y takav da vrijedi $T_k(\vec{x}, e, y)$. To znači da na desnoj strani zapravo piše $U(y_0)$, odnosno stanje registra \mathcal{R}_0 u zadnjoj konfiguraciji kodiranoj s y_0 — što je opet po definiciji 1.10 jednako $f(\vec{x})$, jer ta konfiguracija mora biti završna. \square

Korolar 3.51. *Za svaku parcijalno rekurzivnu funkciju postoji simbolička definicija u kojoj se operator μ pojavljuje točno jednom.*

Dokaz. Samo treba (3.98) zapisati u simboličkom obliku:

$$f = U \circ \mu T_k \circ (I_1^k, I_2^k, \dots, I_k^k, C_e^k). \quad (3.99)$$

Vidimo da su U , T_k , I_n^k i C_e^k primitivno rekurzivne, dakle u njihovim simboličkim definicijama se ne pojavljuje minimizacija. Iz toga slijedi da se pojavljuje isključivo u dobivanju funkcije μT_k . \square

Kad fiksiramo U i sve T_k , te pomoću njih definiramo funkcije comp_k (kao što smo i učinili), Kleenejev teorem o normalnoj formi može se izreći jednostavnije.

Korolar 3.52. *Za svaku $f \in \text{Comp}$ postoje $k \in \mathbb{N}_+$ i $e \in \mathbb{N}$ takvi da je funkcija $\vec{x} \mapsto \text{comp}_k(\vec{x}, e)$ jednaka f .*

Dokaz. Naravno, za k stavimo mjesnost funkcije f , a za e kod nekog RAM-programa koji računa f (koji postoji jer je f RAM-izračunljiva). Tada po Kleenejevom teoremu o normalnoj formi, funkcije o kojima tvrdnja govori imaju istu domenu i podudaraju se na toj domeni, dakle to su jednake funkcije. \square

Definicija 3.53. Za fiksne $k \in \mathbb{N}_+$ i $e \in \mathbb{N}$, k -mjesnu brojevenu funkciju $\vec{x} \mapsto \text{comp}_k(\vec{x}, e)$ označavamo s $\{e\}^k$, ili jednostavno s $\{e\}$ ako joj ne trebamo istaknuti mjesnost.

Broj e zovemo *indeksom* funkcije $\{e\}^k$ (broj k zovemo *mjesnošću* funkcije $\{e\}^k$).

Za funkciju f^k kažemo da *ima indeks* ako postoji $e \in \mathbb{N}$ takav da je $\{e\}^k = f^k$. \triangleleft

Korolar 3.54. *Za sve $k \in \mathbb{N}_+$, za sve $e \in \mathbb{N}$, funkcija $\{e\}^k$ je parcijalno rekurzivna.*

Dokaz. Samo treba zapisati upravo navedenu definiciju simbolički pomoću kompozicije: $\{e\}^k := \text{comp}_k \circ (I_1^k, I_2^k, \dots, I_k^k, C_e^k)$. Sada tvrdnja slijedi iz propozicije 3.48. \square

Korolar 3.55. *Svaka parcijalno rekurzivna funkcija ima indeks.*

Dokaz. Neka je $k \in \mathbb{N}_+$, i f^k parcijalno rekurzivna funkcija. Prema teoremu 2.36, $f \in \text{Comp}$. Sada prema korolaru 3.52 postoje k' i e takvi da je $f^k = \{e\}^{k'}$. Jednake funkcije moraju imati iste mjesnosti, pa je zapravo $k' = k$, odnosno $f^k = \{e\}^k$. \square

Napomena 3.56. Često se govori: „... , dakle funkcija f ima indeks, označimo ga s e_0 ”. Strogo govoreći, to je pogrešno, jer f , ako već ima indeks, sigurno **nema jedinstveni indeks**: programerska intuicija nam kaže da za svaki RAM-program postoji beskonačno mnogo RAM-programa koji su mu ekvivalentni (možemo dodavati irelevantne ili nedostupne instrukcije). Ipak, kako kasnije najčešće ne koristimo nikakva svojstva tog broja e_0 osim da je indeks od f , zapravo taj izričaj možemo shvatiti kao pokratu za „... , dakle funkcija f ima indeks; odaberimo jedan njen indeks, fiksirajmo ga i nazovimo ga e_0 ”.

Alternativno, možemo smatrati da smo uzeli *najmanji* indeks za f , koji sigurno postoji ako f ima indeks, i jednoznačno je određen (skup prirodnih brojeva je dobro

uređen), ali to ima jedan bitan nedostatak: često iz specifikacije neke funkcije f možemo *izračunati* neki indeks za f , ali ne možemo izračunati najmanji indeks za nju. Ugrubo, možemo provjeravati brojeve e redom, ali uvjet zaustavljanja $\{e\} = f$ je jednakost funkcija, koja nije izračunljiva. Čak i da su funkcije totalne (što ne moraju biti), morali bismo provjeriti sve \vec{x} iz \mathbb{N}^k , a ima ih beskonačno mnogo. \triangleleft

Drugim, riječima, tromjesna relacija index (zadana s $\{e\}^k = f$) između \mathbb{N} , \mathbb{N}_+ i Comp nema funkcijsko svojstvo po prvoj varijabli — ali ima po trećoj.

Propozicija 3.57. *Za svaki $k \in \mathbb{N}_+$, za svaki $e \in \mathbb{N}$, vrijedi:*

1. *Ako je $e \in \text{Prog}$, tada je $\{e\}^k$ jedinstvena funkcija koju računa RAM-algoritam P^k , gdje je P jedinstveni RAM-program takav da je $\lceil P \rceil = e$.*
2. *Ako $e \notin \text{Prog}$, tada je $\{e\}^k = \emptyset^k$.*

Dokaz. Ako je $e \in \text{Prog}$, tada postoji $P \in \text{Prog}$ takav da je $\lceil P \rceil = e$, i jedinstven je jer je $\lceil \cdot \rceil$ injekcija. Po korolaru 1.13, postoji jedinstvena funkcija f^k koju P^k računa. Po definiciji 1.10, za tu funkciju vrijedi: za sve $\vec{x} \in \mathcal{D}_f$, P -izračunavanje s \vec{x} stane, pa je po korolaru 3.49(1), $f(\vec{x}) = \text{comp}_k(\vec{x}, e) = \{e\}^k(\vec{x})$. Za sve pak $\vec{x} \notin \mathcal{D}_f$, P -izračunavanje s \vec{x} ne stane, pa po korolaru 3.49(2) izraz $\text{comp}_k(\vec{x}, e) \simeq \{e\}^k(\vec{x})$ nema smisla, baš kao ni $f(\vec{x})$. Dakle uvijek je $f(\vec{x}) \simeq \{e\}(\vec{x})$, odnosno $f = \{e\}$.

S druge strane, ako $e \notin \text{Prog}$, tada opet po korolaru 3.49(2) izraz $\text{comp}(\vec{x}, e) \simeq \{e\}(\vec{x})$ nema smisla, ali ovaj put neovisno o $\vec{x} \in \mathbb{N}^k$. Drugim riječima $\mathcal{D}_{\{e\}^k} = \emptyset^k$, odnosno jedino je moguće $\{e\}^k = \emptyset^k$. \square

Korolar 3.58. *Svaki RAM-algoritam P^k računa funkciju $\{\lceil P \rceil\}^k$.*

Dokaz. Ovo je samo jezgrovit zapis propozicije 3.57(1). \square

Relacija index na taj način može poslužiti kao neka vrsta kodiranja: kad želimo dati algoritmu izračunljivu funkciju kao ulazni podatak, ili vratiti iz algoritma izračunljivu funkciju kao izlazni podatak, možemo prenijeti njen indeks e (k se obično vidi iz konteksta). To nije pravo kodiranje, jer nije jednoznačna funkcija ako kažemo „bilo koji indeks”, a nije izračunljiva funkcija ako kažemo „najmanji indeks” — pogledajte napomenu 3.56. Ipak, ako to slanje indeksa u algoritam i vraćanje indeksa iz algoritama shvatimo kao parcijalnu specifikaciju, to može funkcionirati — pogledajmo primjer.

Primjer 3.59. Recimo da imamo funkciju $F: \text{Comp}_2 \rightarrow \text{Comp}_3$. Drugim riječima, F preslikava dvomjesne RAM-izračunljive funkcije u tromjesne RAM-izračunljive funkcije. Kao i prije, htjeli bismo izračunljivost funkcije F opisati pomoću izračunljivosti prateće funkcije NF , koja prima indeks funkcije f , i vraća indeks funkcije $F(f)$. Zbog

napomene 3.56 funkciju $\mathbb{N}F$ ne možemo time potpuno opisati, ali možemo reći što od nje tražimo: to je da za *svaki* $e \in \mathbb{N}$ bude

$$\{\mathbb{N}F(e)\}^3 = F(\{e\}^2). \quad (3.100)$$

Riječima, ako F preslikava f^2 u g^3 , tada $\mathbb{N}F$ mora preslikavati *svaki* indeks za f^2 u *neki* (ne nužno isti) indeks za g^3 — a ako $f \notin \mathcal{D}_F$, tada *nijedan* indeks za f ne smije biti u $\mathcal{D}_{\mathbb{N}F}$. Takvih funkcija općenito ima neprebrojivo mnogo — jer indeksa za svaku $g \in \mathcal{I}_F$ ima beskonačno mnogo — pa ih ima i neizračunljivih. Ali ako *postoji* bar jedna takva funkcija koja je izračunljiva u nekom smislu i zadovoljava (3.100), kažemo da je F izračunljiva u istom tom smislu.

Takve funkcije možemo i komponirati: ako imamo $G: \text{Comp}_3 \rightarrow \text{Comp}_1$ sa sličnom specifikacijom, koja preslikava g^3 u h^1 , tada iako ne znamo koji će nam indeks od g funkcija $\mathbb{N}F$ dati, funkcija $\mathbb{N}G$ mora ispravno raditi za *sve* indekse od g , pa će na kraju $\mathbb{N}G \circ \mathbb{N}F$ dati neki indeks za h , ako joj damo (bilo koji) indeks za f . \triangleleft

Naravno, to možemo kombinirati s dosad napravljenim kodiranjima: recimo, funkcija $\text{apply}_k: \mathbb{N}^k \times \text{Comp}_k \rightarrow \mathbb{N}$, koja prima \vec{x}^k i f^k , i vraća $f(\vec{x})$ ako je $\vec{x} \in \mathcal{D}_f$, ima svoju prateću funkciju koja prima $\langle \vec{x} \rangle$ i bilo koji indeks za f , i vraća $f(\vec{x})$ ako postoji.

$$\mathbb{N}\text{apply}_k(x, e) : \simeq \text{univ}(x, e), \text{ za } lh(x) = k \quad (3.101)$$

Napomena 3.60. Nažalost, iz toga još uvijek ne slijedi da je takva funkcija izračunljiva, iako su univ , lh i $=$ izračunljive — jer trebamo neki rezultat koji kaže da je restrikcija parcijalno rekurzivne funkcije na (primitivno) rekurzivan skup ponovo parcijalno rekurzivna. To sigurno možemo „na prste” — recimo, probajte pokazati da je

$$(G|_{\mathbb{R}})(\vec{x}) \simeq G(\vec{x}) + \mu y (Sc(y) = \chi_{\mathbb{R}}(\vec{x})) \quad (3.102)$$

— ali zapravo će to trivijalno slijediti iz rezultata u idućoj točki. \triangleleft

3.4.3. Parcijalno rekurzivna verzija teorema o grananju

Na početku točke 2.4 nešto smo rekli o teškoćama koje marljiva evaluacija nosi ako želimo u našem funkcijskom jeziku implementirati grananje s funkcijama koje nisu nužno totalne. Zapravo, jedino što možemo koristiti je kompozicija, jer primitivna rekurzija je definirana samo na totalnim funkcijama, a minimizacija na relacijama čije karakteristične funkcije su također totalne — ali marljiva evaluacija znači da se u kompoziciji sve „unutarnje” funkcije evaluiraju uvijek, i parcijalnost bilo koje od njih znači parcijalnost čitave kompozicije.

Dakle, način da se izvučemo je da našoj funkciji if ne damo već izračunate *vrijednosti* $g(\vec{x})$ i $h(\vec{x})$, već neevaluirane *funkcije* g i h . Tada s obzirom na uvjet odaberemo jednu od njih, i onda je tek izračunamo na \vec{x} : umjesto $f(\vec{x}, y) \simeq if(y, g(\vec{x}), h(\vec{x}))$ imamo $f(\vec{x}, y) \simeq if(y, g, h)(\vec{x})$. Koristeći indekse, to možemo i doslovno napraviti.

Teorem 3.61 (Teorem o grananju, parcijalno rekurzivna verzija). *Neka su $k, l \in \mathbb{N}_+$, neka su $G_0^k, G_1^k, G_2^k, \dots, G_l^k$ parcijalno rekurzivne funkcije, te $R_1^k, R_2^k, \dots, R_l^k$ u parovima disjunktne rekurzivne relacije, sve iste mjesnosti. Tada je i funkcija $F := \{R_1: G_1, R_2: G_2, \dots, R_l: G_l, G_0\}$ također parcijalno rekurzivna.*

Dokaz. Za svaki $i \in [0..l]$, funkcija G_i^k je parcijalno rekurzivna, pa prema korolaru 3.55 ima indeks, označimo ga s e_i (pogledajte napomenu 3.56 za značenje ove fraze). Prema teoremu 2.44 (rekurzivna verzija), funkcija $H^k := \{R_1: C_{e_1}^k, R_2: C_{e_2}^k, \dots, R_l: C_{e_l}^k, C_{e_0}^k\}$ je rekurzivna (konstante su primitivno rekurzivne pa su rekurzivne, a uvjeti su u parovima disjunktne i rekurzivni po pretpostavci). Tvrdimo da je

$$F(\vec{x}) \simeq \text{comp}_k(\vec{x}, H(\vec{x})). \quad (3.103)$$

Doista, neka je $\vec{x} \in \mathbb{N}^k$ proizvoljan. Ako vrijedi $R_i(\vec{x})$ za neki (jedinstveni, zbog disjunktosti) $i \in [1..l]$, tada je prema teoremu 2.44, $H(\vec{x}) = C_{e_i}(\vec{x}) = e_i$, pa je

$$\text{comp}_k(\vec{x}, H(\vec{x})) \simeq \text{comp}_k(\vec{x}, e_i) \simeq \{e_i\}(\vec{x}) \simeq G_i(\vec{x}) \simeq F(\vec{x}). \quad (3.104)$$

Ako pak ne vrijedi $R_i(\vec{x})$ ni za koji i , tada je opet prema teoremu 2.44, $H(\vec{x}) = C_{e_0}(\vec{x}) = e_0$, pa je kao i prije $\text{comp}_k(\vec{x}, H(\vec{x})) \simeq G_0(\vec{x}) \simeq F(\vec{x})$.

Sada parcijalna rekurzivnost slijedi iz (3.103), jer je F dobivena kompozicijom iz parcijalno rekurzivnih funkcija comp_k, H , te koordinatnih projekcija. \square

U teoremu 2.44 nismo mogli ispustiti G_0 , jer njena podrazumijevana vrijednost \emptyset^k nije rekurzivna (nije uopće totalna). Ali \emptyset^k jest parcijalno rekurzivna, tako da je ovdje možemo ispustiti — samo nam treba neki indeks za nju, da bi H bila totalna funkcija.

Za to možemo iskoristiti propoziciju 3.57(2) — svi brojevi iz Prog^c indeksi su prazne funkcije. Posebno lijep takav broj je 0, koji nije u Prog jer uopće nije u Seq : naime, $\overline{\text{part}}(0, \text{lh}(0)) = \overline{\text{part}}(0, 0) = \langle \rangle = 1 \neq 0$.

Korolar 3.62. *Neka su $k, l \in \mathbb{N}_+$, neka su $G_1^k, G_2^k, \dots, G_l^k$ parcijalno rekurzivne funkcije, te $R_1^k, R_2^k, \dots, R_l^k$ u parovima disjunktne rekurzivne relacije, sve iste mjesnosti. Tada je i funkcija $F := \{R_1: G_1, R_2: G_2, \dots, R_l: G_l\}$ parcijalno rekurzivna.*

Dokaz. Direktno iz teorema 3.61, uvrštavajući za G_0 parcijalno rekurzivnu funkciju \emptyset^k , odnosno njen indeks $e_0 = 0$ u dokaz. \square

Korolar 3.63. *Neka je $k \in \mathbb{N}_+$, te neka je G^k parcijalno rekurzivna funkcija, i R^k rekurzivna relacija iste mjesnosti. Tada je i restrikcija $G|_R$ parcijalno rekurzivna.*

Dokaz. Direktno iz korolara 3.62, uvrštavajući $l := 1$, $G_1 := G$, te $R_1 := R$. Lako se vidi da je funkcija $\{R: G\}$ upravo jednaka $G|_R$: domena joj je $\mathcal{D}_G \cap R$, a vrijednosti su joj jednake vrijednostima funkcije G na tom skupu. \square

Poglavlje 4.

Turing-izračunljivost

4.1. Zašto nam treba još jedan model?

Uveli smo dva modela izračunljivosti brojevni funkcija — RAM-izračunljivost i parcijalna rekurzivnost — i dokazali da su ekvivalentni, usprkos bitno različitim pristupima (imperativnom odnosno funkcijskom) definiciji algoritma. To pruža dobar argument u korist Church-Turingove teze, da su izračunljive funkcije iste u svim modelima — odnosno u modernijem obliku, da su sve programske paradigme jednako snažne. Ipak, pritom su zanemarena dva aspekta izračunljivosti.

Prvi je izračunljivost funkcija koje nisu brojevne. Rekli smo u uvodnom poglavlju, i opravdali to mnogo puta kasnije — skup \mathbb{N} je idealan za matematički tretman izračunljivosti, ali nije baš vjeran onom što se događa u praksi. Glavna razlika leži u konačnosti odnosno beskonačnosti skupova relevantnih za izračunavanje.

Recimo, u RAM-modelu, skup mogućih stanja pojedinog registra je beskonačan. Često je problematično shvaćanje beskonačnosti kao „jako puno, pa još malo više”. Zapravo, bliže definiciji beskonačnosti bilo bi „jako puno, pa onda još beskonačno više” — ma koliko velik konačni skup uzeli od beskonačnog skupa, ostatak je jednako velik kao da nismo uzeli ništa.

Konkretno, lako se zavarati primjerima u kojima u RAM-registrima stoje brojevi poput 0, 150 ili 2^{257} , i misliti da su RAM-registri nešto kao obični procesorski registri, samo „malo veći”. Kodiranje pokazuje koliko je to daleko od istine: u RAM-registar možemo staviti i brojeve poput broja e_0 iz primjera 3.29, koji ima 579 690 725 279 *znamenaka*! Štoviše, takva procedura nije ništa neuobičajeno; ona odgovara prirodnom postupku računanja $\text{univ}(\langle 2, 4 \rangle, e_0)$, odnosno konstrukciji rezultata Q^b -izračunavanja s $(2, 4)$, na univerzalnom RAM-stroju (dobivenom kompajliranjem simboličke definicije funkcije *univ*).

Stvarna računala, naravno, takve komplicirane strukture (zamislimo e_0 kao neku vrstu strojnog koda, *bytecode*) ne kodiraju pomoću prirodnih brojeva, već preko nizova bitova, bajtova ili većih procesorskih *riječi*. Ključno je da je skup Γ svih stanja pravog procesorskog registra *konačan*. Način na koji se onda reprezentira potencijalna beskonačnost ulaznih podataka (što moramo, jer jedino beskonačni skupovi imaju

netrivijalnu teoriju izračunljivosti), kao i međurezultata u izračunavanju, je kroz neograničenost same memorije, odnosno broja memorijskih ćelija koje sadrže po jedan element iz Γ .

I ovdje, kao i svugdje, vrijedi univerzalni princip da na konačnim skupovima možemo dopustiti bilo kakve transformacije kao elementarne korake, i jednostavno ih reprezentirati tablicama — dok s beskonačnim skupovima moramo biti oprezni, i restringirati transformacije samo na one izračunljive. Kako je taj beskonačni skup u pravilu trivijalno izomorfan s \mathbb{N} , kao osnovne korake dopuštamo samo prelazak na sljedeći odnosno prethodni (ako već nije 0) prirodni broj.

Zato smo na konfiguracijama RAM-stroja dozvoljavali samo one prijelaze kod kojih se stanje pojedinog registra mijenja za najviše 1 u svakom koraku; dok smo za stanje programskog brojača dozvoljavali skokove na proizvoljnu legalnu vrijednost — upravo jer legalnih vrijednosti programskog brojača, za fiksni RAM-stroj, ima konačno mnogo. Također, osnovna ideja od koje dolazi i ime RAM-stroja, *random access*, znači da njegova „memorijska sabirnica” može adresirati proizvoljni registar u jednom koraku — što možemo upravo jer adresa relevantnih registara, za fiksni RAM-algoritam, ima konačno mnogo (širina algoritma).

Ako želimo beskonačnost prikazati ne kroz veličinu sadržaja pojedinog registra nego kroz broj ćelija potrebnih da se zapiše podatak, zapravo imamo „transponirani” model: na pojedinoj ćeliji (jednom kad dođemo do nje) ćemo moći napraviti proizvoljnu transformaciju u jednom koraku, jer je skup Γ mogućih stanja pojedine ćelije konačan — ali pristup do pojedine ćelije više neće moći biti *random access*, već ćemo u jednom koraku samo moći adresu trenutno adresirane ćelije povećati ili smanjiti (osim ako je već jednaka 0) za jedan. To zovemo *jezični model* izračunavanja, jer odgovara onom kako (bar zapadni) jezici funkcioniraju: od konačnog broja slova u abecedi nizanjem možemo dobiti proizvoljno komplicirane riječi, rečenice i tekstove. Da bismo povećali izražajnost, ne uvodimo nova slova, već pišemo dulje rečenice.

Možda ovdje treba objasniti kako moderna računala postižu *random access* i na potencijalno neograničenoj memoriji. Objašnjenje je jednostavno: varaju. Njihova memorija *nije* potencijalno neograničena, jer ovisi o veličini adresnog prostora. Jedno 64-bitno računalo, koliko god mu virtualne memorije dali, ne može adresirati više od 2^{64} bajtova. To varanje u stvarnom svijetu prolazi jer s trenutnom tehnologijom ne možemo uopće sastaviti funkcionalnu memoriju od 2^{64} bajtova (što je više od osamnaest milijuna terabajta), ali za 32-bitna računala to ograničenje (na 4 GiB u tom slučaju) je bilo vrlo stvarno i nezgodno, i uostalom jedan od glavnih razloga za prijelaz na 64-bitnu arhitekturu.

Ista vrsta varanja koja je nedavno došla do granice svojih mogućnosti je internetski protokol IPv4, koji je dobro spomenuti jer pruža pogled na to kako se takvi problemi mogu riješiti iteriranjem adresiranja: NAT (*Network Address Translation*) pokazuje

da ako se u jednom koraku (DNS) može adresirati $t = 2^{32}$ računala, u dva koraka (DNS + *router*) se može u idealnom slučaju adresirati $t^2 = 2^{64}$ računala. Više nam vjerojatno neće nikada trebati jer ćemo u međuvremenu vjerojatno prijeći na IPv6, ali u n koraka mogli bismo adresirati t^n računala, efektivno zamišljajući generaliziranu IP-adresu kao n -znamenasti broj u bazi t , ili $32n$ -znamenasti broj u bazi 2. Jedina razlika jezičnog modela je što umjesto binarnog zapisa adrese korisimo unarni, u kojem su elementarne operacije samo inkrement, dekrement i uspoređivanje s nulom.

Iako je unarni zapis eksponencijalno lošiji od binarnog (i svih ostalih pozicijskih) zapisa — broj koji u bazama 2, 3, 4, ... ima nekoliko desetaka znamenaka, zapisan unarno može imati milijarde milijardi milijardi ... „znamenaka” — opet, to je samo razlika u složenosti, odnosno u performansama algoritma, ne u samom postojanju algoritma, te nam kao takva neće biti bitna. Ono što nam jest bitno je da u jezičnom modelu adresiranje pojedine ćelije zahtijeva netrivialne algoritme, a bilo kakva promjena sadržaja ćelije je elementarna operacija — upravo suprotno od RAM-modela.

4.1.1. Računanje bez računala

Drugi aspekt koji smo u potpunosti zanemarili u brojevnom modelu je povijesni. Danas, kad smo na svakom koraku okruženi računalima, a većina nas jedno nosi u džepu, lako je zaboraviti da računala ne postoje oduvijek. Zapravo, u upotrebljivom obliku postoje tek nekoliko desetaka godina. S druge strane, algoritmi postoje već milenijima: Euklidov algoritam je nastao prije modernog brojenja godina, a neki babilonski algoritmi potječu sa samih početaka pisane povijesti. Što znači algoritam ako nema računala na kojem se može izvršavati?

Odgovor je jednostavan: algoritme su izvršavali ljudi. Iako ljudski mozak po svojoj prirodi nije savršen supstrat za doslovno slijeđenje instrukcija kroz višeznamenaste brojeve koraka, moderno obrazovanje svjedoči da se može tome naučiti. Doista, većina zadataka iz školske matematike može se svesti na provođenje nekog algoritma. Mnoge od tih zadataka računala obavljaju brže i bolje od ljudi, o čemu svjedoči uspjeh aplikacija kao što je [photomath](#). Iako su ljudi bolji u pronalaženju (logičkih, analogijskih ili asocijativnih) veza među pojmovima, potreba za rješavanjem problema iz stvarnog života koji se mogu precizno klasificirati uvjetovala je pronalazak mnogih algoritama. Motivacija je uvijek bila ista: optimizacija i specijalizacija ljudskog rada. Jedan čovjek može osmisliti algoritam, koji poslije milijuni ljudi mogu provoditi i tako rješavati stvarne probleme, ne razumijevajući nužno zašto algoritam radi. No da bi to uspjelo, koraci algoritma moraju biti takvi da njegovo provođenje ne zahtijeva nikakav angažman pored onog koji je specificiran algoritmom, niti ikakvo vanjsko znanje — osim poznavanja ulaznih podataka, i nekoliko elementarnih vještina poput čitanja i pisanja fiksnog skupa simbola, te odlučivanja na osnovi pročitano, za koje smatramo da su svojstvene svim radno sposobnim ljudima.

Britanski matematičar Alan Mathison Turing prvi je uspješno formalizirao taj koncept. U svom članku [15], prije više od 80 godina i svakako prije nastanka digitalnih elektroničkih računala, Turing govori o ljudima koji računaju decimale nekih konkretno zadanih realnih brojeva — za što danas znamo da je vrlo slično računanju vrijednosti nekih konkretno zadanih brojevnih funkcija. Iako je i prije tog članka bilo pokušaja formalizacije algoritma, Turingov se ističe po tome što detaljno motivira svoje definicije, koristeći tada poznate činjenice vezane uz ljudsku percepciju i kogniciju. Čovjekov fizički rad za vrijeme provođenja algoritma, i njegova misaona stanja kroz koja prolazi, nisu samo incidentni dio opisa algoritma — oni su u tom članku suštinski ugrađeni u definiciju. Tako možemo biti sigurni da doista modeliramo ono što se događa u stvarnom svijetu kad čovjek provodi algoritam, a ne matematičku apstrakciju kao što je recimo λ -račun.

Odakle onda u priči Turingovi *strojevi*? Turing je bio svjestan fenomena da je izuzetno lako pomisliti kako opisujemo postupak koji ne zahtijeva nikakvo eksterno znanje, a da to zapravo nije istina. Razumijevanje napisanog ili izgovorenog jezika, prepoznavanje objekata na slikama, pa čak i osnove socijalnog ponašanja, vještine su koje smo toliko duboko internalizirali da nam se čine elementarnima — a zapravo pretpostavljaju ogromne količine znanja o svijetu koji nas okružuje. Jednostavni primjer: rečenice hrvatskog jezika „Ana i Marija su sestre.” i „Ana i Marija su majke.” imaju potpuno istu sintaksnu strukturu, ali fundamentalno različitu semantiku, za čiju je konstrukciju potrebno netrivialno znanje o ljudskoj biologiji — a ipak nam svaka od te dvije semantike dođe sasvim prirodno čitajući odgovarajuću rečenicu, i uopće ne razmišljamo kako bi moglo biti drugačije, sve dok ih ne vidimo jednu pored druge.

Da bi svoje čitatelje uvjerio kako njegove elementarne operacije doista ne zahtijevaju nikakvo implicitno pretpostavljeno znanje, Turing je paralelno opisao i zamišljeni, idealizirani, *stroj* koji može provoditi te operacije. Taj stroj, odnosno njegovu matematičku formalizaciju (do na neke kasnije ispeglane detalje radi lakšeg razumijevanja) danas nazivamo Turingovim strojem. Ipak, treba razumjeti da „ove operacije su toliko elementarne da bi ih mogao provoditi i mehanički stroj” nije poziv na konstrukciju stvarnog stroja, već apel na intuiciju da smo lišili osnovni opis algoritma svega suvišnog.

U literaturi postoje brojne varijante Turingovog stroja — mi slijedimo [12], uz neke male modifikacije kako bismo lakše dokazivali teoreme. Za početak ponovimo definicije.

Ulazna abeceda, ili jednostavno *abeceda*, je konačan neprazan skup. Obično je označavamo sa Σ , i smatramo fiksnom. Njene elemente zovemo *znakovima* — neodređene znakove označavamo malim grčkim slovima s početka alfabeta (α , β , γ), dok konkretne znakove pišemo u fontu fiksne širine: **a**, **b**, **0**, **1**. *Riječ* (nad Σ) je bilo koji konačan niz znakova, najčešće označen slovom w — pišemo je konkatencijom znakova: recimo, riječ **(0, 1, 1)** pišemo kao **011**. Dakle, skup svih riječi je skup svih konačnih nizova znakova, Σ^* . Duljinu riječi označavamo s $|w|$. Dopusćamo i mogućnost praznog

niza (duljine 0), koji zovemo *praznom riječju* i označavamo s ε . *Jezik* (nad Σ) je bilo koji podskup od Σ^* . *Jezična funkcija* (nad Σ) je bilo koja parcijalna funkcija $\varphi: \Sigma^* \rightarrow \Sigma^*$.

Napomena 4.1. Kao i mjesnost kod brojevnihi funkcija i relacija, tako i abecedu kod jezika i jezičnih funkcija smatramo dijelom njihovog identiteta; preslikavanje nad $\{a, b\}$ koje riječi pridružuje njen reverz (obrnuto čitanu riječ), različito je od preslikavanja nad $\{a, c\}$ zadanog istim pravilom. Ili, jezik svih riječi koje se sastoje samo od znakova a i b je različit kao jezik nad $\{a, b, c\}$ i kao jezik nad $\{a, b, d\}$ — iako se u ovom slučaju radi o skupu s istim elementima. Motivacija je slična kao u slučaju praznih relacija: komplementi su različiti, a i karakteristične funkcije jezika su različite (jer imaju različite domene). \triangleleft

Definicija 4.2. Neka je Σ abeceda. *Turingov stroj* (nad Σ) je matematički (idealizirani) stroj, obično zapisan kao uređena sedmorka $\mathcal{T} = (Q, \Sigma, \Gamma, \sqcup, \delta, q_0, q_z)$, koji sadrži:

- konačan skup *stanja* Q , s istaknutim elementima $q_0 \in Q$ (*početno stanje*) i $q_z \in Q$ (*završno stanje*);
- konačnu *radnu abecedu* $\Gamma \supset \Sigma$, s istaknutim elementom $\sqcup \in \Gamma \setminus \Sigma$ (*praznina*);
- *funkciju prijelaza* $\delta: (Q \setminus \{q_z\}) \times \Gamma \rightarrow Q \times \Gamma \times \{-1, 1\}$. \triangleleft

Umjesto registara, Turingov stroj ima *ćelije* (također adresirane prirodnim brojevima), svaka od kojih u svakom trenutku izračunavanja sadrži proizvoljni element od Γ . Kao što su kod RAM-stroja na početku izračunavanja svi registri osim ulaznih bili inicijalizirani na 0, tako će kod Turingovog stroja sve ćelije osim ulaznih biti inicijalizirane na \sqcup . Po toj analogiji, označimo $\Gamma_+ := \Gamma \setminus \{\sqcup\}$.

Definicija 4.3. Neka je $\mathcal{T} = (Q, \Sigma, \Gamma, \sqcup, \delta, q_0, q_z)$ Turingov stroj. *Konfiguracija* od \mathcal{T} je bilo koja uređena trojka $(q, n, t) \in Q \times \mathbb{N} \times \Gamma^{\mathbb{N}}$, takva da je niz t skoro svuda \sqcup (odnosno, $t^{-1}[\Gamma_+]$ je konačan skup). Komponente konfiguracije zovu se redom *stanje*, *pozicija* i *traka*. Konfiguracija je *završna* ako joj je stanje završno (q_z). *Početna konfiguracija* s ulazom $w = \alpha_0 \alpha_1 \dots \alpha_{|w|-1} \in \Sigma^*$ je trojka $(q_0, 0, w_{\sqcup} \dots)$, gdje je traka

definirana sa $(w_{\sqcup} \dots)_i := \begin{cases} \alpha_i, & i < |w| \\ \sqcup, & \text{inače} \end{cases}$.

Za konfiguracije $c = (q, n, t)$ i $d = (q', n', t')$ istog Turingovog stroja \mathcal{T} kažemo da c *prelazi* u d , i pišemo $c \rightsquigarrow d$, ako je c završna i $c = d$, ili uz oznake $\delta(q, t_n) =: (p, \beta, d)$ vrijedi $q' = p$, $n' = \max\{n + d, 0\}$, $t'_n = \beta$, te $t'_i = t_i$ za sve $i \in \mathbb{N} \setminus \{n\}$. \triangleleft

Traku možemo zamisliti kao s jedne strane ograničen, a s druge strane neograničen, niz ćelija, u kojem su od nekog mjesta nadalje samo prazne ćelije (one u kojima piše praznina). Ulaz za Turingov stroj je riječ nad Σ , koja se na početku izračunavanja

zapiše na lijevi kraj trake redom (ostatak trake je prazan). U svakom koraku, funkcija prijelaza prima trenutno stanje i sadržaj trenutne ćelije, te ih preslikava u novo stanje, novi znak trenutne ćelije, te pomak ulijevo ili udesno na susjednu ćeliju, koja time postaje trenutna u idućem koraku (pomak ulijevo od početne ćelije rezultira ostajanjem na mjestu). To se događa dok konfiguracija ne postane završna, i tada, ako je traka oblika $v_{\square} \dots$ za neku riječ $v \in \Sigma^*$, kažemo da je v izlaz Turingovog stroja s ulazom w .

Lema 4.4. *Neka je \mathcal{T} Turingov stroj. Svaka konfiguracija od \mathcal{T} prelazi u jedinstvenu konfiguraciju od \mathcal{T} .*

Dokaz. Neka je \mathcal{T} Turingov stroj, i $c = (q, n, t)$ proizvoljna njegova konfiguracija. Ako je $q = q_z$, tada $c \rightsquigarrow c$, i ni u koju drugu konfiguraciju jer δ nije definirana u (q_z, t_n) . Ako pak c nije završna, postoje jedinstveni p, β i d takvi da je $\delta(q, t_n) = (p, \beta, d)$, koji jednoznačno (zajedno s q, n i t) određuju q', n' i t' takve da $c \rightsquigarrow (q', n', t')$. \square

Definicija 4.5. Neka je Σ abeceda, neka je $w \in \Sigma^*$ riječ, te neka je \mathcal{T} Turingov stroj nad Σ . \mathcal{T} -izračunavanje s w je niz $(c_n)_{n \in \mathbb{N}}$ konfiguracija od \mathcal{T} , takav da je c_0 početna konfiguracija s ulazom w , te za svaki $i \in \mathbb{N}$, $c_i \rightsquigarrow c_{i+1}$. Kažemo da to izračunavanje stane ako postoji $n_0 \in \mathbb{N}$ takav da je c_{n_0} završna konfiguracija.

Neka je φ jezična funkcija nad Σ . Kažemo da \mathcal{T} računa φ ako za sve $w \in \Sigma^*$ vrijedi:

- Ako je $w \in \mathcal{D}_{\varphi}$, tada \mathcal{T} -izračunavanje s w stane, i završna konfiguracija mu je oblika $(q_z, n, \varphi(w)_{\square} \dots)$ za neki $n \in \mathbb{N}$ (pozicija nije bitna).
- Ako $w \notin \mathcal{D}_{\varphi}$, tada \mathcal{T} -izračunavanje s w ne stane.

Za jezičnu funkciju φ kažemo da je *Turing-izračunljiva* ako postoji Turingov stroj koji je računa. \triangleleft

Kao i za RAM-model, mogli bismo dokazati da za svaki Turingov stroj \mathcal{T} i njegov ulaz w postoji jedinstveno \mathcal{T} -izračunavanje s w , ali više ne vrijedi da svaki Turingov stroj računa neku jezičnu funkciju. Naime, traka u završnoj konfiguraciji ne mora biti oblika $v_{\square} \dots$ za $v \in \Sigma^*$: može sadržavati znakove iz $\Gamma_+ \setminus \Sigma$, ili sadržavati neki znak iz Σ nakon prve praznine. Za takve Turingove strojeve nećemo reći da računaju ikakvu funkciju. Zapravo ih nećemo uopće promatrati, ali dobro ih je imati na umu ako iskazujemo teoreme univerzalno po svim Turingovim strojevima.

4.1.2. Primjer Turing-izračunavanja

Primjer 4.6. Neka je $\Sigma := \{a, b\}$, i promotrimo funkciju $\varphi_h: \Sigma^* \rightarrow \Sigma^*$, takva da je \mathcal{D}_{φ_h} skup svih riječi parne duljine, a $\varphi_h(\alpha_1 \alpha_2 \dots \alpha_{2k}) := \alpha_1 \alpha_2 \dots \alpha_k$ (prva polovica riječi). Konkretno, za $w_0 := \text{aababa}$ vrijedi $|w_0| = 6$, pa je $w_0 \in \mathcal{D}_{\varphi_h}$, i $\varphi_h(w_0) = \text{aab}$.

Funkcija φ_h je Turing-izračunljiva: računa je Turingov stroj

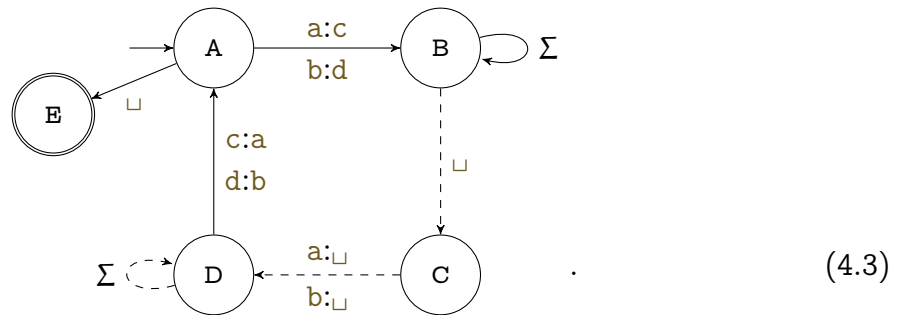
$$\mathcal{T}_h := (\{A, B, C, D, E, F\}, \Sigma, \{\sqcup, a, b, c, d\}, \sqcup, \delta_h, A, E), \quad (4.1)$$

čija je funkcija prijelaza (kao konačna funkcija) zadana tablicom

δ_h	\sqcup	a	b	c	d
A	(E, \sqcup , +1)	(B, c, +1)	(B, d, +1)	(F, c, +1)	(F, d, +1)
B	(C, \sqcup , -1)	(B, a, +1)	(B, b, +1)	(F, c, +1)	(F, d, +1)
C	(F, \sqcup , +1)	(D, \sqcup , -1)	(D, \sqcup , -1)	(F, c, +1)	(F, d, +1)
D	(F, \sqcup , +1)	(D, a, -1)	(D, b, -1)	(A, a, +1)	(A, b, +1)
F	(F, \sqcup , +1)	(F, a, +1)	(F, b, +1)	(F, c, +1)	(F, d, +1)

(4.2)

Kao što vidimo, takav način zadavanja Turingovog stroja nije naročito čitljiv. Zato se obično crtaju dijagrami. Recimo, \mathcal{T}_h bismo mogli prikazati dijagramom



Recimo ponešto o konvencijama pri crtanju takvih dijagrama. Crtamo konačni usmjereni graf, čiji su vrhovi stanja, a bridovi su prijelazi. Opće pravilo je da se prijelaz $\delta(p, \alpha) = (q, \beta, d)$ prikazuje kao strelica od vrha p prema vrhu q, na kojoj piše $\alpha:\beta$. Strelicu crtamo kao punu ako je $d = 1$ (pomak udesno), a kao iscrtkanu ako je $d = -1$ (pomak ulijevo).

Više prijelaza s istim p, q i d može biti prikazano jednom strelicom, na kojoj se nalazi više oznaka $\alpha_1:\beta_1, \dots, \alpha_k:\beta_k$. Ako se znak ne mijenja ($\alpha = \beta$), umjesto $\alpha:\alpha$ pišemo jednostavno α . Ako već imamo oznaku za skup $S \subseteq \Gamma$, možemo je napisati na brid umjesto pojedinih elemenata (kao što smo na dijagramu učinili sa Σ).

Početno stanje označavamo „strelicom niotkud”, kako je na dijagramu označeno stanje A. Završno stanje označavamo dvostrukim krugom, kako je na dijagramu označeno stanje E. I još jedno pravilo koje bitno povećava preglednost dijagrama: ne pišemo stanja ni prijelaze koji više ne mogu voditi do završnog stanja. Običaj je prilikom konstrukcije Turingovog stroja imati jedno stanje q_x (u slučaju \mathcal{T}_h to je stanje F), takvo da se svaki „nemoguć prijelaz”, odnosno nemoguća situacija (q, α) , po δ preslika u $(q_x, \alpha, 1)$. Specijalno to vrijedi i za $q = q_x$, za sve $\alpha \in \Gamma$, iz čega slijedi da nakon što Turingov stroj napravi neki od tih nemogućih prijelaza, više nikada neće stati.

Efektivno, to je „stanje greške”, i ne moramo ga, kao ni prijelaze koji prema njemu vode, crtati na dijagramu: podrazumijevamo da riječi za koje se dogodi neka od tih „nemogućih situacija” nisu u domeni funkcije koju Turingov stroj računa. Važno je da ne mora vrijediti obrat: Turingov stroj ne mora nikada ući u stanje q_x , a da ipak nikada ne stane. Možemo to usporediti s instrukcijom RAM-stroja $i.GO\ TO\ i$ — ako PC ikad postane i , RAM-stroj sigurno ne stane, ali može ne stati i na druge načine.

Konfiguracije Turingovog stroja u konkretnom izračunavanju obično se označavaju skraćeno: ispod trenutno čitanog znaka napišemo trenutno stanje, odnosno umjesto $(q, n, (t_m)_{m \in \mathbb{N}})$ pišemo $t_0 t_1 \dots t_n t_{n+1} \dots$ — u tom skraćenom zapisu ne pišemo (ali podrazumijevamo) $\sqcup \dots$ na kraju.

Koristeći tu notaciju, početna konfiguracija \mathcal{T}_h s ulazom $w_1 := \text{abaa}$ je abaa . Tada je \mathcal{T}_h -izračunavanje s w_1

$$\begin{aligned} & \text{abaa} \rightsquigarrow \text{caaa} \rightsquigarrow \text{cbaa} \rightsquigarrow \text{cbaa} \rightsquigarrow \text{cbaa} \rightsquigarrow \text{cbaa} \rightsquigarrow \text{cba} \rightsquigarrow \text{cba} \rightsquigarrow \text{cba} \rightsquigarrow \\ & \rightsquigarrow \text{aba} \rightsquigarrow \text{ada} \rightsquigarrow \text{ada} \rightsquigarrow \text{ada} \rightsquigarrow \text{ad} \rightsquigarrow \text{ab} \rightsquigarrow \text{ab} \rightsquigarrow \text{ab} \rightsquigarrow \dots, \end{aligned} \quad (4.4)$$

iz čega se vidi izlazni podatak $\text{ab} = \varphi_h(w_1)$. \mathcal{T}_h -izračunavanje s $w_2 := \text{aba}$ je

$$\begin{aligned} & \text{aba} \rightsquigarrow \text{cba} \rightsquigarrow \text{cba} \rightsquigarrow \text{cba} \rightsquigarrow \text{cba} \rightsquigarrow \text{cb} \rightsquigarrow \text{cb} \rightsquigarrow \\ & \rightsquigarrow \text{ab} \rightsquigarrow \text{ad} \rightsquigarrow \text{ad} \rightsquigarrow \text{ad} \rightsquigarrow \text{ad} \rightsquigarrow \text{ad} \rightsquigarrow \text{ad} \rightsquigarrow \dots, \end{aligned} \quad (4.5)$$

pa to izračunavanje ne stane, iz čega vidimo $\text{aba} \notin \mathcal{D}_{\varphi_h}$. ◁

4.2. Prateće funkcije jezičnih funkcija

Kao što je već najavljeno, cilj je pokazati ekvivalentnost Turing-modela s RAM-modelom, odnosno funkcijskim modelom izračunljivosti. Za početak ćemo, koristeći pristup sličan onome iz poglavlja 3, dokazati da su Turing-izračunljive jezične funkcije „parcijalno rekurzivne”.

Da bismo to dokazali, moramo prvo precizirati što to uopće znači. Neka je Σ abeceda, i φ jezična funkcija nad njom. Očito samu funkciju φ ne možemo dobiti kompozicijom, primitivnom rekurzijom i minimizacijom iz inicijalnih (brojevnih) funkcija. Čak i da smislimo neke odgovarajuće „inicijalne jezične funkcije”, kompozicija je jasna (čak jednostavnija nego u brojevnom slučaju, jer su sve funkcije jednomjesne), ali kako definirati primitivnu rekurziju kad riječ nema jedinstvenog sljedbenika? Kako definirati minimizaciju kad kanonski leksikografski uređaj na Σ^* nije dobar uređaj? Kako uopće definirati izračunljive jezike („relacije”) kad karakteristična funkcija jezika nije ni brojevena ni jezična funkcija?

Sve su to problemi o kojima smo već govorili, ponajviše u uvodu. No sada znamo dovoljno o kodiranju da nas to ne treba previše obeshrabriti. Kodirat ćemo Σ^* nekom funkcijom $\mathbb{N}\Sigma^*$ (po uzoru na kodiranje \mathbb{N}^* , označavamo $\langle w \rangle := \mathbb{N}\Sigma^*(w)$), te ćemo pomoću tog kodiranja definirati prateću funkciju $\mathbb{N}\varphi = \mathbb{N}\Sigma^* \circ \varphi \circ (\mathbb{N}\Sigma^*)^{-1}$, za koju je dobro definirano što znači da je parcijalno rekurzivna. Štoviše, pazit ćemo da $\mathbb{N}\Sigma^*$ bude *bijekcija* između Σ^* i \mathbb{N} , tako da će prateće funkcije totalnih funkcija doista biti totalne funkcije. To je bitno jer će onda *rekurzivne* jezične funkcije biti dobro definirane, kao totalne izračunljive jezične funkcije, odnosno one čije su prateće funkcije rekurzivne.

4.2.1. Kodiranje znakova i riječi

Prvo recimo nešto o kodiranju abecede Σ . Kako je to konačan skup, iskoristit ćemo enum-tehniku, kao što smo već napravili kod kodiranja tipova instrukcija RAM-stroja. Ipak, ovdje ćemo početi brojiti od 1 (ne želimo 0 u slici od $\mathbb{N}\Sigma$), iz tehničkog razloga koji će uskoro biti jasan.

Još jedan mali filozofski problem je u tome što su elementi od Σ „apstraktni znakovi” bez ikakve imanentne semantike i međusobnog odnosa, tako da ne možemo fiksirati jedan poredak kao što smo to učinili za tipove RAM-instrukcija. Ipak, za svaku konkretnu abecedu moći ćemo fiksirati kodiranje (*encoding*), a za apstraktne abecede moći ćemo univerzalno kvantificirati tvrdnje u obliku „Za svako kodiranje od $\Sigma \dots$ ”, podrazumijevajući da je očito („Zermelov teorem za konačne skupove”) da kodiranje *postoji*. U praksi, ti znakovi će obično biti dio standarda Unikod, te ćemo ih moći urediti, ako baš nemamo pametniji kriterij, po rednim brojevima u Unikodu.

Definicija 4.7. Neka je Σ abeceda. Označimo $b' := \text{card } \Sigma$. *Kodiranje* od Σ je bilo koja bijekcija $\mathbb{N}\Sigma: \Sigma \leftrightarrow [1..b']$. ◁

Sada bismo mogli, kao za kodiranje RAM-programa, jednostavno kodirati riječi kao kodove konačnih nizova kodova njihovih znakova — ali to je nezgrapno (i ne bi bilo bijekcija). Naime, kodovi RAM-instrukcija mogli su biti proizvoljno veliki, dok za fiksnu abecedu, kodovi znakova mogu biti najviše b' . To sugerira da nam je dovoljno jednostavnije kodiranje.

Najprirodnije kodiranje ograničenih nizova, toliko prirodno da možda nismo ni svjesni kako ga koristimo (za $b' = 10$) svaki put kad čitamo i pišemo višeznamenaste brojeve, je **zapis u bazi** b' . Ipak, s njime postoje dva problema. Prvi problem su početne nule ($((001121)_3 = (1121)_3$), što smo riješili tako što smo počeli kodirati znakove od 1, ali smo na račun toga dobili uključenu gornju granicu. Kako opravdati znamenku b' u bazi b' ? Drugi problem su jednočlane abecede: znamo da se obični pozicijski brojevni sustavi definiraju samo za baze od 2 nadalje. Da, postoji i unarni zapis, ali on nije pozicijski — ili jest?

Zapravo, unarni zapis sasvim odgovara uobičajenom brojevnom zapisu u bazi b' za $b' = 1$: recimo, $(1111)_1 = 1 \cdot 1^3 + 1 \cdot 1^2 + 1 \cdot 1^1 + 1 \cdot 1^0 = 4$, i jedini problem je uključena gornja granica — u bazi 1 imamo znamenku 1. Ali nemamo znamenku 0, i ne smijemo je imati ako želimo jedinstven zapis — no to upravo rješava prvi problem.

Definicija 4.8. Neka je $b \in \mathbb{N}_+$, $n \in \mathbb{N}$, te $z_0, z_1, \dots, z_{n-1} \in [1..b]$. Zapis u *pomaknutoj bazi* b je zapis

$$x = \sum_{i < n} z_i \cdot b^i =: (z_{n-1} \dots z_0)_b, \quad (4.6)$$

dakle isti kao obični zapis u bazi b , samo sa znamenkama iz $[1..b]$ umjesto iz $[0..b]$. \triangleleft

Lema 4.9. Za svaki $b \in \mathbb{N}_+$, svaki $x \in \mathbb{N}$ ima jedinstven zapis u *pomaknutoj bazi* b .

Dokaz. Dokaz egzistencije je isti kao i dokaz da svaki broj ima zapis u običnoj bazi b : znamenke konstruiramo tako da uzastopno dijelimo x s b dok ne dobijemo nulu, i ostatke tih dijeljenja zapišemo obrnutim redom. Jedina razlika je što ovdje ostatci moraju biti između 1 i b : ako x nije djeljiv s b , ništa se ne mijenja, a ako jest, smanjimo količnik za 1. Recimo, 18 podijeljeno s 3 je 5 i pomaknuti ostatak 3. Pomaknuti ostatak je primitivno rekurzivna operacija, po teoremu o grananju.

$$\text{mod}'(x, b) := \begin{cases} b, & b \mid x \\ x \bmod b, & \text{inače} \end{cases} \quad (4.7)$$

Za dokaz jedinstvenosti, treba vidjeti da brojevi različitih duljina zapisa u istoj bazi moraju biti različiti, te da brojevi iste duljine zapisa koji se u nekoj znamenci razlikuju također moraju biti različiti. Obje nejednakosti posljedica su od

$$(tz_{m-1} \dots z_0)_b \leq (tb \dots b)_b < ((t+1)1 \dots 1)_b \leq ((t+1)z'_{m-1} \dots z'_0)_b, \quad (4.8)$$

— za prvu je $t = 0$, a za drugu je t prva znamenka slijeva u kojoj se zapisi razlikuju. Sama stroga nejednakost u (4.8) lako se dobije:

$$\begin{aligned} t \cdot b^m + b \cdot b^{m-1} + b \cdot b^{m-2} + \dots + b &= t \cdot b^m + b^m + b^{m-1} + \dots + b < \\ &< (t+1)b^m + 1 \cdot b^{m-1} + \dots + 1 \cdot b + 1. \end{aligned} \quad (4.9)$$

To po tranzitivnosti zapravo znači da smo zapise poredali po duljini, a zapise iste duljine leksikografski (tzv. *shortlex ordering*). \square

Primjer 4.10. *Shortlex ordering* $\{a, b\}^*$ je: $\varepsilon, \underset{0}{a}, \underset{1}{b}, \underset{2}{aa}, \underset{3}{ab}, \underset{4}{ba}, \underset{5}{bb}, \underset{6}{aaa}, \underset{7}{aab}, \underset{8}{aba}, \dots$ \triangleleft

Definicija 4.11. Neka je Σ abeceda, te $\mathbb{N}\Sigma$ njeno kodiranje. Definiramo kodiranje skupa Σ^* svih riječi nad Σ , s

$$\mathbb{N}\Sigma^*(w) := \langle \alpha_{n-1} \dots \alpha_0 \rangle := (\mathbb{N}\Sigma(\alpha_{n-1}) \dots \mathbb{N}\Sigma(\alpha_0))_{b'} = \sum_{i < n} \mathbb{N}\Sigma(\alpha_i) \cdot (b')^i, \quad (4.10)$$

za svaku riječ $w = \alpha_{n-1} \dots \alpha_0 \in \Sigma^*$ (uz oznake $n := |w|$ i $b' := \text{card } \Sigma$). \triangleleft

Primjer 4.12. U primjeru 4.6 uveli smo abecedu $\Sigma = \{a, b\}$ i riječ nad njome $w_0 = aababa$. Za tu abecedu je $b' = 2$, te fiksirajmo kodiranje $\mathbb{N}\Sigma(a) := 1$, $\mathbb{N}\Sigma(b) := 2$. Tada je $\langle w_0 \rangle = (112121)_2 = 73$. Također je $w_0 \in \mathcal{D}_{\varphi_h}$, pa je $73 \in \mathcal{D}_{\mathbb{N}\varphi_h}$; i lako vidimo $\varphi_h(w_0) = aab$, pa je $\mathbb{N}\varphi_h(73) = \langle aab \rangle = (112)_2 = 8$.

S druge strane, recimo, uzmimo broj 153. Pretvaranje u pomaknutu bazu 2 daje $\frac{153 \ 76 \ 37 \ 18 \ 8 \ 3 \ 1}{1 \ 2 \ 1 \ 2 \ 2 \ 1 \ 1}$, dakle $153 = (1122121)_2 = \langle aabbaba \rangle$. Kako je $|aabbaba| = 7$ neparan broj, zaključujemo $aabbaba \notin \mathcal{D}_{\varphi_h}$, pa $153 \notin \mathcal{D}_{\mathbb{N}\varphi_h}$. \triangleleft

Za dekodiranje trebamo, analogno kao za kodiranje \mathbb{N}^* (propozicija 3.14), duljinu zapisa u određenoj bazi, te određenu znamenku.

Lema 4.13. *Postoje primitivno rekurzivne funkcije slh^2 , sdigit^3 , sconcat^3 i sprefix^3 , takve da za sve $b \in \mathbb{N}_+$, za sve $i, n, m \in \mathbb{N}$ vrijedi:*

$\text{slh}(n, b)$ je duljina zapisa broja n u pomaknutoj bazi b .

$\text{sdigit}(n, i, b)$ za $i < \text{slh}(n, b)$, je vrijednost i -te znamenke zapisa broja n u pomaknutoj bazi b , gdje brojimo od nule slijeva.

$\text{sconcat}(m, n, b)$ je broj čiji zapis u pomaknutoj bazi b se dobije konkatencijom istih takvih zapisa za m i n redom. Pišemo ga i kao $m \frown_b n$.

$\text{sprefix}(n, i, b)$ za $i \leq \text{slh}(n, b)$, je broj čiji je zapis u pomaknutoj bazi b prefiks (početak) duljine i istog takvog zapisa broja n .

$$\text{slh}(n, b) := (\mu t \leq n) \left(\sum_{i \leq t} b^i > n \right) \quad (4.11)$$

$$\text{sconcat}(m, n, b) := m \frown_b n := m \cdot b^{\text{slh}(n, b)} + n \quad (4.12)$$

$$\text{sprefix}(n, i, b) := (\mu m \leq n) (\exists t \leq n) (n = m \frown_b t \wedge \text{slh}(m, b) = i) \quad (4.13)$$

$$\text{sdigit}(n, i, b) := \text{mod}'(\text{sprefix}(n, \text{Sc}(i), b), b) \quad (4.14)$$

Dokaz. Za slh , treba uočiti da zapisa duljine i ima točno b^i . Dakle, samo trebamo zbrajati takve brojeve dok ne prijeđemo n . Također, očito je $\text{slh}(n, b) \leq n$. (4.11)

Sada jednostavno možemo dobiti konkatenciju: m pomaknemo udesno da n taman stane u dobiveni prostor, i samo ga pribrojimo. (4.12)

Pomoću konkatencije možemo dobiti sprefix : prefiks je ono što se može konkatencirati s nečim tako da se dobije početni zapis n . Očito su dijelovi manji ili jednaki n . (4.13)

Sada nije teško napisati ni sdigit : to je posljednja znamenka prefiksa čija je duljina za jedan veća od pozicije znamenke. (4.14) \square

Propozicija 4.14. *Neka je Σ abeceda, i $\mathbb{N}\Sigma$ njeno kodiranje. Tada je $\mathbb{N}\Sigma^*$ bijekcija između Σ^* i \mathbb{N} .*

Dokaz. Označimo $b' := \text{card } \Sigma$. Ako je $w \neq w'$ za $w, w' \in \Sigma^*$, tada su w i w' ili različitih duljina (pa je $\text{slh}(\langle w \rangle, b') \neq \text{slh}(\langle w' \rangle, b')$), ili su jednake duljine ali se na nekom mjestu razlikuju (pa je za neki i , $\text{sdigit}(\langle w \rangle, i, b') \neq \text{sdigit}(\langle w' \rangle, i, b')$).

U svakom slučaju mora biti $\langle w \rangle \neq \langle w' \rangle$, dakle $\mathbb{N}\Sigma^*$ je injekcija.

Za surjektivnost, neka je $x \in \mathbb{N}$ proizvoljan. Prema lemi 4.9, postoje znamenke $\bar{z} \in [1..b']^*$ takve da je $x = (\bar{z})_{b'}$. Ako sad za svaku z_i označimo $\alpha_i := \mathbb{N}\Sigma^{-1}(z_i) \in \Sigma$, riječ sastavljena od tih znakova ima upravo kod x . \square

4.2.2. Parcijalna rekurzivnost jezičnih funkcija

Definicija 4.15. Neka je Σ abeceda, $\mathbb{N}\Sigma$ njeno kodiranje, te $\varphi: \Sigma^* \rightarrow \Sigma^*$ jezična funkcija nad njom. *Prateća funkcija* od φ je jednomjesna funkcija $\mathbb{N}\varphi$ s domenom $\mathbb{N}\Sigma^*[\mathcal{D}_\varphi]$, zadana s

$$\mathbb{N}\varphi(\langle w \rangle) \simeq \langle \varphi(w) \rangle. \quad (4.15)$$

Kako je svaki prirodni broj kod jedinstvene riječi iz Σ^* , definicija je dobra. \triangleleft

Kroz čitavu ovu točku, imat ćemo fiksiranu abecedu Σ_0 , njeno kodiranje $\mathbb{N}\Sigma_0$ uz oznaku $b' := \text{card } \Sigma_0$, Turing-izračunljivu jezičnu funkciju φ_0 nad Σ_0 , te fiksni Turingov stroj $\mathcal{T}_0 = (Q_0, \Sigma_0, \Gamma_0, \sqcup, \delta_0, q_0, q_z)$ koji računa φ_0 . Cilj će nam biti, kodirajući komponente i izračunavanje stroja \mathcal{T}_0 , dokazati da je $\mathbb{N}\varphi_0$ parcijalno rekurzivna funkcija. To je slično pristupu u poglavlju 3, samo je jednostavnije jer ne moramo pisati interpreter za proizvoljni RAM-program zadan kodom, već samo „ručno” prevesti jedan konkretni Turingov stroj \mathcal{T}_0 u funkcijski jezik.

Prvo kodirajmo skup Q_0 . Kako se radi o konačnom skupu, možemo jednostavno staviti $a := \text{card } Q_0$ i fiksirati bijekciju $\mathbb{N}Q_0: Q_0 \leftrightarrow [0..a]$. Za konstruktor trebamo samo fiksirati kod početnog stanja — prirodnim se čini definirati $\mathbb{N}Q_0(q_0) := 0$ — a što se komponenata tiče, sve što trebamo je usporedba s q_z , što ćemo dobiti ako i njegov kod bude fiksiran broj — recimo, $\mathbb{N}Q_0(q_z) := 1$.

Hm, hoće li onda $\mathbb{N}Q_0$ biti dobro definirana — što ako je $q_0 = q_z$? U definiciji Turingovog stroja ništa ne sprečava da se to dogodi. Ipak, lako se vidi da $q_0 \neq q_z$ smijemo pretpostaviti bez smanjenja općenitosti.

Lema 4.16. *Za svaki Turingov stroj \mathcal{T} koji računa neku jezičnu funkciju φ , postoji ekvivalentan (računa istu funkciju φ) Turingov stroj \mathcal{T}' , kojem se početno i završno stanje razlikuju.*

Dokaz. Ako već u \mathcal{T} vrijedi $q_0 \neq q_z$, stavimo $\mathcal{T}' := \mathcal{T}$. Inače, vrijedi $q_0 = q_z$, i tvrdimo da je tada φ identiteta na Σ^* . Doista, za svaku riječ $w \in \Sigma^*$, početna konfiguracija stroja \mathcal{T} s ulazom w , $(q_0, 0, w_\sqcup \dots) = (q_z, 0, w_\sqcup \dots)$, ujedno je i završna konfiguracija, te izračunavanje uvijek stane (φ je totalna) i iz završnog oblika trake čitamo $\varphi(w) = w$.

Dakle, sad samo trebamo konstruirati Turingov stroj s različitim početnim i završnim stanjem, koji računa identitetu. To doista nije teško: jedan takav je

$$\mathcal{T}' := (\{0, 1\}, \Sigma, \Sigma \cup \{\sqcup\}, \sqcup, \delta', 0, 1), \quad (4.16)$$

$$\delta'(0, \alpha) := (1, \alpha, 1) \text{ za sve } \alpha \in \Sigma \cup \{\sqcup\}. \quad (4.17)$$

Tada za svaku $w \in \Sigma^*$ imamo \mathcal{T}' -izračunavanje s w : $(0, 0, w_{\sqcup} \dots) \rightsquigarrow (1, 1, w_{\sqcup} \dots) \rightsquigarrow (1, 1, w_{\sqcup} \dots) \rightsquigarrow \dots$, te je izlazni podatak opet w , odnosno \mathcal{T}' računa identitetu. \square

Slično možemo kodirati i Γ_0 — ali kako već imamo kodiranje skupa $\Sigma_0 \subset \Gamma_0$, želimo da znakovi ulazne abecede imaju iste kodove. Dakle, označimo $b := \text{card } \Gamma_0$, i proširimo bijekciju $\mathbb{N}\Sigma_0: \Sigma_0 \leftrightarrow [1..b']$ na bijekciju $\mathbb{N}\Gamma_0: \Gamma_0 \leftrightarrow [0..b]$, tako da \sqcup preslikamo u 0, a ostale elemente iz $\Gamma_0 \setminus \Sigma_0$ bijektivno u skup $[b'..b]$.

Primijetite da smo sada upotrijebili nulu, i to kao $\mathbb{N}\Gamma_0(\sqcup)$. To pruža opravdanje za frazu „traka je s konačnim nosačem” (praznine su kodirane nulama, pa ne-nula ima konačno mnogo), a bit će esencijalno i za kodiranje trake.

Naime, sada imamo sličan problem kao sa stanjem registara RAM-stroja: da bismo kodirali proizvoljnu traku, moramo nekako skupiti kodove beskonačno mnogo ćelija, ali takve da su svi osim konačno mnogo njih jednaki 0. Alternativno, želimo „kodiranje” konačnih nizova, ali tako da dodavanje nule na kraj ne promijeni kod.

U slučaju RAM-registara to smo riješili rastavom na prim-faktore, odnosno malom modifikacijom kodiranja \mathbb{N}^* — umjesto od 1, eksponenti su išli od 0. Možemo li ovdje naći neku malu modifikaciju kodiranja Σ^* (zapis u bazi) da dobijemo analogni rezultat?

Zapravo da, i slična ideja funkcionira: kontraprimjer za injektivnost preslikavanja koje broji znakove od 0 bit će upravo putokaz kako treba napisati kodiranje trake. Umjesto od 1, znamenke će ići od 0. Tamo smo rekli da je $(001121)_3 = (1121)_3$, no to upravo znači da je taj broj prirodno gledati kao kod trake $\text{abaa}_{\sqcup\sqcup\sqcup} \dots = \text{abaa}_{\sqcup} \dots$. Drugim riječima, ovdje imamo obični zapis u (ne pomaknutoj) bazi b .

Definicija 4.17. Za proizvoljnu traku $t: \mathbb{N} \rightarrow \Gamma_0$ (takvu da je skoro svuda \sqcup), definiramo kod trake

$$\|t\| = \|t_0 t_1 t_2 \dots \sqcup \sqcup \dots\| := \sum_{i \in \mathbb{N}} \mathbb{N}\Gamma_0(t_i) \cdot b^i. \quad (4.18)$$

Zbog konačnog nosača i činjenice da je $\mathbb{N}\Gamma_0(\sqcup) = 0$, samo konačno mnogo članova tog „reda potencija” bit će pozitivno, pa je suma dobro definirana. \triangleleft

S ovom idejom postoje dvije smetnje. Prva je što, očito, moramo promijeniti bazu iz b' u b , jer na traci se mogu naći i znakovi s većim kodovima. Druga je posljedica nesretne povijesne okolnosti da u zapadnom svijetu pišemo riječi slijeva nadesno (i tako doživljavamo traku), ali smo zapis brojeva preuzeli iz jezika (arapskog) koji se piše u suprotnom smjeru. Zato dodavanje nule *slijeva* u zapis broja u bazi b odgovara dodavanju prazne ćelije *zdesna* pri pomaku udesno od zadnje posjećene ćelije.

Iskusniji računarci, posebno oni s iskustvom mrežnog programiranja, znaju da taj problem postoji i u njihovom svijetu. Radi se o *byteorder*-dilemi, sasvim analognoj upravo opisanom problemu, iz sličnih razloga. Većina modernih procesora pamti višebajtnu podatke tako da na početnoj adresi podatka stoji najmanje značajni bajt, jer se na taj način semantika *casta* pokazivača podudara sa semantikom *casta* pokazanih vrijednosti. Nakon `int x=0x61626364;`, izraz `(char)x` ima vrijednost `0x64`, te bi bilo logično očekivati da i `*(char*)&x` ima istu vrijednost. (Gledano s druge strane, nakon `y=25` htjeli bismo da se na adresi `&y` nalazi bajt 25, neovisno o tome kojeg je `y` tipa.) No to je jedino moguće ako se `x` stavlja na adrese `[p..p + sizeof x)` tako da se na adresu `p` stavi bajt `0x64`, na adresu `p + 1` bajt `0x63` (za *cast* u *short*), ..., odnosno „obrnutim” redom (*little-endian*).

Naravno, „obrnutim” u odnosu na to kako smo navikli pisati brojeve, i kako smo uostalom napisali taj broj u naredbi koja inicijalizira `x`. To posebno dolazi do izražaja u mrežnom programiranju, jer mrežni standardi propisuju da se višebajtni brojevi preko mreže prenose u *big-endian* redoslijedu bajtova, kako bi bilo lakše pratiti što se događa u slučaju greške. Zato mnoge standardne biblioteke imaju funkcije za pretvaranje (*ntoh/hton*) redoslijeda bajtova iz mrežnog (koji je lakše čitati) u procesorski (s kojim je lakše računati) i obrnuto. Mi ćemo napraviti nešto slično, i usput pretvoriti baze.

Lema 4.18. *Postoji primitivno rekurzivna funkcija Recode^3 takva da za svaku riječ $w \in \Sigma_0^*$ vrijede jednakosti*

$$\text{Recode}(\langle w \rangle, b', b) = \|w_{\sqcup} \dots\|, \quad (4.19)$$

$$\text{Recode}(\|w_{\sqcup} \dots\|, b, b') = \langle w \rangle. \quad (4.20)$$

Dokaz. Kao što smo već rekli, $\text{Recode}(n, b_1, b_2)$ samo treba ekstrahirati znamenke od `n` u bazi `b1`, i slagati ih obrnutim redom u bazi `b2`. Kako je $w \in \Sigma_0^*$, u zapisu nema znamenke 0, pa možemo koristiti funkcije `slh` i `sdigit` bez obzira na to je li baza `b1` pomaknuta.

$$\text{Recode}(n, b_1, b_2) := \sum_{i < \text{slh}(n, b_1)} \text{sdigit}(n, i, b_1) \cdot b_2^i \quad (4.21)$$

Neka je $w = \alpha_0 \alpha_1 \dots \alpha_{l-1} \in \Sigma^*$ proizvoljna ($l := |w|$). Tada, ako označimo $n := \langle w \rangle$, vrijedi $\text{slh}(n, b') = l$ i $d_i := \text{sdigit}(n, i, b') = \mathbb{N}\Sigma_0(\alpha_i) = \mathbb{N}\Gamma_0(\alpha_i)$ za sve $i < l$. Iz toga

$$\text{Recode}(n, b', b) = \sum_{i < l} d_i \cdot b^i = \sum_{i=0}^{l-1} \mathbb{N}\Gamma_0(\alpha_i) \cdot b^i = \|w_{\sqcup} \dots\|, \quad (4.22)$$

i analogno (4.20). Ključno je bilo da se $\mathbb{N}\Sigma_0$ i $\mathbb{N}\Gamma_0$ podudaraju na svim $\alpha \in \Sigma_0$.

Za dokaz primitivne rekurzivnosti, samo treba primijeniti napomenu 2.51, lemu 4.13, te primjer 2.11. \square

Primjer 4.19. Neka je $\Sigma := \{a, b, c\} \subset \Gamma := \{\sqcup, a, b, c, A, B, C\}$, neka je kodiranje zadano redom kojim su napisani znakovi, te neka je $w := \text{bbc}$. Tada je $\langle w \rangle = (223)_3 = 27$, a $\|w_{\sqcup} \dots\| = (322)_7 = 163$, dakle $\text{Recode}(27, 3, 7) = 163$ i $\text{Recode}(163, 7, 3) = 27$. \triangleleft

4.2.3. Kodiranje funkcije prijelaza i Turing-izračunavanja

Mogli bismo se uplašiti, znajući koliko kodiranje funkcija može biti komplicirano (sjetite se indeksa). Ali δ_0 je *konačna* funkcija (*lookup table*), pa zapravo neće biti problema.

Prvo, δ_0 je funkcija s dva ulaza i tri izlaza, tako da je zapravo kodiramo kroz tri koordinatne funkcije. Drugo, već imamo kodiranja za ulaze i prva dva izlaza, $\mathbb{N}Q_0: Q_0 \leftrightarrow [0..a]$ i $\mathbb{N}\Gamma_0: \Gamma_0 \leftrightarrow [0..b]$, samo još trebamo kodirati pomake. Kako oni već jesu „numerički” u \mathbb{Z} , najlakše ih je samo povećati za 1 da upadnu u \mathbb{N} , tako da „pomak ulijevo” -1 kodiramo brojem 0, a „pomak udesno” 1 kodiramo brojem 2.

Lema 4.20. *Postoje primitivno rekurzivne dvomjesne funkcije **newstate**, **newsymbol** i **direction** koje preslikavaju $(\mathbb{N}Q_0(q), \mathbb{N}\Gamma_0(\gamma))$ redom u $\mathbb{N}Q_0(q')$, $\mathbb{N}\Gamma_0(\gamma')$ i $1 + d$, gdje*

$$je (q', \gamma', d) := \begin{cases} \delta_0(q, \gamma), & q \neq q_z \\ (q, \gamma, 0), & \text{inače} \end{cases}.$$

Dokaz. Za proizvoljni par $(k, g) \in [0..a] \times [0..b]$, ako je $k \neq 1$, označimo $(q', \gamma', d) := \delta_0(\mathbb{N}Q_0^{-1}(k), \mathbb{N}\Gamma_0^{-1}(g))$, i definirajmo **newstate**(k, g) := $\mathbb{N}Q_0(q')$, **newsymbol**(k, g) := $\mathbb{N}\Gamma_0(\gamma')$, i **direction**(k, g) := $1 + d$. Također, za svaki $g \in [0..b]$, dodefinirajmo **newstate**(1, g) := **direction**(1, g) := 1 i **newsymbol**(1, g) := g .

Koliko god to komplicirana pravila bila, njima definirane funkcije su konačne (sve tri imaju domen $[0..a] \times [0..b]$ s ab elemenata), pa prema korolaru 2.48 za svaku od njih postoji primitivno rekurzivna funkcija koja se s njom podudara na $[0..a] \times [0..b]$ (proširenje nulom). Sada je iz definicije tih triju funkcija jasno da vrijedi tvrdnja leme (sjetimo se, $1 = \mathbb{N}Q_0(q_z)$, te su $[0..a]$ i $[0..b]$ slike od $\mathbb{N}Q_0$ i $\mathbb{N}\Gamma_0$ redom). \square

Primjer 4.21. U primjeru 4.6 smo vidjeli funkciju prijelaza δ_h , definiranu s (4.2). Ako stanja i znakove kodiramo kao $\frac{Q_h}{\mathbb{N}Q_h} \begin{array}{c|c} A & B & C & D & E & F \end{array} \begin{array}{c} 0 \\ 3 \\ 4 \\ 5 \\ 1 \\ 2 \end{array}$ i $\frac{\Gamma_h}{\mathbb{N}\Gamma_h} \begin{array}{c|c} a & b & c & d & \sqcup \end{array} \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 0 \end{array}$, tada su funkcije **newstate**, **newsymbol** i **direction** redom zadane sljedećim tablicama:

	0	1	2	3	4	5	...		0	1	2	3	4	5	...		0	1	2	3	4	5	...
0	1	3	3	2	2	0	...	0	0	3	4	3	4	0	...	0	2	2	2	2	2	0	...
1	1	1	1	1	1	0	...	1	0	1	2	3	4	0	...	1	1	1	1	1	1	0	...
2	2	2	2	2	2	0	...	2	0	1	2	3	4	0	...	2	2	2	2	2	2	0	...
3	4	3	3	2	2	0	...	3	0	1	2	3	4	0	...	3	0	2	2	2	2	0	...
4	2	5	5	2	2	0	...	4	0	0	0	3	4	0	...	4	2	0	0	2	2	0	...
5	2	5	5	0	0	0	...	5	0	1	2	1	2	0	...	5	2	0	0	2	2	0	...
6	0	0	0	0	0	0	...	6	0	0	0	0	0	0	...	6	0	0	0	0	0	0	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱	⋮	⋮	⋮	⋮	⋮	⋮	⋱	

Δ

Upravo dokazana lema je donekle analogna lemi 3.32. Vrijeme je za analogon leme 3.33. Za razliku od tih lema, koje su definirale *uniformne* funkcije koje su

primale RAM-program kao argument, ove će se odnositi na fiksni Turingov stroj \mathcal{T}_0 — jer je tako lakše, a ne treba nam puna općenitost; sve što će nam kasnije trebati su neki indeksi izračunljivih funkcija, a njih smo dobili s RAM-strojevima.

Lema 4.22. *Funkcije definirane sa*

$$\text{State}(\langle w \rangle, n) := \mathbb{N}Q_0(q_n), \quad \text{Position}(\langle w \rangle, n) := p_n, \quad \text{Tape}(\langle w \rangle, n) := \|t_n\|, \quad (4.23)$$

gdje je $((q_n, p_n, t_n))_{n \in \mathbb{N}}$ \mathcal{T}_0 -izračunavanje s w , primitivno su rekurzivne.

Dokaz. Kao i kod RAM-izračunavanja, upotrijebit ćemo simultanu primitivnu rekurziju. Za početak, trebaju nam funkcije G_1 , G_2 i G_3 , koje daju njihove vrijednosti za $n = 0$. Stanje počinje od q_0 koje smo kodirali nulom, pozicija počinje od nule (lijevi rub), a traka počinje od $w_\square \dots$, čiji kod se iz koda $x = \langle w \rangle$ može dobiti primjenom funkcije *Recode*.

$$G_1(x) := G_2(x) := 0, \quad G_3(x) := \text{Recode}(x, b', b). \quad (4.24)$$

Dakle, $G_1 = G_2 = Z$ (inicijalna funkcija), a G_3 je primitivno rekurzivna po lemi 4.18 i propoziciji 2.19.

Za korak, trebamo funkcije H_1 , H_2 i H_3 , od kojih svaka prima po pet argumenata: kôd riječi s kojom se računa $x = \langle w \rangle$, broj dosad napravljenih koraka izračunavanja n , kôd prethodnog stanja $q = \mathbb{N}Q_0(q_n)$, prethodnu poziciju $p = p_n$, i kôd prethodne trake $t = \|t_n\|$. Redom trebaju vratiti $\mathbb{N}Q_0(q_{n+1})$, p_{n+1} i $\|t_{n+1}\|$. Pri tome će nam pomoći funkcije (redom) *newstate*, *direction* i *newsymbol*, ali na čemu ih pozvati? Prvi argument je kod trenutnog stanja q , a drugi je kod trenutno čitanog znaka g , koji možemo dobiti kao vrijednost znamenke od t mjesne vrijednosti b^p . (Tu ne možemo koristiti *sdigit* jer se na traci mogu nalaziti i praznine, te se mjesne vrijednosti broje zdesna u zapisu — ali standardni postupak za ekstrakciju znamenke funkcionira.)

Sada H_1 jednostavno treba vratiti *newstate*(q, g). H_2 treba vratiti p pomaknut za *direction*(q, g), ali za jedan manje jer smo pomake za d kodirali s $d + 1$. H_3 je najkompliciranija: treba p -tu znamenku u zapisu t u bazi b zamijeniti sa znamenkom vrijednosti *newsymbol*(q, g). To radimo tako da od t oduzmemo trenutnu mjesnu vrijednost te znamenke $g \cdot b^p$, i dodamo mjesnu vrijednost nove. Dakle, primitivno rekurzivne funkcije zadane s

$$H_1(x, n, q, p, t) := \text{newstate}(q, g), \quad (4.25)$$

$$H_2(x, n, q, p, t) := p - \text{direction}(q, g), \quad (4.26)$$

$$H_3(x, n, q, p, t) := t - g \cdot b^p + \text{newsymbol}(q, g) \cdot b^p, \quad (4.27)$$

$$\text{uz pokratu } g := t // b^p \bmod b, \quad (4.28)$$

zadovoljavaju parcijalnu specifikaciju koju smo napisali. Sada su *State*, *Position* i *Tape* definirane simultanom primitivnom rekurzijom iz G_1 , G_2 , G_3 , H_1 , H_2 i H_3 , te su

primitivno rekurzivne. Tvrdnju leme sada dokazujemo matematičkom indukcijom po n , sasvim analogno kao u RAM-modelu.

Za $n = 0$, početno stanje je q_0 , pa je $\text{State}(x, 0) = G_1(x) = Z(x) = 0 = \mathbb{N}Q_0(q_0)$ doista njegov kod. Početna pozicija je $0 = Z(x) = G_2(x) = \text{Position}(x, 0)$. Početna traka je $w_\square \dots$, čiji kod je $\|w_\square \dots\|$, što je prema (4.19) jednako $\text{Recode}(\langle w \rangle, b', b) = G_3(\langle w \rangle) = \text{Tape}(\langle w \rangle, 0)$.

Sada pretpostavimo da je nakon k koraka, $q := \text{State}(\langle w \rangle, k)$ kod stanja, $p := \text{Position}(\langle w \rangle, k)$ pozicija, a $t := \text{Tape}(\langle w \rangle, k)$ kod trake stroja. Ako je ta konfiguracija završna, tada je $q = \mathbb{N}Q_0(q_z) = 1$, pa prema definicijama iz leme 4.20 vrijedi $\text{newstate}(q, g) = \text{direction}(q, g) = 1$ i $\text{newsymbol}(q, g) = g$. Tada H_1 daje q , H_2 daje $\text{pd}(p + 1) = \text{pd}(\text{Sc}(p)) = p$, a H_3 daje $t \ominus g \cdot b^p + g \cdot b^p = t$ (jer je $g = t \parallel b^p \bmod b \leq t \parallel b^p \leq \frac{t}{b^p}$, pa je $g \cdot b^p \leq t$, odnosno \ominus je zapravo $-$). Dakle $\text{State}(\langle w \rangle, k + 1) = \text{State}(\langle w \rangle, k)$, i analogno za Position i Tape , odnosno završna konfiguracija ostaje ista, kao što i treba.

Ako ta konfiguracija nije završna, tad je $q \in [0..a) \setminus \{1\}$, pa su vrijednosti newstate , newsymbol i direction zadane preko funkcije δ_0 , i predstavljaju upravo novo stanje, novi znak na trenutnoj poziciji, i pomak na novu poziciju. Recimo, ako s g označimo kod trenutno čitanog znaka, a s d treću komponentu odgovarajuće vrijednosti $\delta_0(q, g)$, tada je $\text{direction}(q, g) = d + 1$, pa je

$$\begin{aligned} \text{Position}(\langle w \rangle, k + 1) &= H_2(\langle w \rangle, k, \text{State}(\langle w \rangle, k), \text{Position}(\langle w \rangle, k), \text{Tape}(\langle w \rangle, k)) = \\ &= H_2(\langle w \rangle, k, q, p, t) = \text{pd}(p + \text{direction}(q, g)) = \text{pd}(p + d + 1) = \\ &= \max\{p + d + 1 - 1, 0\} = \max\{\text{Position}(\langle w \rangle, k) + d, 0\}. \end{aligned} \quad (4.29)$$

Slično se dobiju i odgovarajuće vrijednosti za $\text{State}(\langle w \rangle, k + 1)$ i $\text{Tape}(\langle w \rangle, k + 1)$. \square

Sad kada imamo funkciju State^2 , lako je prepoznati završnu konfiguraciju: to je ona u kojoj je vrijednost te funkcije jednaka $\mathbb{N}Q_0(q_z) = 1$. Štoviše, ako \mathcal{T}_0 -izračunavanje s w stane, tada je najmanji broj za koji to vrijedi upravo broj koraka nakon kojeg izračunavanje stane. (Nemamo probleme kao u primjeru 3.34, jer nam je Turingov stroj fiksni i ne prenosimo ga kao kod, a svaki prirodni broj je kod neke riječi iz Σ_0^* .)

Lema 4.23. *Funkcija stop^1 , definirana sa*

$$\text{stop}(\langle w \rangle) := \text{„broj koraka nakon kojeg } \mathcal{T}_0\text{-izračunavanje s } w \text{ stane}”, \quad (4.30)$$

parcijalno je rekurzivna (naravno, $\mathcal{D}_{\text{stop}} = \mathcal{D}_{\mathbb{N}Q_0} = \{\langle w \rangle \mid w \in \mathcal{D}_{\varphi_0}\}$).

Dokaz. Tvrdimo da je

$$\text{stop}(x) \simeq \mu n (\text{State}(x, n) = 1) \quad (4.31)$$

dobivena minimizacijom primitivno rekurzivne relacije.

Neka je $x \in \mathbb{N}$ proizvoljan, i označimo $w := (\mathbb{N}\Sigma_0^*)^{-1}(x)$. Ako je $w \in \mathcal{D}_{\varphi_0}$, tada po definiciji 4.5 \mathcal{T}_0 -izračunavanje s w stane, te označimo s $n_0 := \text{stop}(x)$ broj koraka nakon kojeg se to dogodi. Prema lemi 4.22 vrijedi $\text{State}(x, n_0) = \mathbb{N}Q_0(q_z) = 1$, ali isto tako za sve $n < n_0$ vrijedi $\text{State}(x, n) \neq 1$, jer stanje još nije završno, a $\mathbb{N}Q_0$ je injekcija. Dakle n_0 je jednak $\mu n(\text{State}(x, n) = 1)$.

Ako pak $w \notin \mathcal{D}_{\varphi_0}$, tada \mathcal{T}_0 -izračunavanje s w ne stane, pa ne postoji $n \in \mathbb{N}$ takav da vrijedi $\text{State}(x, n) = 1$ (opet, jer je $\mathbb{N}Q_0$ injekcija), te izraz $\mu n(\text{State}(x, n) = 1)$ nema smisla, baš kao ni $\text{stop}(x)$. \square

Sada napokon možemo dokazati teorem zbog kojeg smo sve ovo radili.

Teorem 4.24. *Neka je Σ_0 abeceda, $\mathbb{N}\Sigma_0$ njeno kodiranje, i φ_0 Turing-izračunljiva funkcija nad njom. Tada je prateća funkcija $\mathbb{N}\varphi_0$ parcijalno rekurzivna.*

Dokaz. Po pretpostavci, postoji Turingov stroj $\mathcal{T}_0 = (Q_0, \Sigma_0, \Gamma_0, \sqcup, \delta_0, q_0, q_z)$ koji računa φ_0 . Označimo $b' := \text{card } \Sigma_0$ i $b := \text{card } \Gamma_0$. Primjenjujući na taj \mathcal{T}_0 sve što smo napravili u ovoj točki (kodiramo mu stanja, radnu abecedu u skladu s $\mathbb{N}\Sigma_0$, traku, funkciju prijelaza, i izračunavanje s proizvoljnom rječju), dobijemo (među ostalim) funkcije **Recode**, **Tape** i **stop**, sa svojstvima iz odgovarajućih lema. Tvrdimo da je

$$\mathbb{N}\varphi_0(x) \simeq \text{Recode}(\text{Tape}(x, \text{stop}(x)), b, b'), \quad (4.32)$$

pa je $\mathbb{N}\varphi_0$ parcijalno rekurzivna kao kompozicija takvih. Sama parcijalna jednakost (4.32) je zapravo (4.15) iz definicije 4.15, samo izrečena u „izračunljivom obliku”. Za svaki $x \in \mathbb{N}$, imamo dva slučaja ovisno o tome je li $w := (\mathbb{N}\Sigma_0^*)^{-1}(x) \in \mathcal{D}_{\varphi_0}$.

Ako nije, tada $\mathbb{N}\varphi_0(x)$ nema smisla, a niti desna strana od (4.32) nema smisla — \mathcal{T}_0 računa φ_0 , pa $w \notin \mathcal{D}_{\varphi_0}$ znači da \mathcal{T}_0 -izračunavanje s w ne stane. Tada $x \notin \mathcal{D}_{\text{stop}}$ prema lemi 4.23, a onda po marljivoj evaluaciji (2.2) također x nije ni u domeni desne strane.

Ako je pak $w \in \mathcal{D}_{\varphi_0}$, tada \mathcal{T}_0 -izračunavanje s w stane, te prema lemi 4.23, to se dogodi nakon $s := \text{stop}(\langle w \rangle) = \text{stop}(x)$ koraka. Prema lemi 4.22, kod trake završne konfiguracije je onda $t := \text{Tape}(x, s)$, što je $\|\varphi_0(w)\sqcup \dots\|$ jer \mathcal{T}_0 računa φ_0 . Sada je prema (4.20), $\text{Recode}(t, b, b') = \langle \varphi_0(w) \rangle$, kao što i treba biti. \square

4.3. Pretvorba RAM-stroja u Turingov stroj

Pokušajmo sada dokazati obrat teorema 4.24. Kako za parcijalno rekurzivne funkcije imamo kompajler u RAM-strojeve, možemo ga iskoristiti i za kompajliranje u Turingove strojeve. To je također poznata pojava u računarstvu: dolaskom nove arhitekture, često ne moramo iznova kompajlirati izvorni kod. Ako već imamo kod na sličnoj razini (kao što je recimo već kompajlirani RAM-program), možemo ga direktno prevesti („transpilirati”) u kompajlirani kod za drugu arhitekturu. To ćemo učiniti ovdje.

Kroz čitavu ovu točku, imat ćemo fiksiranu abecedu Σ_0 , njeno kodiranje $\mathbb{N}\Sigma_0$, kodiranje $\mathbb{N}\Sigma_0^*$ u pomaknutoj bazi $b' = \text{card } \Sigma_0$, jezičnu funkciju φ_0 nad Σ_0 takvu da je $\mathbb{N}\varphi_0 \in \text{Comp}_1$, te fiksni RAM-algoritam P_0^1 koji računa $\mathbb{N}\varphi_0$.

Cilj će nam biti konstruirati Turingov stroj \mathcal{T}_0 koji računa φ_0 , tako da primi ulaz w na traci, kodira ga u $x := \langle w \rangle$, od toga konstruira početnu konfiguraciju RAM-stroja \mathcal{S}_0 s programom P_0 i ulazom x (efektivno, stavi x u \mathcal{R}_1), te simulira redom korake kroz koje P_0 -izračunavanje s x prolazi. Ako/kad \mathcal{S}_0 uđe u završnu konfiguraciju, \mathcal{T}_0 će sadržaj \mathcal{R}_0 „dekodirati” u riječ $v := \varphi_0(w)$, premjestiti v na početak trake i obrisati sve ostalo s nje. Ako se to nikad ne dogodi, \mathcal{T}_0 će vječno simulirati rad \mathcal{S}_0 , odnosno nikad neće otići u završno stanje — što i treba da bi računao φ_0 , jer činjenica da P_0 -izračunavanje s $\langle w \rangle$ ne stane znači da $w \notin \mathcal{D}_{\varphi_0}$. Efektivno, definiciju prateće funkcije smo „izvrnuli” iznutra van: $\mathbb{N}\varphi_0 = \mathbb{N}\Sigma_0^* \circ \varphi_0 \circ (\mathbb{N}\Sigma_0^*)^{-1}$ znači $\varphi_0 = (\mathbb{N}\Sigma_0^*)^{-1} \circ \mathbb{N}\varphi_0 \circ \mathbb{N}\Sigma_0^*$.

Stroj $\mathcal{T}_0 = (Q, \Sigma_0, \Gamma, \sqcup, \delta, q_0, q_z)$ ćemo konstruirati u pet dijelova (*fragmenata*), koji provode razne faze opisanog postupka simulacije stroja \mathcal{S}_0 . Zasad već imamo fiksiran Σ_0 , te u Q imamo (različita) stanja q_0 i q_z . Ostala stanja uvodit ćemo postepeno, kako budemo specifikirali funkciju δ . Dodat ćemo još jedno stanje, stanje greške q_x , uz standardnu konvenciju da za bilo koji par (q, α) na kojem nismo eksplicitno definirali δ , vrijedi $\delta(q, \alpha) := (q_x, \alpha, 1)$. To je samo birokratski detalj da bi δ bila totalna, jer \mathcal{T}_0 nikada neće otići u stanje q_x : jedini način da računa beskonačno dugo bit će da odgovarajuće P_0 -izračunavanje ne stane.

S druge strane, radnu abecedu već možemo preciznije specifikirati. Prvo, u njoj se svakako nalaze svi znakovi iz Σ_0 , kodirani po $\mathbb{N}\Sigma_0$ u brojeve iz $[1..b']$. Znak $\mathbb{N}\Sigma_0^{-1}(t)$ označit ćemo s α_t . Oni predstavljaju „znamenke” od 1 do b' u pomaknutoj bazi b' .

Za označavanje početka i kraja (upotrijebljenog dijela) trake koristimo graničnik, koji se obično piše $\$$. Također, moramo nekako odvojiti dio za ulaz i dio za računanje (simulaciju): za to ćemo koristiti separator koji se obično piše $\#$. Traka će u početku izgledati otprilike kao $\$ \sqcup^l v \# u \sqcup \dots$, a kasnije kao $\$ \sqcup^k \# uv \$ \sqcup \dots$, gdje je $v \in \Sigma_0^*$ dio za ulaz odnosno izlaz, a $u \in (B^m)^*$ dio za reprezentaciju konfiguracije od \mathcal{S}_0 — o kojem ćemo govoriti nešto kasnije. Sve u svemu, zasad znamo $\Gamma := \Sigma_0 \cup \{\$, \#\} \cup B^m$, gdje je B^m skup koji ćemo kasnije definirati — i sadržavat će prazninu \sqcup .

4.3.1. Inicijalizacija RAM-stroja

Cilj prve faze je dovesti traku u oblik $\$ w \# \sqcup \dots$, tako da možemo početi kodirati. Kako je traka na početku $w \sqcup \dots$, proizlazi da moramo pomaknuti riječ za jednu ćeliju udesno.

Lema 4.25. *Postoji fragment Turingovog stroja koji prevodi početnu konfiguraciju $w = (q_0, 0, w \sqcup \dots)$ u konfiguraciju $\$ w \# = (q_2, |w|, \$ w \# \sqcup \dots)$.*

Dokaz. Standardni način da se to napravi je otići na kraj riječi, pa pomicati slova s kraja redom za po jedno mjesto udesno. To možemo i na Turingovom stroju (pokušajte!), ali

možemo i jednostavnije: obrisat ćemo prvi znak (zamijeniti ga s $\$$), i *zapamtiti ga* u stanju stroja — imat ćemo po jedno stanje n_α za prijenos svakog znaka $\alpha \in \Sigma_0$ (4.33).

Tada, ako u stanju n_α čitamo znak α' , brišemo ga s trake (zamjenjujemo ga s α), i pamtimo u stanju (odlazimo u stanje α') (4.34). To činimo sve dok ne pročitamo prvu \sqcup na traci: ako je to u stanju n_α , pišemo α na traku i prelazimo u novo stanje q_1 (4.35). Ako pak pročitamo \sqcup već u stanju q_0 , to znači da je ulazna riječ bila prazna ($w = \varepsilon$), pa samo treba zamijeniti tu prazninu s $\$$ i odmah otići u stanje q_1 (4.36). U tom stanju samo treba dopisati $\#$ na kraj trake, i pomaknuti se ulijevo (4.37). \square

$$\delta(q_0, \alpha) := (n_\alpha, \$, 1), \text{ za sve } \alpha \in \Sigma_0 \quad (4.33)$$

$$\delta(n_\alpha, \alpha') := (n_{\alpha'}, \alpha, 1), \text{ za sve } \alpha, \alpha' \in \Sigma_0 \quad (4.34)$$

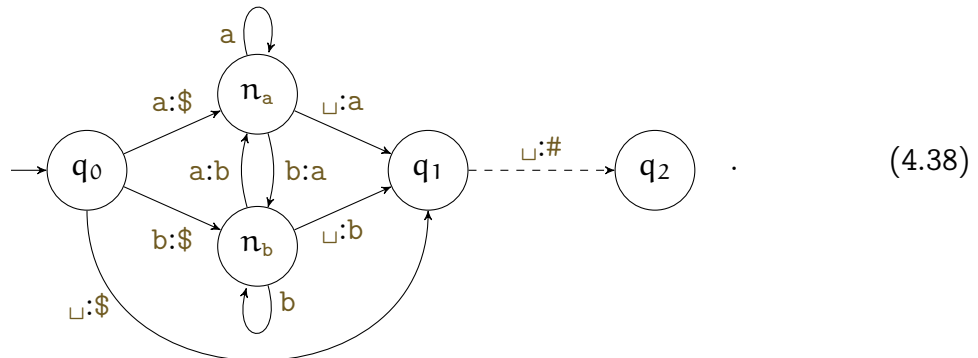
$$\delta(n_\alpha, \sqcup) := (q_1, \alpha, 1), \text{ za sve } \alpha \in \Sigma_0 \quad (4.35)$$

$$\delta(q_0, \sqcup) := (q_1, \$, 1) \quad (4.36)$$

$$\delta(q_1, \sqcup) := (q_2, \#, -1) \quad (4.37)$$

Primijetimo da su jednadžbe (4.33) i (4.36) „specijalni slučajevi” od (4.34) i (4.35) redom, ako stanje q_0 nazovemo $n_\$$. Ipak, neka je vrst običaja početno stanje Turingovog stroja označavati s q_0 .

Primjer 4.26. Za $\Sigma_0 := \{a, b\}$, prvi fragment je prikazan dijagramom



Za riječ aab imali bismo sljedeću šetnju kroz konfiguracije:

$$\begin{array}{ccccccc} aab & \rightsquigarrow & \$ab & \rightsquigarrow & \$ab & \rightsquigarrow & \$aa\sqcup \\ q_0 & & n_a & & n_a & & n_b \end{array} \rightsquigarrow \begin{array}{ccc} \$aab\sqcup & \rightsquigarrow & \$aab\# \\ q_1 & & q_2 \end{array} \quad (4.39)$$

U slučaju općenite abecede Σ , umjesto vertikale koja prolazi kroz n_a i n_b , između q_0 i q_1 bismo imali potpun usmjeren graf s $(\text{card } \Sigma)^2$ bridova, nad skupom vrhova $\{n_\alpha \mid \alpha \in \Sigma\}$. \triangleleft

Sljedeći korak je konstruirati početnu konfiguraciju od S_0 , desno od separatora $\#$ na poziciji $|w|$. Znamo da to mora biti element od $(B^m)^*$, pa pokušajmo s unarnim kodiranjem: uzmimo jedan konkretan element $r_1 \in B^m$, i cilj nam je desno od $\#$ napisati $r_1^{(w)}$. Što je točno r_1 (i čitav B^m), reći ćemo u idućoj točki.

Lema 4.27. *Postoji fragment Turingovog stroja koji prevodi konfiguraciju*

$\$w\# = (q_2, |w|, \$w\# \sqcup \dots)$ u konfiguraciju $\$ \sqcup^{|w|} \# r_1^{\langle w \rangle} = (p_0, |w| + 2, \$ \sqcup^{|w|} \# r_1^{\langle w \rangle} \sqcup \dots)$.

Dokaz. To je pretvorba zapisa između različitih pomaknutih baza: broj $\langle w \rangle$ čitamo u pomaknutoj bazi b' pomoću znamenaka $\alpha_t, t \in [1..b']$, i pišemo u pomaknutoj bazi 1 pomoću znamenke r_1 . To se može riješiti tako da „dekrementiramo” w , i za svaki uspješni dekrement dodamo po jedan r_1 na kraj.

Kako dekrementirati broj zapisan u pomaknutoj bazi? Vrlo slično kao i zapis u običnoj bazi. Ako zadnji znak nije α_1 , smanjimo njegov kod za 1 (4.40), i odemo desno (4.41) do prvog razmaka, koji zamijenimo s r_1 (4.42) i vraćamo se ponovo lijevo (4.43). Ako naiđemo na α_1 , zamijenimo ga „najvećom znamenkom” $\alpha_{b'}$, i nastavljamo smanjivati dalje lijevo (4.44). Ako smanjujući dođemo do lijevog ruba broja (4.45), prvu znamenku mu obrišemo (4.46). Ako smo ostali bez znamenaka, gotovi smo: prijedemo desno iza separatora, u stanje p_0 (4.47).

$$\delta(q_2, \alpha_t) := (q_3, \alpha_{t-1}, 1), \text{ za sve } t \in [2..b'] \quad (4.40)$$

$$\delta(q_3, \alpha) := (q_3, \alpha, 1), \text{ za sve } \alpha \in \Sigma_0 \dot{\cup} \{\#, r_1\} \quad (4.41)$$

$$\delta(q_3, \sqcup) := (q_2, r_1, -1) \quad (4.42)$$

$$\delta(q_2, \gamma) := (q_2, \gamma, -1), \text{ za } \gamma \in \{\#, r_1\} \quad (4.43)$$

$$\delta(q_2, \alpha_1) := (q_2, \alpha_z, -1) \quad (4.44)$$

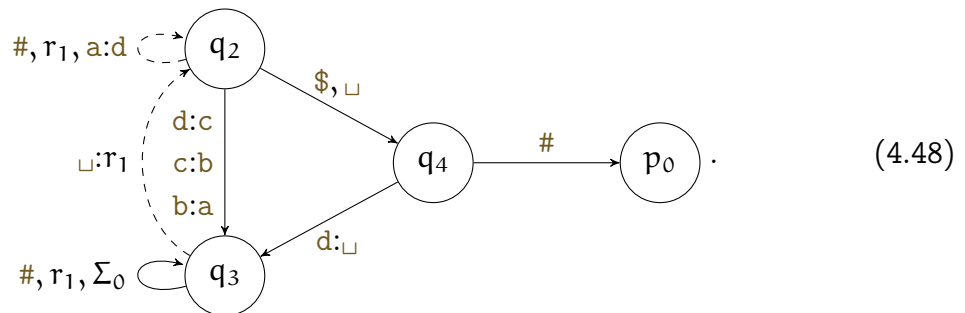
$$\delta(q_2, \gamma) := (q_4, \gamma, 1), \text{ za } \gamma \in \{\$, \sqcup\} \quad (4.45)$$

$$\delta(q_4, \alpha_{b'}) := (q_3, \sqcup, 1) \quad (4.46)$$

$$\delta(q_4, \#) := (p_0, \#, 1) \quad (4.47)$$

Za dokaz da to stvarno dovede do željene konfiguracije, dovoljno je primijetiti da se nakon svakog lijevo-desno prolaza nađemo ponovo u stanju q_2 , s trakom $\$ \sqcup^{|w|-|w'|} w' \# r_1^{\langle w \rangle - \langle w' \rangle}$ i pozicijom pri desnom kraju. Na početku je to istina jer je $w = w'$ i još nemamo nijedan r_1 , u svakom koraku ostaje istina jer se dekrementom $\langle w' \rangle$ smanjuje za 1, dok se broj znakova r_1 povećava za 1, te na kraju imamo $\$ \sqcup^{|w|} \# r_1^{\langle w \rangle}$, jer je tada $w' = \varepsilon$ s kodom $\langle \varepsilon \rangle = 0$. \square

Primjer 4.28. Za $\Sigma_0 := \{a, b, c, d\}$, drugi fragment je prikazan dijagramom



Za riječ **aab** imamo šetnju kroz konfiguracije (pišemo samo neke od njih):

$$\begin{aligned}
 & \$_{q_2}aab\# \rightsquigarrow \$_{q_3}aaa\# \rightsquigarrow \$_{q_3}aaa\#_{\perp} \rightsquigarrow \$_{q_2}aaa\#r_1 \rightsquigarrow^4 \$_{q_2}ddd\#r_1 \rightsquigarrow \$_{q_4}ddd\#r_1 \rightsquigarrow \$_{q_3}\#ddr_1 \rightsquigarrow^4 \\
 & \rightsquigarrow^4 \$_{q_3}\#ddr_1\# \rightsquigarrow \$_{q_2}\#ddr_1r_1 \rightsquigarrow \$_{q_2}\#ddr_1r_1 \rightsquigarrow \$_{q_2}\#ddr_1^2 \rightsquigarrow \$_{q_3}\#dc\#r_1^2 \rightsquigarrow^3 \$_{q_3}\#dc\#r_1^2\# \rightsquigarrow^4 \\
 & \rightsquigarrow^4 \$_{q_2}\#dc\#r_1^3 \rightsquigarrow^* \$_{q_2}\#db\#r_1^4 \rightsquigarrow^* \$_{q_2}\#da\#r_1^5 \rightsquigarrow^* \$_{q_2}\#cd\#r_1^6 \rightsquigarrow^* \$_{q_2}\#ca\#r_1^9 \rightsquigarrow^* \$_{q_2}\#ba\#r_1^{13} \rightsquigarrow^* \\
 & \rightsquigarrow^* \$_{q_2}\#aa\#r_1^{17} \rightsquigarrow^* \$_{q_2}\#dd\#r_1^{18} \rightsquigarrow^* \$_{q_2}\#aa\#r_1^{21} \rightsquigarrow \$_{q_2}\#dd\#r_1^{21} \rightsquigarrow \$_{q_4}\#dd\#r_1^{21} \rightsquigarrow \$_{q_3}\#dd\#r_1^{21} \rightsquigarrow^* \\
 & \rightsquigarrow^* \$_{q_3}\#dd\#r_1^{21}\# \rightsquigarrow^* \$_{q_2}\#dd\#r_1^{22} \rightsquigarrow \$_{q_4}\#dd\#r_1^{22} \rightsquigarrow \$_{p_0}\#dd\#r_1^{22}, \quad (4.49)
 \end{aligned}$$

što je točno kako treba biti jer je $b' = 4$, pa je $\langle aab \rangle = (112)_4 = 22$. Primijetimo da je „invarijanta petlje” zadovoljena u svim istaknutim konfiguracijama gdje se q_2 nalazi ispod zadnjeg znaka prije separatora, npr. za $\$_{q_2}\#cd\#r_1^6$ je $w' = cd$, čiji je kod $\langle cd \rangle = (34)_4 = 16$, te je $\langle w \rangle - \langle w' \rangle = 22 - 16 = 6$. \triangleleft

4.3.2. Registri kao tragovi trake

Vrijeme je da opišemo kako ćemo točno reprezentirati konfiguraciju RAM-stroja S_0 na traci. Zapravo, vrijednost programskog brojača je jedna od konačno mnogo njih, i na njoj su dopuštene proizvoljne transformacije, pa ju je bolje prikazati kroz stanje stroja. Na traci će stajati samo stanje registara — i to samo onih relevantnih. U tu svrhu, označimo s $m := m_{P_0^1} = \max\{m_{P_0}, 2\}$ širinu algoritma P_0^1 ; tada znamo da je dovoljno pratiti registre \mathcal{R}_0 do \mathcal{R}_{m-1} .

Kako to najbolje učiniti? Kao što smo rekli na početku ovog poglavlja, jezični model je „transponirani” brojevni model: u jezičnom modelu broj ćelija nije ograničen, ali broj mogućih sadržaja svake ćelije jest, dok je u brojevnom modelu obrnuto. To znači da možemo stanje relevantnih (prvih m) registara prikazati kao riječ nad „abecedom m -bitnih procesorskih riječi”

$$B^m := \prod_{i < m} \{\circ, \bullet\} = \left\{ \begin{bmatrix} \beta_0 \\ \vdots \\ \beta_{m-1} \end{bmatrix} \mid (\forall i < m)(\beta_i \in \{\circ, \bullet\}) \right\}, \quad (4.50)$$

tako da se dio trake desno od separatora sastoji od m „redaka” (*tragova*), u svakom od kojih piše $\bullet^t \circ \dots$, gdje je t sadržaj odgovarajućeg registra. U tom kontekstu, B^m je „prostor stupaca” sa svim (2^m njih) kombinacijama \bullet i \circ visine m . Recimo,

$$B^3 := \left\{ \begin{array}{c} \circ \\ \circ \end{array}, \begin{array}{c} \bullet \\ \circ \end{array}, \begin{array}{c} \circ \\ \bullet \end{array}, \begin{array}{c} \bullet \\ \bullet \end{array}, \begin{array}{c} \circ \\ \circ \end{array}, \begin{array}{c} \bullet \\ \circ \end{array}, \begin{array}{c} \circ \\ \bullet \end{array}, \begin{array}{c} \bullet \\ \bullet \end{array} \right\}. \quad (4.51)$$

Konkretno, ako (za $m = 4$) u nekom trenutku imamo konfiguraciju $(1, 4, 0, 5, 0, 0, \dots)$, te je $|w| = 7$, traka će biti

$$\$\begin{array}{cccccccc} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \end{array} \# \dots, \quad (4.52)$$

iz čega vidimo dvije stvari. Prvo, \sqcup je očito ovaj prvi element u B^m , koji ima sve kružice prazne. Tako direktno postizemo da traka bude prazna od nekog mjesta nadalje (konkretno, maksimum svih m vrijednosti u registrima), odnosno da bude s konačnim nosačem. Drugo, naš znak r_1 , koji smo upotrijebili da postavimo \mathcal{R}_1 na ulaznu vrijednost $\langle w \rangle$, je treći element u B^m , napisan transponirano kao $(\circ \bullet \circ^{m-2})^\top$. Stupčano,

$$\sqcup := \begin{array}{c} \circ \\ \circ \\ \vdots \\ \circ \end{array}, \quad r_1 := \begin{array}{c} \circ \\ \bullet \\ \vdots \\ \circ \end{array}. \quad (4.53)$$

Primjer 4.29. Nad abecedom $\Sigma := \{a, b\}$ (kodiranom s $a \mapsto 1$, $b \mapsto 2$) pogledajmo jezičnu funkciju zadanu s $\varphi_a(w) := wa$ (dopisivanje a zdesna). Baza je $b' = 2$, te je $N\varphi_a(\langle w \rangle) = \langle wa \rangle = \langle w \rangle \frown \langle a \rangle = \langle w \rangle \frown 1 = \langle w \rangle \cdot 2^{\text{slh}(1,2)} + 1 = 2\langle w \rangle + 1$, odnosno $N\varphi_a(x) = 2x + 1$. Očito je $N\varphi_a \in \text{Comp}_1$: računa je jednostavan RAM-program

$$P_a := \begin{bmatrix} 0. \text{ DEC } \mathcal{R}_1, 4 \\ 1. \text{ INC } \mathcal{R}_0 \\ 2. \text{ INC } \mathcal{R}_0 \\ 3. \text{ GO TO } 0 \\ 4. \text{ INC } \mathcal{R}_0 \end{bmatrix} \quad (4.54)$$

čija je širina $m = 2$. Za ulaznu riječ $w_2 := ab$ imamo $\langle w_2 \rangle = (12)_2 = 4$, pa je konfiguracija Turingovog stroja kroz faze (početna, nakon prve faze i nakon druge faze)

$$\begin{array}{c} ab \\ q_0 \end{array} \rightsquigarrow^* \begin{array}{c} \$ab\# \\ q_2 \end{array} \rightsquigarrow^* \begin{array}{c} \$\circ\circ\circ\#\bullet\bullet\bullet\bullet\bullet \\ p_0 \end{array}. \triangleleft \quad (4.55)$$

Rekli smo da ćemo vrijednost programskog brojača držati u stanju stroja, i to je već učinjeno kroz supskript stanja p_0 : ta nula znači da je PC upravo postao 0. Općenito, ako je P_0 duljine $n := n_{p_0}$, imat ćemo stanja $p_i, i \in [0..n]$, i ulazak u stanje p_n značit će završetak P_0 -izračunavanja.

Ipak, stanje p_i neće biti dovoljno za izvršavanje instrukcije I_i : za svaki $i \leq n$ imat ćemo još jedno stanje s_i , koje će zapravo izvesti osnovnu logiku odgovarajuće instrukcije (s_n će biti zaduženo za nastavak rada \mathcal{T}_0 nakon simulacije \mathcal{S}_0). Stanje p_i je „pripremno stanje” za s_i , i jedini njegov zadatak je fiksirati poziciju odmah nakon znaka $\#$.

Lema 4.30. *Za svaki $i \in [0..n]$, postoji fragment Turingovog stroja koji prevodi bilo koju konfiguraciju oblika $(p_i, t, \$ \sqcup^k \# v \sqcup \dots)$, gdje je $v \in (B^m)^*$ i $t > k \in \mathbb{N}$, u konfiguraciju $(s_i, k+2, \$ \sqcup^k \# v \sqcup \dots)$.*

Dokaz. Vrlo je jednostavno. Dok god čitamo znak iz B^m , pomičemo se lijevo (4.56). Kad pročitamo $\#$ (nakon $t - k - 1$ koraka), pomaknemo se desno (4.57).

$$\delta(p_i, \gamma) := (p_i, \gamma, -1), \text{ za sve } \gamma \in B^m \quad (4.56)$$

$$\delta(p_i, \#) := (s_i, \#, 1) \quad (4.57)$$

Dijagramatski, $B^m \xrightarrow{\quad} p_i \xrightarrow{\#} s_i$. □

Da bismo implementirali prijelaze iz stanja s_i , moramo naučiti „adresirati bitove” na traci oblika (4.52). Svaki element od B^m prirodno vidimo kao m -bitnu riječ, i za promjenu pojedinog bita koristimo istu tehniku kao „pravi” procesor: kompletnu procesorsku riječ zamijenimo drugom, koja se podudara s njom u svim bitovima osim onog koji želimo promijeniti.

Definicija 4.31. Neka je $j \in [0..m]$, neka su $q, q' \in Q$, neka su $\beta, \beta' \in B := \{\circ, \bullet\}$, te neka je $d \in \{-1, 1\}$. Pišemo $\delta_{(j)}(q, \beta) := (q', \beta', d)$ kao pokratu za $\delta(q, \gamma) := (q', \gamma', d)$, za sve $\gamma, \gamma' \in B^m$ takve da je $\gamma_j = \beta$, $\gamma'_j = \beta'$, te $\gamma_i = \gamma'_i$ za sve $i \in [0..m] \setminus \{j\}$.

U dijagramima, na strelici od q prema q' pišemo $\beta : \beta' @ j$. ◁

Definicija 4.32. Za $k, m, n \in \mathbb{N}$ takve da je $m \geq 2$, za proizvoljnu RAM-konfiguraciju c sa svojstvom $c(\mathcal{R}_j) = 0$ za sve $j \geq m$, i $c(PC) \leq n$, *reprezentacija* je Turing-konfiguracija

$$\text{Turing}_{kmn}(c) := \left(s_{c(PC)}, k + 2, \$ \sqcup^k \# \begin{bmatrix} \bullet c(\mathcal{R}_0) \circ \dots \\ \bullet c(\mathcal{R}_1) \circ \dots \\ \vdots \\ \bullet c(\mathcal{R}_{m-1}) \circ \dots \end{bmatrix} \right). \quad (4.58)$$

◁

Recimo, u (4.52) je navedena traka konfiguracije $\text{Turing}_{74n}((1, 4, 0, 5, 0, 0, \dots))$. Također, izračunavanje (4.55) se po lemi 4.30 nastavlja u konfiguraciju $\text{Turing}_{225}(c_0)$, gdje je c_0 početna konfiguracija stroja \mathcal{S}_0 s programom P_0 iz (4.54), s ulazom *ab*.

Propozicija 4.33. Neka je $P = \left[t. I_t \right]_{t < n}$ RAM-program, i označimo $m := m_{P1}$. Tada za svaki $i < n$ postoji fragment Turingovog stroja, koji za svaku RAM-konfiguraciju c takvu da je $c(PC) = i$, za svaki $k \in \mathbb{N}$, prevodi reprezentaciju $\text{Turing}_{kmn}(c)$ u reprezentaciju $\text{Turing}_{kmn}(d)$, gdje smo s d označili jedinstvenu RAM-konfiguraciju takvu da $c \rightsquigarrow d$ po programu P .

Dokaz. Prvo primijetimo da zbog $c(PC) = i < n = n_P$ konfiguracija c sigurno nije završna, pa postoji instrukcija $I_i = I_{c(PC)}$. Konstruiramo traženi fragment ovisno o tipu te instrukcije. Taj fragment počinje u stanju s_i , a trebao bi završiti u stanju $s_{i'}$, gdje smo označili $i' := d(PC)$ — no zbog leme 4.30 dovoljno je stati u $p_{i'}$, na bilo kojoj poziciji $p > k$, i s trakom kakva je u $\text{Turing}_{kmn}(d)$.

Ako je I_i tipa INC, recimo i. INC \mathcal{R}_j , trebamo otići do kraja znakova \bullet u tragu j (4.59), i dopisati još jedan \bullet tamo, te se pripremiti za sljedeću instrukciju, onu rednog broja $i + 1$ (4.60).

$$\delta_{(j)}(s_i, \bullet) := (s_i, \bullet, 1) \quad (4.59)$$

$$\delta_{(j)}(s_i, \circ) := (p_{i+1}, \bullet, -1) \quad (4.60)$$

Ako je I_i tipa DEC, recimo i. DEC \mathcal{R}_j, l , trebamo kao i za INC doći do kraja znaka \bullet (4.61), te kada pročitamo \circ , pomaknuti se lijevo u novo „stanje odluke” t_i (4.62). Ako u tom stanju čitamo $\#$, znači da je $c(\mathcal{R}_j) = 0$, te prelazimo na instrukciju rednog broja l (4.63). Ako ne, tada u tragu j zamijenimo \bullet s \circ , i prelazimo na sljedeću instrukciju (4.64).

$$\delta_{(j)}(s_i, \bullet) := (s_i, \bullet, 1) \quad (4.61)$$

$$\delta_{(j)}(s_i, \circ) := (t_i, \circ, -1) \quad (4.62)$$

$$\delta(t_i, \#) := (p_l, \#, 1) \quad (4.63)$$

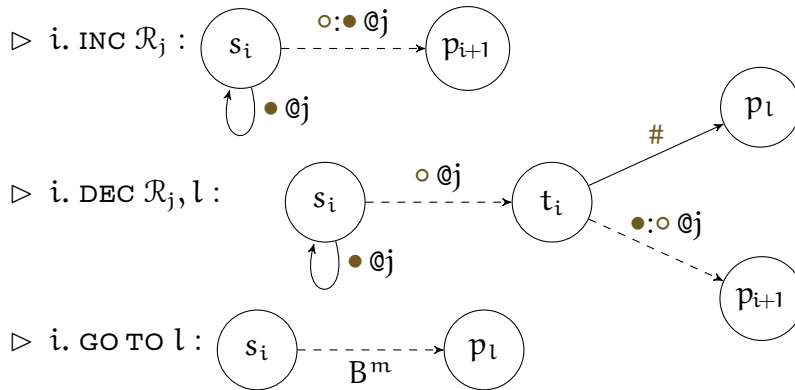
$$\delta_{(j)}(t_i, \bullet) := (p_{i+1}, \circ, -1) \quad (4.64)$$

Ako je I_i tipa GO TO, recimo i. GO TO l , samo odemo na pripremu za instrukciju I_l .

$$\delta(s_i, \gamma) := (p_l, \gamma, -1), \text{ za sve } \gamma \in B^m \quad (4.65)$$

Sada samo još treba na svako stanje oblika p_i , „našarafiti” konstrukciju iz leme 4.30. Lako je vidjeti da je pozicija u svakom od tih stanja strogo veća od k , ako je na početku u stanju s_i bila $k + 2$; recimo primjenom (4.63) će pozicija biti upravo $k + 2 > k$, a primjenom (4.64) će biti $k + c(\mathcal{R}_j)$, što je veće od k jer je u tom slučaju $c(\mathcal{R}_j) > 0$. \square

Sva tri tipa fragmenata iz dokaza možemo prikazati sljedećim dijagramima:



4.3.3. Prepoznavanje završne konfiguracije

Korolar 4.34. Neka je P RAM-program, i označimo $n := n_P$, $m := m_{P1}$. Označimo $s_f := \{\ulcorner P \urcorner\}^1$ jednomjesnu funkciju koju P računa. Tada postoji fragment Turingovog stroja takav da za sve $k \in \mathbb{N}$:

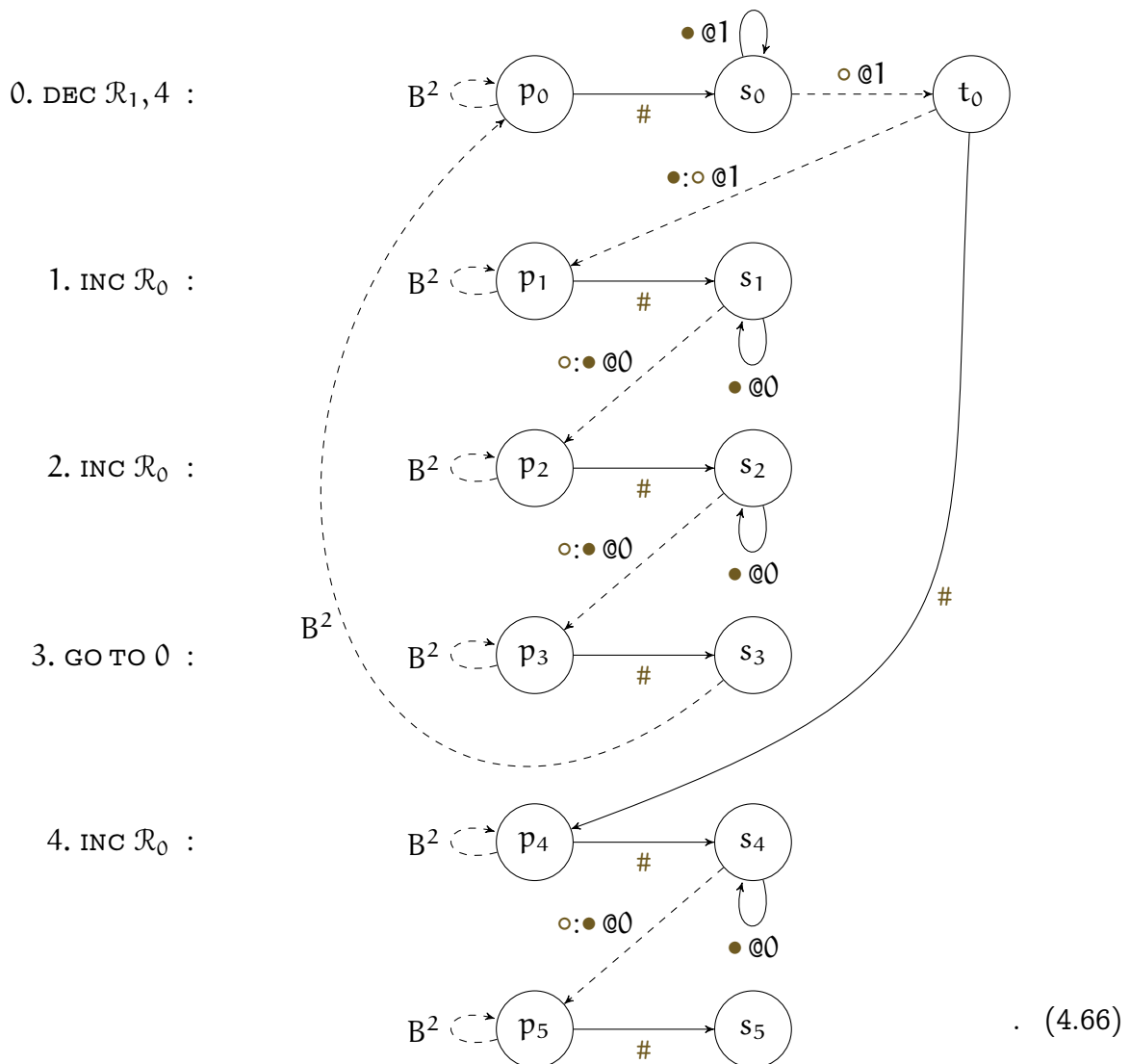
- za sve $x \in \mathcal{D}_f$, prevodi reprezentaciju (Turing_{kmn}) početne konfiguracije s ulazom x u reprezentaciju odgovarajuće završne konfiguracije;
- za sve $x \in \mathcal{D}_f^c$, počevši od reprezentacije početne konfiguracije s ulazom x , nikada ne stigne u stanje s_n .

Dokaz. Za svaki $x \in \mathbb{N}$, označimo sa $(c_l)_l$ P-izračunavanje s x , i indukcijom po l dokažimo da tako sastavljeni fragment Turingovog stroja dostigne konfiguraciju $\text{Turing}_{kmn}(c_l)$. Baza je primjena leme 4.30 (za $i = 0$) na konfiguraciju koju smo dobili iz leme 4.27. Korak je primjena propozicije 4.33. Dakle, ako P-izračunavanje stane u l_0 koraka, doći će u stanje s_n , u reprezentaciji završne konfiguracije c_{l_0} .

No vrijedi i obrat: u simulaciji jednog RAM-prijelaza $c \rightsquigarrow d$, naš fragment posjećuje samo stanja p_i , s_i i t_i za $i = c(\text{PC})$ ili za $i = d(\text{PC})$. Dakle, jedini način da dođe u stanje s_n je da doista neka RAM-konfiguracija u izračunavanju preslika PC u n , a to znači da P-izračunavanje s x stane. \square

Primjer 4.35. U primjeru 4.29 smo vidjeli RAM-program P_a (4.54) duljine 5 i širine 2, koji računa funkciju $\mathbb{N}\varphi_a(x) := 2x + 1$, prateću funkciju dopisivanja a na kraj riječi.

Fragment Turingovog stroja koji odgovara (treći fragment) je prikazan dijagramom



P_a -izračunavanje s $\langle \mathbf{ab} \rangle = 4$ je prikazano tablicom

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	...
$\mathbf{c}_i(\mathcal{R}_0)$	0	0	1	2	2	2	3	4	4	4	5	6	6	6	7	8	8	8	9	9
$\mathbf{c}_i(\mathcal{R}_1)$	4	3	3	3	3	2	2	2	2	1	1	1	1	0	0	0	0	0	0	0
$\mathbf{c}_i(\mathbf{PC})$	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	4	5	5

s izlaznim podatkom $9 = 2 \cdot 4 + 1$, i odgovara mu (po preslikavanju Turing_{225}) šetnja kroz Turing-konfiguracije (ne pišemo, svima zajednički, početak $\$ \circ \circ \circ \#$)

[illegible]

koja se prirodno nastavlja na onu započetu u (4.55) (kroz lemu 4.30 za $i = 0$). \triangleleft

Korolar 4.36. *Dosad izgrađeni Turingov stroj dostigne stanje s_n ako i samo ako je pokrenut s ulazom $w \in \mathcal{D}_{\varphi_0}$, i u tom trenutku na traci nakon $\#$ u tragu 0 piše $\bullet y_0 \dots$, gdje je $y = \mathbb{N}\varphi_0(\langle w \rangle) = \langle \varphi_0(w) \rangle$.*

Dokaz. Samo treba primijeniti upravo dokazani korolar na RAM-program P_0 (dakle $f = \mathbb{N}\varphi_0$), na $x := \langle w \rangle$ i $k := |w|$. \square

Napomena 4.37. Ostatak Turingovog stroja koji konstruiramo vezat će se s dosad izgrađenim dijelom isključivo kroz stanje s_n , i njegovo računanje će uvijek stati, pa ćemo na kraju moći zaključiti da čitav Turingov stroj stane ako i samo ako dođe u stanje s_n , odnosno ako i samo ako je ulazna riječ u domeni od φ_0 . \triangleleft

Time smo riješili pitanje domene, i potrebno je još samo natrag dekodirati y (ako postoji) u riječ $v := \varphi_0(w)$, i postaviti je na početak trake.

4.3.4. Dekodiranje izlazne vrijednosti

Ovaj će postupak biti sličan onome u drugoj fazi, samo u obrnutom smjeru. Imamo broj y zapisan unarno (pomaknuta baza 1) u tragu 0, i trebamo ga zapisati u pomaknutoj bazi b' — skidajući po jedan \bullet iz traga 0 i inkrementirajući riječ iz Σ_0^* svaki put. Ipak, nekoliko tehničkih detalja čine ovaj postupak kompliciranijim.

Prvo, umjesto pisanja konstantnog znaka $r_1 \in B^m$, morat ćemo čitati bilo koji znak koji u tragu 0 ima \bullet . Srećom, tehniku za to smo napravili (definicija 4.31).

Drugo i važnije, kod dekrementiranja smo znali da riječ neće postati dulja, te smo prazninama skraćivali broj koliko je već potrebno, dok ga nismo dekrementirali sasvim

do nule. Kod inkrementiranja nemamo tu garanciju, i općenito se može dogoditi da puno puta moramo dodati po jednu znamenku slijeva (recimo, za $b' = 5$, kad inkrementiramo 555 u 1111).

Zato je dobro riječ graditi unatrag, samo moramo biti sigurni da smo zauzeli dovoljno trake za to (da ne udarimo u lijevi rub, odnosno graničnik \$, prilikom dodavanja nove znamenke slijeva). Posljedica toga je da ne možemo učiniti intuitivno očitu stvar i upotrijebiti samo dio trake lijevo od separatora # — jer je taj dio ograničen duljinom ulazne riječi, a izlazna riječ može biti puno dulja od ulazne.

Spasit ćemo se tako što ćemo za rekonstrukciju izlazne riječi upotrijebiti dio *desno* od separatora, unutar kojeg smo simulirali P_0 -izračunavanje s w . Taj dio će sigurno biti dovoljno dugačak, jer sadrži barem $y = \langle \varphi_0(w) \rangle$ znakova \bullet u tragu 0.

Lema 4.38. *Za svaku riječ u vrijedi $|u| \leq \langle u \rangle$.*

Dokaz. Označimo s b' broj znakova abecede Σ nad kojom je riječ w . Svaki pribrojnik u sumi (4.10) je pozitivan, jer je svaka znamenka $\mathbb{N}\Sigma(\alpha_i) \in [1..b']$, te je $b' > 0$ zbog $\Sigma \neq \emptyset$. Dakle, svaki pribrojnik je barem 1, pa je suma veća ili jednaka broju pribrojnika. No suma je po definiciji $\langle w \rangle$, a broj pribrojnika je upravo $n = |w|$.

Za $b' = 1$ vrijedi jednakost (jer je jedina moguća znamenka 1). \square

Štoviše, lema 4.38 pokazuje da riječ možemo dekodirati korak po korak: u svakom trenutku će nam u tragu 0 ostati y' znakova \bullet , a od desnog kraja će biti napisana riječ u' čiji je kod $\langle u' \rangle = y - y'$, pa će sigurno stati u prostor od $y - y'$ ćelija.

Lema 4.39. *Postoji fragment Turingovog stroja koji prevodi reprezentaciju završne konfiguracije $(s_n, |w| + 2, \$ \sqcup^{|w|} \# t \sqcup \dots)$, gdje t u tragu 0 ima oblik $\bullet^y \circ^{z-y}$, u konfiguraciju $(q_5, |w| + 1, \$ \sqcup^{|w|} \# uv \$ \sqcup \dots)$, gdje je $\langle v \rangle = y$, te $u \in (B^m)^{z-|v|}$.*

$$\delta(s_n, \gamma) := (s_n, \gamma, 1), \text{ za sve } \gamma \in B_+^m := B^m \setminus \{\sqcup\} \quad (4.68)$$

$$\delta(s_n, \sqcup) := (q_5, \$, -1) \quad (4.69)$$

$$\delta_{(0)}(q_5, \circ) := (q_5, \circ, -1) \quad (4.70)$$

$$\delta_{(0)}(q_5, \bullet) := (q_6, \circ, 1) \quad (4.71)$$

$$\delta(q_6, \alpha) := (q_6, \alpha, 1), \text{ za sve } \alpha \in \Sigma_0 \dot{\cup} B^m \quad (4.72)$$

$$\delta(q_6, \$) := (q_7, \$, -1) \quad (4.73)$$

$$\delta(q_7, \alpha_{b'}) := (q_7, \alpha_1, -1) \quad (4.74)$$

$$\delta(q_7, \alpha_t) := (q_5, \alpha_{t+1}, -1), \text{ za sve } t \in [1..b'] \quad (4.75)$$

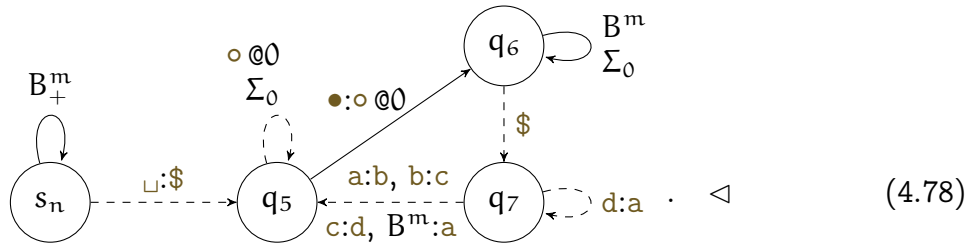
$$\delta(q_7, \gamma) := (q_5, \alpha_1, -1), \text{ za sve } \gamma \in B^m \quad (4.76)$$

$$\delta(q_5, \alpha) := (q_5, \alpha, -1), \text{ za sve } \alpha \in \Sigma_0 \quad (4.77)$$

Dokaz. Kao što rekosmo, dokaz je sličan dokazu leme 4.27, samo u suprotnom smjeru. Prvo odemo do kraja upotrijebljenog (nepraznog) dijela trake (4.68), i tamo zapišemo

krajnji graničnik (4.69). Zatim se krećemo ulijevo kroz znakove \circ u tragu 0 (4.70) dok ne dođemo do \bullet , koji maknemo (4.71) i prenosimo ga desno kroz „znamenke” i „stupce bitova” (4.72), dok ne dođemo do krajnjeg graničnika (4.73). Tada se opet krećemo ulijevo, inkrementirajući riječ: zamjenjujući „devetke jedinica” (4.74), te inkrementirajući prvu „ne-devetku” (4.75), ili ako takve nema, dodajući „jedinicu” slijeva (4.76). Zatim prođemo kroz ostale znamenke do „praznog prostora” (4.77), i opet izvršavamo cijeli postupak dok ne dođemo do separatora $\#$. Argument zašto to funkcionira je praktički isti kao u dokazu leme 4.27. \square

Primjer 4.40. Za $\Sigma_0 := \{a, b, c, d\}$, četvrti fragment možemo prikazati dijagramom



Primijetimo da zbog $\langle v \rangle = y = \langle \varphi_0(w) \rangle$, i propozicije 4.14, zapravo vrijedi $v = \varphi_0(w)$, odnosno (ako uopće dođemo u situaciju da možemo primijeniti lemu 4.39) već imamo izlaznu riječ na traci. Još je treba premjestiti na početak, i obrisati s trake sve ostalo.

Lema 4.41. *Postoji fragment Turingovog stroja koji prevodi konfiguraciju $(q_5, |w| + 1, \$ \sqcup^{|w|} \# uv \$ \sqcup \dots)$, gdje je $v \in \Sigma_0^*$, te $u \in (B^m)^*$, u konfiguraciju $(q_z, |v|, v \sqcup \dots)$.*

$$\delta(q_5, \#) := \delta(q_8, \gamma) := (q_8, \sqcup, 1), \text{ za sve } \gamma \in B^m \quad (4.79)$$

$$\delta(q_8, \alpha) := \delta(m_\alpha, \sqcup) := (m_\alpha, \sqcup, -1), \text{ za sve } \alpha \in \Sigma_0 \quad (4.80)$$

$$\delta(m_\alpha, \alpha') := (l_\alpha, \alpha', 1), \text{ za sve } \alpha, \alpha' \in \Sigma_0 \quad (4.81)$$

$$\delta(m_\alpha, \$) := \delta(l_\alpha, \sqcup) := (q_8, \alpha, 1), \text{ za sve } \alpha \in \Sigma_0 \quad (4.82)$$

$$\delta(q_8, \$) := \delta(q_9, \sqcup) := (q_9, \sqcup, -1) \quad (4.83)$$

$$\delta(q_9, \alpha) := (q_z, \alpha, 1), \text{ za sve } \alpha \in \Sigma_0 \quad (4.84)$$

$$\delta(q_9, \$) := (q_z, \sqcup, -1) \quad (4.85)$$

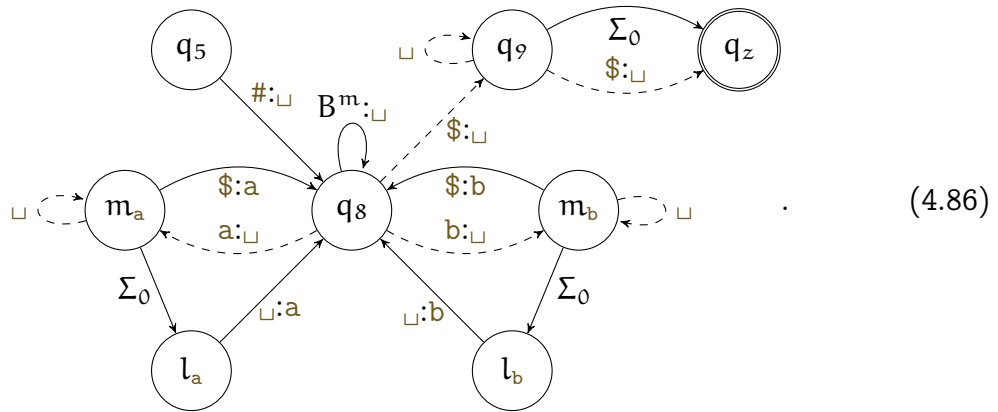
Dokaz. Prvi korak je brisanje svih ostataka izračunavanja, i separatora, tako da na traci ostanu jedino izlazna riječ v i dva graničnika — početni i krajnji (4.79). Time dobivamo konfiguraciju $(q_8, l + 1, \$ \sqcup^l v \$ \sqcup \dots)$, gdje je $l := |w| + |u| + 1$.

Drugi korak je uzimanje jednog po jednog znaka od v , i njegov prijenos na početak trake (4.80) — s tim da će prvi znak od v naići na početni graničnik i prebrisati ga (4.82), a svaki sljedeći znak će naići na onaj prije njega koji je već prenesen (4.81), i zapisati se desno od njega (4.82). Time dobijemo konfiguraciju $(q_8, l + |v| + 1, v \sqcup^{l+1} \$ \sqcup \dots)$ ako je $v \neq \varepsilon$. Za $v = \varepsilon$, ovog koraka nema, odnosno odmah smo u konfiguraciji $(q_8, l + 1, \$ \sqcup^l \$ \sqcup \dots)$.

Treći korak počinje čitanjem i brisanjem završnog graničnika \$, i nastavlja prolaskom kroz praznine ulijevo (4.83), sve dok ne udarimo u zadnji znak od v ako je $v \neq \varepsilon$ (4.84), odnosno u početni graničnik (koji tada treba obrisati, ostavivši traku sasvim praznom) u slučaju $v = \varepsilon$ (4.85).

U slučaju neprazne riječi v , očito će završna pozicija biti neposredno iza zadnjeg znaka od v , dakle $|v|$. U slučaju prazne riječi, prijelaz (4.85) će se dogoditi na lijevom rubu trake, pokušati pomak lijevo, i ostati na istoj poziciji $0 = |\varepsilon| = |v|$. U svakom slučaju dobili smo stanje q_z , traku $v \sqcup \dots$, i poziciju $|v|$. \square

Primjer 4.42. Za $\Sigma_0 := \{a, b\}$, peti fragment možemo prikazati dijagramom



Za općenitu abecedu Σ_0 , umjesto dva „viseća trokuta” ispod q_8 , imat ćemo ih b' : po jedan $\triangle q_8 m_\alpha l_\alpha$ za svaki znak $\alpha \in \Sigma_0$. \triangleleft

4.3.5. Objedinjavanje

Teorem 4.43. Neka je Σ_0 abeceda, $\mathbb{N}\Sigma_0$ njeno kodiranje, i φ_0 funkcija nad njom. Ako je $\mathbb{N}\varphi_0 \in \text{Comp}_1$, tada je φ_0 Turing-izračunljiva.

Dokaz. Po pretpostavci postoji RAM-program P_0 koji računa $\mathbb{N}\varphi_0$. Označimo $n := n_{P_0}$ i $m := m_{P_0}$. Tvrdimo da tada funkciju φ_0 računa Turingov stroj

$$\mathcal{T}_0 := (Q, \Sigma_0, \Gamma, \sqcup, \delta, q_0, q_z), \quad (4.87)$$

s komponentama

$$Q := \bigcup_{i \leq 9} \{q_i\} \cup \bigcup_{i \leq n} \{p_i, s_i, t_i\} \cup \bigcup_{\alpha \in \Sigma_0} \{n_\alpha, m_\alpha, l_\alpha\} \cup \{q_x, q_z\}, \quad (4.88)$$

$$\Gamma := \Sigma_0 \cup \{\$, \#\} \cup B^m, \text{ gdje je } B^m := \prod_{i < m} \{\circ, \bullet\}, \quad (4.89)$$

$$\sqcup := (\circ, \circ, \dots, \circ)^T \in B^m \text{ (zapisana kao stupac)}, \quad (4.90)$$

i funkcijom prijelaza δ zadanom jednakostima (4.33)–(4.37), (4.40)–(4.47), (4.56)–(4.57), (4.59)–(4.65), (4.68)–(4.77) i (4.79)–(4.85), uz dodatnu konvenciju da se svi parovi $(q, \alpha) \in (Q \setminus \{q_z\}) \times \Gamma$ navedeni u tim jednakostima preslikavaju u $(q_x, \alpha, 1)$.

Doista, neka je $w \in \Sigma_0^*$ proizvoljna riječ, i označimo s $k := |w|$ njenu duljinu. Promotrimo prvo slučaj kada je $w \in \mathcal{D}_{\varphi_0}$.

Tada će početnu konfiguraciju $(q_0, 0, w_{\sqcup} \dots)$ prvi fragment od \mathcal{T}_0 (lema 4.25) prebaciti u $(q_2, k, \$w\#_{\sqcup} \dots)$, a nju će pak drugi fragment prebaciti u $(p_0, k + 2, \$_{\sqcup}^k \#r_1^{(w)}_{\sqcup} \dots)$, gdje je $r_1 = (\circ, \bullet, \circ, \dots, \circ)^{\top}$ (lema 4.27). Kako je $w \in \mathcal{D}_{\varphi_0}$, vrijedi $x := \langle w \rangle \in \mathcal{D}_{\mathbb{N}\varphi_0}$, pa P_0 -izračunavanje s x stane, i izlazni podatak mu je $y := \mathbb{N}\varphi_0(x) = \langle \varphi_0(w) \rangle$. Prema korolaru 4.36, \mathcal{T}_0 -izračunavanje s w (treća faza) će zato doći u konfiguraciju $(s_n, k + 2, \$_{\sqcup}^k \#t_{\sqcup} \dots)$, gdje t u tragu 0 ima oblik $\bullet^y \circ^{z-y}$. Tu će pak konfiguraciju četvrti fragment (lema 4.39) prebaciti u $(q_5, k + 1, \$_{\sqcup}^k \#uv\$_{\sqcup} \dots)$, gdje je $v = \varphi_0(w)$, te $u \in (B^m)^{z-|v|}$. Peti će fragment (lema 4.41) napokon prebaciti tu konfiguraciju u završnu konfiguraciju $(q_z, |v|, v_{\sqcup} \dots)$, gdje se na traci nalazi izlazni podatak $\varphi_0(w)$.

Ako pak $w \notin \mathcal{D}_{\varphi_0}$, tada će se prve dvije faze izvršiti jednako, ali iz $x \notin \mathcal{D}_{\mathbb{N}\varphi_0}$ slijedi da P_0 -izračunavanje s x nikad neće stati, pa po korolaru 4.36 niti \mathcal{T}_0 -izračunavanje neće doći do s_n . Prema napomeni 4.37, to znači da ono nikad neće stati (doći do q_z), jer jedini put od q_0 do q_z vodi kroz stanje s_n . \square

Primijetimo iznenađujuću sličnost upravo provedenog dokaza s dokazom propozicije 1.31 (semantika funkcijskog makroa). Kao što smo tamo omogućili makro-stroju da u izoliranoj okolini izvrši neki konkretni RAM-program, tako smo to ovdje omogućili Turingovom stroju. Kao i tamo (1.29), postupak se sastoji od pet faza: „otvaranje okvira” pomicanjem udesno, prijenos (kodiranje) argumenata u simulaciju (namještanje početne konfiguracije), sama simulacija (izvršavanje pojedinih instrukcija RAM-programa), te ako ona stane, prijenos (dekodiranje) njene povratne vrijednosti iz završne konfiguracije, i „zatvaranje okvira” (pospremanje nereda koji smo napravili).

Prikažimo postupak Turing-izračunavanja kroz faze (sa z je označen najveći broj u ikojem relevantnom registru od S_0 u završnoj konfiguraciji P_0 -izračunavanja s $\langle w \rangle$):

	stanje	pozicija	traka	
početna konfiguracija	q_0	0	$w_{\sqcup} \dots$	
nakon prve faze	q_2	$ w $	$\$w\#_{\sqcup} \dots$	
nakon druge faze	p_0	$ w + 2$	$\$_{\sqcup}^{ w } \#r_1^{(w)}_{\sqcup} \dots$	
nakon treće faze	s_n	$ w + 2$	$\$_{\sqcup}^{ w } \# \bullet^{(v)} ?^{z-(v)}_{\sqcup} \dots$	(4.91)
nakon četvrte faze	q_5	$ w + 1$	$\$_{\sqcup}^{ w } \# \circ^{z- v } v\$_{\sqcup} \dots$	
prvi korak pete faze	q_8	$ w + 2 + z - v $	$\$_{\sqcup}^{ w +1+z- v } v\$_{\sqcup} \dots$	
drugi korak pete faze	q_8	$ w + 2 + z$	$v_{\sqcup}^{ w +2+z- v } \$_{\sqcup} \dots$	
završna konfiguracija	q_z	$ v $	$v_{\sqcup} \dots$	

Primjer 4.44. Primjer 4.29 sad možemo pratiti kroz sve faze:

$$\text{ulaz} \quad \text{ab} \quad (4.92)$$

$$\text{uokvirivanje ulaza} \rightsquigarrow^* \underset{q_2}{\$ab\#} \quad (4.93)$$

$$\langle \text{ab} \rangle = (12)_2 = 4 \quad \rightsquigarrow^* \$ \text{ } \text{ } \text{ } \# \overset{\text{oooo}}{\underset{\text{p}_0}{\bullet \bullet \bullet \bullet}} \quad (4.94)$$

$$2 \cdot 4 + 1 = 9; \text{ raspisano u (4.67)} \quad \rightsquigarrow^* \$ \underbrace{\text{L L L}}_{S_5} \# \begin{matrix} \bullet \bullet \bullet \bullet \bullet \bullet \bullet \\ \circ \circ \circ \circ \circ \circ \circ \end{matrix} \quad (4.95)$$

$$9 = (121)_2 = \langle aba \rangle \quad \rightsquigarrow^* \$_{\text{uu}} \#_{\text{uuuuuu}} aba \quad (4.96)$$

$$\text{uokvirivanje izlaza} \quad \rightsquigarrow^* \quad \$\underbrace{\hspace{1.5cm}}_{q_8}\text{aba}\$ \quad (4.97)$$

$$\text{premještanje izlaza} \quad \rightsquigarrow^* \text{aba}_{\text{q}_8} \$ \quad (4.98)$$

$$\text{izlaz} \rightsquigarrow^* \text{aba}_{\text{q}_z} \quad (4.99)$$

Naravno, dodati **a** na kraj riječi Turingov stroj može puno jednostavnije — ali ovaj primjer zapravo pokazuje kako može računati bilo koju funkciju koju RAM-stroj može računati. Pravo računanje odvija se u primjeru 4.35 — ovo je samo hrpa „papirologije” prije i poslije, koju treba riješiti da bi taj primjer bio koristan. \triangleleft

Ukratko, dokazali smo da je za *jezične* funkcije izračunljivost u jezičnom modelu jednako snažna kao izračunljivost njihovih pratećih funkcija u brojevnom modelu. Primijetimo da smo time napokon opravdali neovisnost o korištenom *encodingu*, i riješili „filozofski problem” s početka točke 4.2.1.

Korolar 4.45. *Neka je Σ abeceda, te φ jezična funkcija nad njom. Tada parcijalna rekurzivnost funkcije $\mathbb{N}\varphi$ ne ovisi o izboru kodiranja $\mathbb{N}\Sigma$.*

Dokaz. Neka su $\mathbb{N}\Sigma$ i $\mathbb{N}'\Sigma$ dva kodiranja Σ . Ako je $\mathbb{N}\varphi_0$ parcijalno rekurzivna, tada je po teoremu 2.36 RAM-izračunljiva. Tada je po teoremu 4.43 (primijenjenom na kodiranje $\mathbb{N}\Sigma$) φ Turing-izračunljiva, a onda je po teoremu 4.24 (primijenjenom na kodiranje $\mathbb{N}'\Sigma$) $\mathbb{N}'\varphi$ parcijalno rekurzivna. Potpuno isto (zamjenom ta dva kodiranja) se dokaže drugi smjer, dakle $\mathbb{N}\varphi$ je parcijalno rekurzivna ako i samo ako je $\mathbb{N}'\varphi$ parcijalno rekurzivna. \square

4.4. Turing-izračunljivost brojevnihi funkcija

Sve dosad napravljeno može se iskazati u jednom vrlo općenitom obliku.

Propozicija 4.46. *Neka je Σ proizvoljna abeceda, i $\mathbb{N}\Sigma$ proizvoljno njeno kodiranje. Tada je $\varphi \mapsto \mathbb{N}\varphi$ bijekcija između skupa $\mathcal{T}\text{Comp}_{\Sigma}$ svih Turing-izračunljivih jezičnih funkcija nad Σ , i skupa Comp_1 jednomjesnih RAM-izračunljivih funkcija.*

Dokaz. Teorem 4.24 (zajedno s teoremom 2.36) kaže da za svaku $\varphi \in \mathcal{T}\text{Comp}_\Sigma$ vrijedi $\mathbb{N}\varphi \in \text{Comp}_1$. Dakle preslikavanje doista „ide kamo treba”.

Injektivnost je lako dokazati: neka su $\varphi_1, \varphi_2 \in \mathcal{T}\text{Comp}_\Sigma$ različite. Ako imaju različite domene, bez smanjenja općenitosti možemo pretpostaviti da postoji $w \in \mathcal{D}_{\varphi_1} \setminus \mathcal{D}_{\varphi_2}$. Tada je $\langle w \rangle \in \mathcal{D}_{\mathbb{N}\varphi_1}$, ali isto tako $\langle w \rangle \notin \mathcal{D}_{\mathbb{N}\varphi_2}$ jer je kodiranje riječi injekcija, pa $\langle w \rangle$ ne može biti jednak nijednom $\langle w' \rangle$ za $w' \in \mathcal{D}_{\varphi_2}$. To znači da prateće funkcije $\mathbb{N}\varphi_1$ i $\mathbb{N}\varphi_2$ imaju različite domene, pa su različite.

Ako pak φ_1 i φ_2 imaju istu domenu D , ali se razlikuju na nekoj riječi $w \in D$, tada (opet po injektivnosti kodiranja riječi) vrijedi

$$\mathbb{N}\varphi_1(\langle w \rangle) = \langle \varphi_1(w) \rangle \neq \langle \varphi_2(w) \rangle = \mathbb{N}\varphi_2(\langle w \rangle), \quad (4.100)$$

pa se $\mathbb{N}\varphi_1$ i $\mathbb{N}\varphi_2$ razlikuju na elementu $\langle w \rangle$, te su različite.

Za surjektivnost, uzmimo proizvoljnu $F^1 \in \text{Comp}_1$, i tražimo $\varphi \in \mathcal{T}\text{Comp}_\Sigma$ takvu da je $F = \mathbb{N}\varphi$. Očito, u domeni joj moraju biti upravo sve riječi w za koje je $\langle w \rangle \in \mathcal{D}_F$, te svaku takvu riječ mora preslikavati u (jedinственu) riječ v čiji kod je $F(\langle w \rangle)$. Time je jezična funkcija φ potpuno određena, te iz $\mathbb{N}\varphi = F \in \text{Comp}_1$ po teoremu 4.43 slijedi $\varphi \in \mathcal{T}\text{Comp}_\Sigma$. Dobivenu funkciju označavamo s $\mathbb{N}^{-1}F$. \square

Precizno, imamo vezu

$$F = \mathbb{N}\varphi = \mathbb{N}\Sigma^* \circ \varphi \circ (\mathbb{N}\Sigma^*)^{-1} \iff \varphi = \mathbb{N}^{-1}F = (\mathbb{N}\Sigma^*)^{-1} \circ F \circ \mathbb{N}\Sigma^* \quad (4.101)$$

te vrijedi $\mathbb{N}\mathbb{N}^{-1}F = F$ i $\mathbb{N}^{-1}\mathbb{N}\varphi = \varphi$.

Korolar 4.47. *Neka je Σ abeceda s kodiranjem $\mathbb{N}\Sigma$, i F^1 brojevena funkcija. Tada je F parcijalno rekurzivna ako i samo ako je $\mathbb{N}^{-1}F$ Turing-izračunljiva.*

Dokaz. Ako je $F = \mathbb{N}\mathbb{N}^{-1}F$ parcijalno rekurzivna, tada je po teoremu 2.36 RAM-izračunljiva, te je po teoremu 4.43 $\mathbb{N}^{-1}F$ Turing-izračunljiva. S druge strane, ako je $\mathbb{N}^{-1}F$ Turing-izračunljiva, onda je po teoremu 4.24 $\mathbb{N}\mathbb{N}^{-1}F = F$ parcijalno rekurzivna. \square

Vidimo da korolar 4.47 vrijedi bez obzira na abecedu i kodiranje. Važan specijalni slučaj dobijemo za jednočlanu abecedu, za koju je kodiranje jedinstveno i podudara se s duljinom (lema 4.38).

Definicija 4.48. *Unarna abeceda je $\Sigma_\bullet := \{\bullet\}$. Kad nema opasnosti od zabune, umjesto Σ_\bullet pišemo samo \bullet . Kodiranje unarne abecede je jedino moguće: $\mathbb{N}\Sigma_\bullet(\bullet) := 1$.*

Za jednomjesnu brojevnju funkciju F^1 , jezičnu funkciju $\mathbb{N}^{-1}F$ nad \bullet zovemo *unarnom reprezentacijom* od F , i označavamo je $\bullet F$. \triangleleft

Svaka riječ u nad unarnom abecedom ($u \in \bullet^*$) je oblika $u = \bullet^n$, gdje je $n = |u| = \langle u \rangle$. Sada definicija preslikavanja \mathbb{N}^{-1} kaže da je $\bullet F(\bullet^n) = \bullet^{F(n)}$ ako je $n \in \mathcal{D}_F$, a $\bullet^n \notin \mathcal{D}_{\bullet F}$ inače. Po napomeni 1.3, pišemo $\bullet F(\bullet^n) \simeq \bullet^{F(n)}$.

Primjer 4.49. $\bullet \text{factorial}(\bullet \text{prime}(\bullet)) = \bullet \text{factorial}(\bullet \bullet \bullet) = \bullet \bullet \bullet \bullet \bullet \bullet$, jer je $p_1! = 3! = 6$. Za inicijalne funkcije, $\bullet \text{Sc}(w) = w\bullet$, a $\bullet \text{Z}(w) = \varepsilon$ za sve $w \in \bullet^*$. \triangleleft

Korolar 4.50. *Neka je F^1 brojevena funkcija. Tada je F parcijalno rekurzivna ako i samo ako je $\bullet F$ Turing-izračunljiva.*

Dokaz. Već rekosmo, ovo je specijalni slučaj korolara 4.47, za unarnu abecedu. \square

4.4.1. Izračunljivost jezika

Neka je Σ proizvoljna abeceda, s b' znakova, $\mathbb{N}\Sigma$ njeno kodiranje, te $L \subseteq \Sigma^*$ jezik nad njom. Što bi značilo da je L izračunljiv?

Izračunljivost L u brojevnom modelu gledamo preko kodova: definiramo jednomjesnu brojevenu relaciju

$$\langle L \rangle := \{ \langle w \rangle \mid w \in L \} = \mathbb{N}\Sigma^*[L], \quad (4.102)$$

i prirodno je reći da L ima neko svojstvo izračunljivosti (npr. da je primitivno rekurzivan) ako relacija $\langle L \rangle$, odnosno njena karakteristična funkcija $\chi_{\langle L \rangle}$, ima to svojstvo.

Recimo, prazan jezik \emptyset je primitivno rekurzivan jer je $\chi_{\langle \emptyset \rangle} = \chi_{\emptyset} = \text{Z}$ primitivno rekurzivna (štoviše, inicijalna). Također, univerzalni jezik Σ^* je primitivno rekurzivan, jer je $\langle \Sigma^* \rangle = \mathbb{N}\Sigma^*[\Sigma^*] = \mathbb{J}_{\mathbb{N}\Sigma^*} = \mathbb{N}$, čija je karakteristična funkcija $\text{C}_1^\dagger = \text{Sc} \circ \text{Z}$.

U jezičnom modelu, moramo vidjeti što znači da neki Turingov stroj računa χ_L . Očito, to treba biti Turingov stroj nad Σ , i ulaz $w \in \Sigma^*$ mu se daje na isti način kao i običnom Turingovom stroju: kroz početnu konfiguraciju $(q_0, 0, w_\square \dots)$. No za izlaz (*true* ili *false*, ovisno o tome je li $w \in L$) postoji samo konačno mnogo mogućnosti, pa ga je prirodnije predstaviti stanjem. Zato takvi Turingovi strojevi imaju *dva* završna stanja, q_{true} i q_{false} , te njima signaliziraju je li riječ u jeziku ili nije (kažemo da *prihvaćaju* odnosno *odbijaju* riječ).

Na prvi pogled, to je slično stanjima q_z i q_x koje smo imali u našim Turingovim strojevima, samo što je propisano da konfiguracija sa stanjem q_{false} (kao i ona s q_{true}) prelazi u samu sebe, a ne po funkciji δ . Ali postoji jedna bitna razlika: kako su karakteristične funkcije nužno totalne, od takvih strojeva zahtijevamo da **uvijek stanu** (za svaki ulaz dostignu konfiguraciju s jednim od ta dva završna stanja). Za takve strojeve kažemo da su *odlučitelji* (*deciders*). Svaki odlučitelj \mathcal{T} tako dijeli Σ^* na dva dijela,

$$L(\mathcal{T}) := \{ w \in \Sigma^* \mid \mathcal{T}\text{-izračunavanje s } w \text{ stane u stanju } q_{\text{true}} \} \text{ i} \quad (4.103)$$

$$(L(\mathcal{T}))^c = \{ w \in \Sigma^* \mid \mathcal{T}\text{-izračunavanje s } w \text{ stane u stanju } q_{\text{false}} \}, \quad (4.104)$$

i kažemo da *prepoznaje* $L(\mathcal{T})$.

Nije preteško vidjeti da su te dvije karakterizacije, brojevena i jezična, povezane — pogotovo jer već imamo napravljen najveći dio posla.

Teorem 4.51. *Neka je L jezik. Ako postoji Turingov odlučitelj koji prepoznaje L , tada je jednomjesna relacija $\langle L \rangle$ rekurzivna.*

Dokaz. Pretpostavimo da je \mathcal{T} odlučitelj za L , i provedimo s tim strojem postupak iz točke 4.2.2. Ovdje navodimo samo detalje koje je potrebno promijeniti.

Prvo, kako imamo dva završna stanja, moramo fiksirati njihove kodove: recimo, $\mathbb{NQ}(q_{true}) := 1$, a $\mathbb{NQ}(q_{false}) := 2$. (Lako je dokazati analogon leme 4.16, da bez smanjenja općenitosti možemo pretpostaviti $q_0 \neq q_{true}$ i $q_0 \neq q_{false}$ — a po definiciji odlučitelja mora biti $q_{true} \neq q_{false}$.)

Drugo, kako nam δ sad nije definirana na oba završna stanja, treba promijeniti uvjet u lemi analognoj lemi 4.20, u $q \notin \{q_{true}, q_{false}\}$. Tu se ništa bitno ne mijenja, osim što će *direction*-tablica (vidjeti primjer 4.21) imati dva retka jedinica, a ne samo jedan.

I treće, umjesto parcijalno rekurzivne funkcije *stop*, imat ćemo funkciju zadanu sa

$$\text{stop}'(x) := \mu n (\text{State}(x, n) \in \{1, 2\}), \quad (4.105)$$

za koju lako vidimo da je rekurzivna. Naime, parcijalno je rekurzivna po propoziciji 2.39, jer je skup $\{1, 2\}$ primitivno rekurzivan (po korolaru 2.46), pa je dobivena minimizacijom primitivno rekurzivne relacije. Totalna je (pa smo u (4.105) mogli koristiti znak ‘:=’) po definiciji odlučitelja: ako je $x = \langle w \rangle$, tada mora biti $\text{State}(x, n) \in \{1, 2\}$ za neki n , jer \mathcal{T} -izračunavanje s w mora stati. Naravno, tada je $\text{stop}'(x) = \text{stop}'(\langle w \rangle)$ upravo broj koraka nakon kojeg to izračunavanje stane.

Tvrdimo da je

$$x \in \langle L \rangle \iff \text{State}(x, \text{stop}'(x)) = 1, \quad (4.106)$$

iz čega odmah slijedi rekurzivnost od $\langle L \rangle$ po lemi 2.31 i korolaru 2.33.

Doista, ako je $x \in \langle L \rangle = \mathbb{N}\Sigma^*[L]$, to znači da postoji $w \in L$ takva da je $x = \langle w \rangle$. Tada \mathcal{T} -izračunavanje s w mora stati u stanju q_{true} — označimo s n_0 broj koraka nakon kojeg se to dogodi. Tada je $\text{State}(x, n_0) = 1 \in \{1, 2\}$, dok za sve $n < n_0$ konfiguracija nakon n koraka nije završna (sasvim analogno propoziciji 1.12 — jer završne konfiguracije prelaze isključivo u same sebe, postoji najviše jedna završna konfiguracija u izračunavanju), pa vrijedi $\text{State}(x, n) \notin \{1, 2\}$. Dakle $n_0 = \text{stop}'(x)$, pa je $\text{State}(x, \text{stop}'(x)) = \text{State}(x, n_0) = 1$.

U drugom smjeru, pretpostavimo $\text{State}(x, \text{stop}'(x)) = 1$. Po propoziciji 4.14, postoji jedinstvena riječ $w \in \Sigma^*$ takva da je $x = \langle w \rangle$. \mathcal{T} je odlučitelj, pa \mathcal{T} -izračunavanje s w mora stati — označimo s n_0 broj koraka nakon kojeg se to dogodi. Tada je (kao i prije) $\text{State}(x, n) \notin \{1, 2\}$ za $n < n_0$, i očito $\text{State}(x, n_0) = 1 \in \{1, 2\}$, pa je $\text{stop}'(x) = n_0$. Sada pretpostavka glasi $\text{State}(x, n_0) = 1$, odnosno završno stanje je q_{true} , pa \mathcal{T} prihvaća w . To znači da je $w \in L$, odnosno $x = \langle w \rangle \in \langle L \rangle$. \square

Teorem 4.52. *Neka je L jezik (nad nekom abecedom Σ). Ako je relacija $\langle L \rangle$ rekurzivna, tada postoji Turingov odlučitelj za L .*

Dokaz. Pretpostavka zapravo znači da je $\chi_{\langle L \rangle}$ rekurzivna funkcija. Dakle $\chi_{\langle L \rangle}$ je parcijalno rekurzivna, pa po teoremu 2.36 postoji RAM-program koji je računa; i totalna je, pa taj RAM-program stane za svaki ulaz. Na taj RAM-program htjeli bismo primijeniti postupak iz točke 4.3. Da bismo to mogli, moramo $\chi_{\langle L \rangle}$ prikazati kao $\mathbb{N}\varphi$ za neku jezičnu funkciju φ nad Σ . Možemo li to?

Svakako: dokaz propozicije 4.46 kaže da je $\varphi := \mathbb{N}^{-1}\chi_{\langle L \rangle} = (\mathbb{N}\Sigma^*)^{-1} \circ \chi_{\langle L \rangle} \circ \mathbb{N}\Sigma^*$. Ta funkcija je totalna kao kompozicija tri totalne, te je Turing-izračunljiva prema korolaru 4.47 (naravno, uzeli smo isto kodiranje pomoću kojeg smo izračunali $\langle L \rangle$). Pogledajmo malo detaljnije kako ona djeluje.

Ako joj damo riječ $w \in \Sigma^* \setminus L$, tada (kodiranje riječi je injekcija) vrijedi $\langle w \rangle \notin \langle L \rangle$, pa je $\chi_{\langle L \rangle}(\langle w \rangle) = 0$. Po definiciji od φ je tada

$$\varphi(w) = (\mathbb{N}\Sigma^*)^{-1}(\chi_{\langle L \rangle}(\mathbb{N}\Sigma^*(w))) = (\mathbb{N}\Sigma^*)^{-1}(\chi_{\langle L \rangle}(\langle w \rangle)) = (\mathbb{N}\Sigma^*)^{-1}(0) = \varepsilon. \quad (4.107)$$

Dakle, riječi izvan L ona preslikava u praznu riječ. Kako je kodiranje injekcija, riječi unutar L ne smije preslikavati u praznu riječ, a jer je φ totalna, mora ih preslikavati nekamo. Dakle, $\varphi(w)$ je neprazna za svaku $w \in L$. Vidimo da, baš kao i u \mathbb{N} , imamo prirodnu reprezentaciju od bool u Σ^* : prazna riječ je lažna, neprazne su istinite. Svi programski jezici koji definiraju istinitost stringova, definiraju je upravo na taj način.

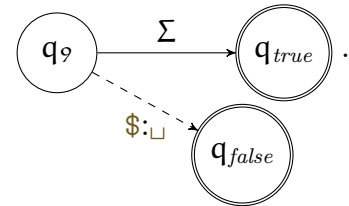
Naš Turingov stroj koji računa φ dobiven je transpiliranjem RAM-programa koji računa $\chi_{\langle L \rangle}$. U opisu pete (zadnje) faze rada tog Turingovog stroja, razlikovali smo slučajeve kad je izlazna riječ prazna i kad nije. Kao što smo upravo vidjeli, to nam može poslužiti za odabir završnog stanja u kojem ćemo završiti — samo zamijenimo prijelaze (4.84) i (4.85) s

$$\delta(q_?, \alpha) := (q_{true}, \alpha, 1), \text{ za sve } \alpha \in \Sigma, \quad (4.108)$$

$$\delta(q_?, \$) := (q_{false}, \sqcup, -1). \quad (4.109)$$

Po teoremu 4.43, \mathcal{T} će „računati” (pod navodnicima jer odlučitelji zapravo ne služe za računanje funkcija) funkciju φ , pa će za $w \in L$ rezultat biti neprazan, te će stroj u trećem koraku pete faze, krećući se lijevo, doći do znaka $\alpha \in \Sigma$ i ući u stanje q_{true} . Za $w \notin L$, rezultat će biti ε , te će stroj doći do graničnika, obrisati ga i ući u stanje q_{false} . Budući da za svaku riječ $w \in \Sigma^*$ vrijedi jedna od te dvije mogućnosti, zaključujemo da smo doista konstruirali odlučitelj. \square

Dijagramatski, gornji desni kut (4.86) zamijenimo s



4.4.2. Turing-izračunljivost višemjesnih funkcija

Dosadašnji rezultati pokazuju da je za jezične funkcije, i za *jednomjesne* brojevne funkcije, svejedno računamo li ih na Turingovom stroju ili na nekom brojevnom modelu izračunavanja (npr. RAM-stroju). Možemo li to isto dokazati i za *višemjesne* brojevne funkcije? Svakako, samo se prvo trebamo dogovoriti oko reprezentacije njihovog ulaza.

Već smo u uvodu nagovijestili, koristit ćemo kontrakciju: u abecedu ćemo dodati separator /, te ćemo više ulaznih podataka razdvojiti separatorima: npr. $(1, 0, 5, 0)$ prenijet ćemo kao $\bullet//\bullet\bullet\bullet\bullet\bullet/$.

Definicija 4.53. *Binarna abeceda* je abeceda $\Sigma_\beta := \{\bullet, /\}$. Za svaki neprazni konačni niz prirodnih brojeva $\vec{x} = (x_1, x_2, \dots, x_k)$, definiramo *binarnu reprezentaciju* kao

$$\beta(\vec{x}) := \bullet^{x_1} / \bullet^{x_2} / \dots / \bullet^{x_k} \in \Sigma_\beta^*. \quad (4.110)$$

Za svaki $k \in \mathbb{N}_+$, označavamo $\beta^k := \beta|_{\mathbb{N}^k}$. ◀

Ponekad se reprezentacija također zove „kodiranje”, ali nama su kodiranja funkcije čije povratne vrijednosti su prirodni brojevi. Funkcija β ide u suprotnom smjeru (s dobrim razlogom — sad trebamo promatrati jezične funkcije kao osnovne, jer za njih imamo teoreme 4.24 i 4.43), ali zapravo je možemo promatrati u bilo kojem smjeru.

Propozicija 4.54. *Funkcija β je bijekcija između \mathbb{N}^+ i Σ_β^* .*

Dokaz. Najlakše je konstruirati inverznu funkciju, i pokazati da je to doista inverz. Dakle, uzmimo proizvoljnu riječ $u \in \Sigma_\beta^*$, i pitamo se kojeg niza je ona kod. Kao i uvijek kod konačnih nizova, trebamo odrediti njegovu duljinu k , i zatim pojedine članove x_1, x_2, \dots, x_k . Duljina je očito sljedbenik broja pojavljivanja separatora / u riječi u (jer u (4.110) ima $k - 1$ separatora), i to je doista pozitivan broj, dakle niz je neprazan. Prvi član mu možemo odrediti brojeći kružice do prvog separatora (ili do kraja riječi ako je $k = 1$), drugi brojeći ih između prvog i drugog separatora, itd. Posljednji član x_k je broj kružica od zadnjeg separatora do kraja riječi u .

Tvrdimo da je tako konstruirano preslikavanje desni inverz od β : odnosno, ako tako dobijemo niz \vec{x} , tada je $\beta(\vec{x}) = u$. Doista, te dvije riječi imaju isti broj separatora ($k - 1$) i isti broj kružica ($\sum \vec{x}$), te se podudaraju na pozicijama svih separatora (malo pomaknute parcijalne sume od \vec{x}) — što je dovoljno da zaključimo da su jednake.

To je također i lijevi inverz: odnosno, ako primijenimo taj postupak na riječ $\beta(\vec{x})$, dobit ćemo upravo \vec{x} : imat će točnu duljinu k , i točan svaki član, jer između i -tog i $(i + 1)$ -vog separatora u (4.110) ima točno x_i kružica. □

Za same funkcije, koristit ćemo istu ideju kao u (4.15): reprezentaciju ulaza preslikamo u reprezentaciju izlaza (ako je izlaz definiran), dok ne-reprezentacije ne preslikamo nikamo. Ovdje treba biti oprezan zbog propozicije 4.54, ali brojevne funkcije imaju

fiksnu mjesnost. Zato ćemo ne-reprezentacijama za funkciju f^k proglasiti sve one riječi koje nisu reprezentacije k -torki (odnosno one koje su reprezentacije l -torki za $l \neq k$). Na isti način, na izlazu ćemo dati reprezentaciju samog broja ($k = 1$), dakle riječ bez separatora $\bullet^y = \beta(y) \in \beta[\mathbb{N}] = \mathcal{I}_{\beta^1} = \bullet^*$.

Definicija 4.55. Neka je $k \in \mathbb{N}_+$, i f^k brojevnja funkcija. Za jezičnu funkciju βf zadanu s

$$\beta f(u) := \beta(f(\bar{x})), \text{ za } u = \beta(\bar{x}^k) \quad (4.111)$$

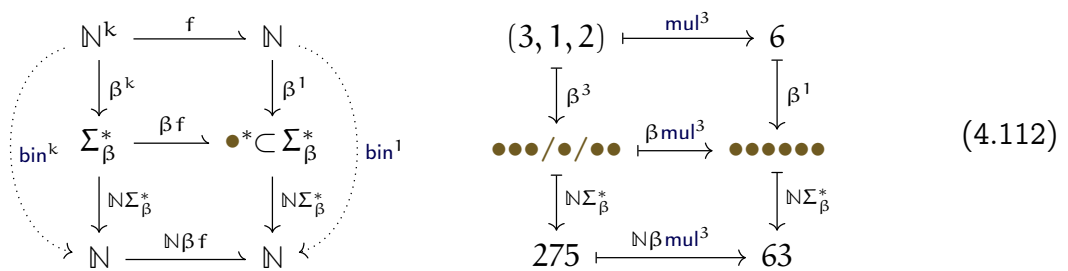
nad binarnom abecedom Σ_β , kažemo da je *binarna reprezentacija* funkcije f . \triangleleft

Vjerojatno ste nekad napisali Turingov stroj za βadd^2 , koristeći $\beta \text{add}^2(u/v) = uv$, ako su $u, v \in \bullet^*$. Ipak, već tada ste zasigurno vidjeli koliko je teško napisati βmul^2 , a pogotovo βpow — ostale divote iz poglavlja 3 ($\beta \text{part}^?$!) da i ne spominjemo.

Sljedeći veliki cilj je dokazati da Turingovi strojevi doista mogu računati (binarno reprezentirane) *sve* parcijalno rekurzivne funkcije, pa tako i funkciju βuniv — čineći tako jedan od modela univerzalne izračunljivosti. Ipak, prvo ćemo prvo dokazati obrat: brojevnje funkcije, čije su binarne reprezentacije Turing-izračunljive, su parcijalno rekurzivne. To nije toliko impresivan rezultat, ali lakši je za dokazati, a mnoge dijelove dokaza moći ćemo upotrijebiti i u dokazu obrata.

Dakle, neka je $k \in \mathbb{N}_+$, i f^k brojevnja funkcija takva da je βf Turing-izračunljiva. Fiksirajmo neko kodiranje $\mathbb{N}_{\Sigma_\beta}$ (pomaknuta baza $b' = 2$) — korolar 4.45 kaže da je svejedno koje kodiranje uzmemo, pa kako smo već definirali $\langle \bullet \rangle = 1$, samo dodefinirajmo $\langle / \rangle := 2$. Sada znamo, po teoremu 4.24, da je $\mathbb{N}\beta f$ parcijalno rekurzivna. Možemo li nekako iz toga dobiti da je f parcijalno rekurzivna? Drugim riječima, f^k i $(\mathbb{N}\beta f)^1$ su obje brojevnje funkcije, ali su povezane preko jezične funkcije βf . Možemo li nekako naći brojevnju vezu između njih? Precizno, možemo li naći brojevnje funkcije g^1 i h^k takve da je $f = g \circ \mathbb{N}\beta f \circ h$? Ako bi g i h bile izračunljive i totalne (recimo primitivno rekurzivne), iz toga bi odmah slijedila parcijalna rekurzivnost od f , jer je skup parcijalno rekurzivnih funkcija zatvoren na kompoziciju.

U traženju tih funkcija može pomoći dijagram.



Lijevo je općenita slika, desno je jedan konkretan primjer (računanje mul^3 na $(3, 1, 2)$). Brojevi desno dolje su razumljiviji u pomaknutoj bazi 2: $275 = (11121211)_2$, te $63 = (111111)_2$. Gornji pravokutnik predstavlja definiciju (4.111), dok donji predstavlja (4.15) za specijalni slučaj binarne abecede, i jezične funkcije βf nad njom.

Lema 4.56. *Dijagram prikazan lijevo u (4.112) (samo pune strelice) komutira.*

Dokaz. Drugim riječima, za svaki od ta dva pravokutnika, pa onda i za veliki pravokutnik sastavljen od ta dva, je svejedno kojim putem dođemo od njegovog gornjeg lijevog do donjeg desnog vrha.

Prvo, dakle, trebamo dokazati (gornji pravokutnik) da je $\beta f \circ \beta^k = \beta^1 \circ f$. Domena desne funkcije je \mathcal{D}_f jer je β^1 totalna, a domena lijeve funkcije je $\{\vec{x} \in \mathbb{N}^k \mid \beta^k(\vec{x}) \in \mathcal{D}_{\beta f}\}$, što je opet jednako \mathcal{D}_f po definiciji domene $\mathcal{D}_{\beta f}$. Neka je sad $\vec{x} \in \mathcal{D}_f$ proizvoljan. Na njemu je lijeva funkcija $\beta f(\beta^k(\vec{x}))$, što je po definiciji (4.111) jednako $\beta^1(f(\vec{x}))$, što je upravo jednako $(\beta^1 \circ f)(\vec{x})$. Dakle, $\beta f \circ \beta^k$ i $\beta^1 \circ f$ imaju istu domenu i na njoj se podudaraju, pa su to jednake funkcije.

Donji pravokutnik je puno jednostavniji jer je $\mathbb{N}\Sigma_\beta^*$ bijekcija: ako označimo $\varphi := \beta f$,

$$\mathbb{N}\beta f \circ \mathbb{N}\Sigma_\beta^* = \mathbb{N}\varphi \circ \mathbb{N}\Sigma_\beta^* = \mathbb{N}\Sigma_\beta^* \circ \varphi \circ (\mathbb{N}\Sigma_\beta^*)^{-1} \circ \mathbb{N}\Sigma_\beta^* = \mathbb{N}\Sigma_\beta^* \circ \varphi = \mathbb{N}\Sigma_\beta^* \circ \beta f. \quad (4.113)$$

Sada iz te dvije jednakosti slijedi

$$\mathbb{N}\Sigma_\beta^* \circ \beta^1 \circ f = \mathbb{N}\Sigma_\beta^* \circ \beta f \circ \beta^k = \mathbb{N}\beta f \circ \mathbb{N}\Sigma_\beta^* \circ \beta^k, \quad (4.114)$$

odnosno čitav „vanjski okvir” komutira. \square

Upravo dokazana jednakost je korisna, jer pruža način da se u potpunosti izbjegn timer jezične funkcije. Naime, vrhovi vanjskog pravokutnika su sasvim brojevni, i njegove stranice se mogu opisati kao brojeвне funkcije. Gornju i donju stranicu već imamo, još je preostalo precizirati lijevu i desnu.

4.4.3. Funkcije bin^k — binarno kodiranje

Definicija 4.57. Za svaki $k \in \mathbb{N}_+$, definiramo $\text{bin}^k := \mathbb{N}\Sigma_\beta^* \circ \beta^k$. \triangleleft

Te funkcije su brojeвне, i na dijagramu su prikazane točkanim strelicama: lijeva stranica vanjskog pravokutnika je bin^k , a desna bin^1 . Svaka bin^k je očito injekcija kao kompozicija dvije injekcije. Jesu li izračunljive bez pozivanja na jezični model?

Kako bismo iz $(3, 1, 2)$ izračunali $(11121211)_2 = 275$, ili iz 6 izračunali $(111111)_2 = 63$, koristeći samo brojeвне funkcije? Ovo drugo se svakako čini lakšim.

Lema 4.58. *Funkcija bin^1 je primitivno rekurzivna.*

Dokaz. Možemo direktno napisati točkovnu definiciju $\text{bin}(x) = \sum_{i < x} 2^i$, pa će primitivna rekurzivnost slijediti iz leme 2.50 — ali svaki računarac zna da to ima i zatvoreni oblik: $\text{bin}(x) = 2^x - 1 = \text{pd}(\text{pow}(2, x))$ (jer je uvijek $2^x \geq 1$). \square

Kako sada pomoću funkcije bin^1 možemo dobiti bin^2 ? Naravno, konkatencijom. Riječ $\beta(x, y) = \bullet^x / \bullet^y$ je sastavljena od tri dijela: $\bullet^x = \beta(x)$, $/$ i $\bullet^y = \beta(y)$, čije kodove znamo — to su redom $\text{bin}(x)$, 2 i $\text{bin}(y)$. Dakle, da dobijemo njen kod, trebamo ih konkatencirati u pomaknutoj bazi 2. Operacija je definirana s (4.12).

Propozicija 4.59. *Neka je Σ abeceda, $\mathbb{N}\Sigma$ njeno kodiranje, te b broj znakova u njoj. Tada za sve $u, v \in \Sigma^*$ vrijedi $\langle uv \rangle = \langle u \rangle \frown_b \langle v \rangle$.*

Dokaz. Uz oznake $u = \alpha_1 \alpha_2 \cdots \alpha_{|u|}$ i $v = \beta_1 \beta_2 \cdots \beta_{|v|}$, vrijedi

$$\begin{aligned} \langle uv \rangle &= \langle \alpha_1 \alpha_2 \cdots \alpha_{|u|} \beta_1 \beta_2 \cdots \beta_{|v|} \rangle = (\mathbb{N}\Sigma(\alpha_1) \cdots \mathbb{N}\Sigma(\alpha_{|u|}) \mathbb{N}\Sigma(\beta_1) \cdots \mathbb{N}\Sigma(\beta_{|v|}))_b = \\ &= (\mathbb{N}\Sigma(\alpha_1) \cdots \mathbb{N}\Sigma(\alpha_{|u|}))_b \frown_b (\mathbb{N}\Sigma(\beta_1) \cdots \mathbb{N}\Sigma(\beta_{|v|}))_b = \\ &= \langle \alpha_1 \cdots \alpha_{|u|} \rangle_b \frown_b \langle \beta_1 \cdots \beta_{|v|} \rangle_b = \langle u \rangle_b \frown_b \langle v \rangle. \end{aligned} \quad (4.115)$$

Za $u = \varepsilon$ ili $v = \varepsilon$ tvrdnja također vrijedi jer je ε neutralni element za konkatenciju, a lako se vidi da je $\langle \varepsilon \rangle = 0$ neutralni element za \frown_b :

$$0 \frown_b x = \text{sconcat}(0, x, b) = 0 \cdot b^{\text{slh}(x, b)} + x = 0 + x = x, \quad (4.116)$$

$$x \frown_b 0 = \text{sconcat}(x, 0, b) = x \cdot b^{\text{slh}(0, b)} + 0 = x \cdot b^0 = x, \quad (4.117)$$

jer je $\text{slh}(0, b) = (\mu t \leq 0)(\sum_{i \leq t} b^i > 0) = 0$ zbog $\sum_{i \leq 0} b^i = b^0 = 1 > 0$. \square

Korolar 4.60. *Za sve $x, y \in \mathbb{N}$, za sve $b \in \mathbb{N}_+$, vrijedi $\text{slh}(x \frown_b y, b) = \text{slh}(x, b) + \text{slh}(y, b)$.*

Dokaz. Neka su x, y, b proizvoljni (b pozitivan). Abeceda $\Sigma_b := [1..b]$ očito ima b elemenata — kodirajmo je identitetom. Po propoziciji 4.14, postoje $u, v \in \Sigma_b^*$ takve da je $x = \langle u \rangle$ i $y = \langle v \rangle$. Po definiciji slh , vrijedi $\text{slh}(\langle w \rangle, b) = |w|$ za sve $w \in \Sigma_b^*$, pa specijalno za $w = x$, $w = y$ i $w = xy$. Sada imamo

$$\begin{aligned} \text{slh}(x, b) + \text{slh}(y, b) &= \text{slh}(\langle u \rangle, b) + \text{slh}(\langle v \rangle, b) = |u| + |v| = |uv| = \text{slh}(\langle uv \rangle, b) = \\ &[\text{propozicija 4.59}] = \text{slh}(\langle u \rangle \frown_b \langle v \rangle, b) = \text{slh}(x \frown_b y, b) \quad \square. \end{aligned} \quad (4.118)$$

Propozicija 4.61. *Za svaki $b \in \mathbb{N}_+$, \frown_b je primitivno rekurzivna asocijativna operacija.*

Dokaz. Primitivna rekurzivnost slijedi direktno iz leme 4.13. Za asocijativnost,

$$\begin{aligned} (x \frown_b y) \frown_b z &= (x \cdot b^{\text{slh}(y, b)} + y) \cdot b^{\text{slh}(z, b)} + z = \\ &= (x \cdot b^{\text{slh}(y, b)} \cdot b^{\text{slh}(z, b)} + y \cdot b^{\text{slh}(z, b)}) + z = x \cdot b^{\text{slh}(y, b) + \text{slh}(z, b)} + (y \cdot b^{\text{slh}(z, b)} + z) = \\ &[\text{korolar 4.60}] = x \cdot b^{\text{slh}(y \frown_b z, b)} + (y \frown_b z) = x \frown_b (y \frown_b z) \quad \square. \end{aligned} \quad (4.119)$$

Zbog asocijativnosti ćemo ubuduće pisati izraze poput $x \frown_b y \frown_b z$ bez zagrada. Primijetimo samo da je za asocijativnost ključno da se radi o pomaknutoj bazi: recimo, u običnoj bazi 10 je $(1 \frown_{10}' 0) \frown_{10}' 2 = 10 \frown_{10}' 2 = 102 \neq 1 \frown_{10}' (0 \frown_{10}' 2) = 1 \frown_{10}' 2 = 12$, pa ne vrijedi asocijativnost.

Propozicija 4.62. *Za svaki $k \in \mathbb{N}_+$, funkcija bin^k je primitivno rekurzivna.*

Dokaz. Matematičkom indukcijom po k . Za $k = 1$ (baza), to je upravo lema 4.58. Pretpostavimo da je bin^l primitivno rekurzivna za neki $l \in \mathbb{N}_+$. Tada je

$$\begin{aligned} \text{bin}^{l+1}(\vec{x}, y) &= \langle \beta(\vec{x}, y) \rangle = \langle \bullet^{x_1} / \dots / \bullet^{x_l} / \bullet^y \rangle = \langle \beta(\vec{x}) / \beta(y) \rangle = \\ &[\text{propozicija 4.59}] = \langle \beta(\vec{x}) \rangle \hat{\ } \langle / \rangle \hat{\ } \langle \beta(y) \rangle = \text{bin}^k(\vec{x}) \hat{\ } 2 \hat{\ } \text{bin}^l(y), \end{aligned} \quad (4.120)$$

te primitivna rekurzivnost od bin^{l+1} slijedi iz pretpostavke indukcije, propozicije 4.61 i leme 4.58. \square

Promotrimo sada dijagram lijevo u (4.112), u svjetlu definicije 4.57, odnosno gledajući i točkane strelice. Definicija 4.57 zapravo kaže da „kružni odsječak” sa svake strane tog dijagrama komutira, pa iz toga i leme 4.56 slijedi da čitav „vanjski oval” također komutira.

Korolar 4.63. *Neka je $k \in \mathbb{N}_+$, te f^k funkcija. Tada vrijedi $\text{bin}^l \circ f = \mathbb{N}\beta f \circ \text{bin}^k$.*

Dokaz. To je jednakost (4.114) u koju je uvrštena (s obje strane) definicija 4.57. \square

Sjetimo se, naš cilj je izraziti f pomoću $\mathbb{N}\beta f$. S korolarom 4.63 smo već jako blizu, treba nam samo još (izračunljiv) lijevi inverz za bin^1 . Kako možemo iz $63 = (111111)_2 = \text{bin}(6)$ natrag dobiti 6? Samo treba prebrojiti jedinice, odnosno znamenke.

Lema 4.64. *Definiramo funkciju blh s $\text{blh}(n) := \text{slh}(n, 2)$.*

Funkcija blh je primitivno rekurzivna, i vrijedi $\text{blh} \circ \text{bin}^1 = I_1^1$.

Dokaz. Primitivna rekurzivnost slijedi iz simboličke definicije $\text{blh} = \text{slh} \circ (I_1^1, C_2^1)$, leme 4.13 i propozicije 2.19. Za kompoziciju, uzmimo proizvoljni $n \in \mathbb{N}$, i računamo $\text{blh}(\text{bin}(n)) = \text{blh}(2^n - 1)$. Relacija koju minimiziramo (po $t \leq 2^n - 1$) je

$$\sum_{i \leq t} 2^i > 2^n - 1 \iff 2^{t+1} - 1 > 2^n - 1 \iff t + 1 > n \iff t \geq n \quad (4.121)$$

pa je očito najmanji takav t upravo $n = I_1^1(n)$. \square

Teorem 4.65. *Neka je $k \in \mathbb{N}_+$, i f^k funkcija takva da je βf Turing-izračunljiva. Tada je f parcijalno rekurzivna.*

Dokaz. Prema teoremu 4.24, primijenjenom na abecedu Σ_β , kodiranje $\bullet \mapsto 1, / \mapsto 2$ i funkciju βf , funkcija $\mathbb{N}\beta f$ je parcijalno rekurzivna. Iz leme 4.64 i korolara 4.63 sada imamo

$$f = I_1^1 \circ f = \text{blh} \circ \text{bin}^1 \circ f = \text{blh} \circ \mathbb{N}\beta f \circ \text{bin}^k, \quad (4.122)$$

odnosno f je dobivena kompozicijom iz tri funkcije, od kojih je srednja parcijalno rekurzivna, a prva i zadnja su primitivno rekurzivne (lema 4.64 i propozicija 4.62), pa su rekurzivne (korolar 2.33) a time i parcijalno rekurzivne. Kako je skup parcijalno rekurzivnih funkcija zatvoren na kompoziciju, f je parcijalno rekurzivna. \square

Sad je jasno da isti komutirajući dijagram može poslužiti i za dokazivanje obrata teorema 4.65. Jednadžba

$$\text{bin}^1 \circ f = \mathbb{N}\beta f \circ \text{bin}^k \quad (4.123)$$

iz korolara 4.63 može poslužiti i za dobivanje $\mathbb{N}\beta f$ iz f — samo trebamo umjesto desne obrnuti lijevu stranicu pravokutnika, odnosno naći desni inverz za bin^k . Nažalost, to je iz tri razloga zapetljaniije nego ovo što smo napravili u prethodnoj točki.

Prvi razlog je sasvim birokratski, i tiče se naše odluke sa samog početka, da ćemo promatrati brojevne funkcije s jednim izlaznim podatkom. Kako je $\mathcal{D}_{\text{bin}^k} = \mathbb{N}^k$, traženi desni inverz bi morao imati k izlaznih podataka, te ćemo ga reprezentirati kroz k brojevnihi funkcija $\text{arg}_1, \text{arg}_2, \dots, \text{arg}_k$ — tako nazvanihi jer ekstrahiraju pojedine argumente funkcije f iz jedinog argumenta a funkcije $\mathbb{N}\beta f$. (Te funkcije bit će neovisne o k : recimo, $\text{arg}_2(\text{bin}^2(x, y)) = \text{arg}_2(\text{bin}^4(x, y, z, t)) = y$, pa ih nećemo morati označavati s dva broja, kao što smo recimo morali označavati koordinatne projekcije l_n^k .)

Drugi razlog je tehnički: implementacija je bitno kompliciranija nego u lemi 4.64. Na neki način, dok su riječi nad jednočlanom abecedom trivijalno izomorfne s \mathbb{N} ($w \mapsto |w|$ je izomorfizam), riječi nad dvočlanom abecedom imaju bogatiju strukturu, i za njeno raščlanjivanje treba nam neka vrst „standardne biblioteke” za rad sa stringovima. Ovdje ćemo napraviti samo minimum potreban za primitivnu rekurzivnost funkcija arg_i , ali lako je vidjeti kako tu ima puno mogućnosti za implementaciju raznih algoritama.

Treći razlog, napokon, sasvim je stvaran i nije tako lako rješiv programskim tehnikama: funkcija bin^k nije surjekcija, i kao takva *nema* totalni desni inverz. Precizno, $\mathbb{N}\beta f$ nije jednaka $\text{bin}^1 \circ f \circ (\text{arg}_1, \dots, \text{arg}_k)$, već je restrikcija te funkcije na J_{bin^k} . Recimo u slučaju kad a ima 5 znamenaka 2 u pomaknutoj bazi 2, vrijednost $\mathbb{N}\beta \text{mul}^3(a)$ nije definirana, dok je $2^{\text{arg}_1(a) \cdot \text{arg}_2(a) \cdot \text{arg}_3(a)} = 1$ sasvim definirano. Srećom, imamo tehniku (restrikcija na rekurzivan skup) koja će uskladiti domene.

4.4.4. Standardna biblioteka za Σ_β^*

Propozicija 4.66. *Za riječi $v, w \in \Sigma_\beta^*$ kažemo da je v prefiks od w ako postoji riječ u takva da je $w = vu$. Ta relacija je refleksivni parcijalni uređaj na skupu Σ_β^* .*

Dokaz. Za refleksivnost, dovoljno je staviti $u := \varepsilon$. Za tranzitivnost, ako je $w = vu$ i $v = v'u'$, tada je $w = (v'u')u = v'(u'u)$, pa je v' prefiks od w .

Za antisimetričnost, ako je $w = vu$ i $v = wu'$, tada je kao za tranzitivnost $w = w(uu')$, te vrijedi $|w| = |w(uu')| = |w| + |uu'|$, iz čega $|uu'| = 0$. Kako je ε jedina riječ duljine 0, imamo $uu' = \varepsilon$, i analogno $u = u' = \varepsilon$, dakle $w = vu = v\varepsilon = v$. \square

Korolar 4.67. *Dvomisna brojevna relacija \sqsubseteq_β , zadana s*

$$x \sqsubseteq_\beta y :\iff \text{„}x \text{ je kod nekog prefiksa riječi čiji kod je } y\text{”} \quad (4.124)$$

(kodovi su po $\mathbb{N}\Sigma_\beta^$), primitivno je rekurzivan refleksivni parcijalni uređaj na \mathbb{N} .*

Dokaz. Činjenica da se radi o parcijalnom uređaju slijedi iz propozicija 4.66 i 4.14, a primitivna rekurzivnost iz leme 4.13: $x \sqsubseteq_{\beta} y \iff x = \text{sprefix}(y, \text{blh}(x), 2)$. \square

Primjer 4.68. Recimo, vrijedi $4 = (12)_2 = \langle \bullet / \rangle \sqsubseteq_{\beta} \langle \bullet / \bullet \bullet \bullet \rangle = (12111)_2 = 39$. \triangleleft

Za sljedeću lemu, trebat će nam *uokvireni* brojevi: oni čiji zapisi u pomaknutoj bazi 2 imaju bar dvije znamenke, te počinju i završavaju znamenkom 2. Naime, ono što zapravo želimo napraviti je „isprogramirati” dokaz propozicije 4.54, no taj dokaz ima nekoliko slučajeva ovisno o tome nalazimo li se prije prvog separatora, nakon zadnjeg, ili između njih. Jednostavnije je kad riječ ima separatore na lijevom i na desnom kraju: tada svaki x_i možemo odrediti brojeći jedinice između susjednih separatora.

Lema 4.69. *Postoje primitivno rekurzivne funkcije pos2 i streak1 , takve da za svaki uokvireni broj x , za svaki $i \in \mathbb{N}$, vrijedi:*

$\text{pos2}(x, i)$ je pozicija i -te dvojke (ili $\text{blh}(x)$ ako takva ne postoji), te

$\text{streak1}(x, i)$ je duljina i -tog niza uzastopnih jedinica (ili 0 ako takav ne postoji), počevši od $i = 0$ slijeva, u zapisu broja x u pomaknutoj bazi 2.

Dokaz. Za početak, definirajmo pomoćnu primitivno rekurzivnu (lema 2.52) funkciju koja broji dvojke (separatore) u zapisu broja u pomaknutoj bazi 2.

$$\text{count2}(x) := (\#i < \text{blh}(x)) (\text{sdigit}(x, i, 2) = 2) \quad (4.125)$$

Tvrdimo da funkcije zadane točkovno s

$$\text{pos2}(x, i) := \text{blh}((\mu z < x)(z \hat{<} 2 \sqsubseteq_{\beta} x \wedge \text{count2}(z) = i)), \quad (4.126)$$

$$\text{streak1}(x, i) := \text{pos2}(x, \text{Sc}(i)) \ominus \text{Sc}(\text{pos2}(x, i)), \quad (4.127)$$

zadovoljavaju tražene specifikacije (očito su primitivno rekurzivne).

Za $i < \text{count2}(x)$, uvjet pod operatorom minimizacije jednoznačno određuje z . Doista, kad bi postojala dva broja z i z' s tim svojstvom, morali bi biti kodovi različitih riječi v i v' , takvih da su $v/$ i $v'/$ prefiksi od w , čiji kod je x . Prefiksi iste riječi iste duljine su jednaki (sprefix je funkcija), pa duljine od $v/$ i $v'/$ moraju biti različite: bez smanjenja općenitosti pretpostavimo $|v/| < |v'/| = |v'| + 1$, dakle $v/$ je zapravo prefiks od v' , pa mora imati manje ili jednako dvojki. Dakle

$$\begin{aligned} i = \text{count2}(z') &= \text{count2}(\langle v' \rangle) \geq \text{count2}(\langle v/ \rangle) = \text{count2}(\langle v \rangle \hat{<} \langle / \rangle) = \\ &= \text{count2}(\langle v \rangle \hat{<} 2) = \text{count2}(\langle v \rangle) + 1 = \text{count2}(z) + 1 = i + 1, \end{aligned} \quad (4.128)$$

kontradikcija. No ako je z jedinstven, to znači da je dovoljno naći *neki* takav i taj će biti minimalan, a „prefiks” od x do isključivo pozicije i -te dvojke zadovoljava to svojstvo. Njegova duljina je onda upravo ta pozicija (brojeći od 0), kao što i treba biti.

Za $i \geq \text{count2}(x)$, takav z ne postoji (jer $z \hat{>} 2$ ima više dvojki nego x , pa ne može biti $z \hat{>} 2 \sqsubseteq_{\beta} x$), te minimizacija ($\mu z < x$) daje x , odnosno $\text{pos2}(x, i) = \text{blh}(x)$, opet, kao što i treba biti po specifikaciji.

Sada je za funkciju **streak1** dovoljno oduzeti odgovarajuće pozicije: poziciju prvog znaka nakon i -te dvojke, od pozicije sljedeće dvojke. Za $i < \text{count2}(x) - 1$, to funkcionira točno kako treba: kako pomaknuta baza 2 ima samo znamenke 1 i 2, između susjednih dvojki nalaze se samo jedinice.

Za $i = \text{count2}(x) - 1$, i -ta dvojka je upravo zadnja, pa je $\text{pos2}(x, i) = \text{blh}(x) - 1$ (možemo pisati obične minuse jer uokvireni brojevi imaju bar dvije dvojke — na početku i na kraju). Također po specifikaciji **pos2** vrijedi $\text{pos2}(x, i + 1) = \text{blh}(x)$, pa je $\text{streak1}(x, i) = \text{blh}(x) \ominus \text{Sc}(\text{blh}(x) - 1) = 0$.

Za $i \geq \text{count2}(x)$ imamo $\text{pos2}(x, i + 1) = \text{pos2}(x, i) = \text{blh}(x)$, pa je opet $\text{streak1}(x, i) = \text{blh}(x) \ominus \text{Sc}(\text{blh}(x)) = 0$. \square

Sada je za ekstrakciju pojedinog x_i samo potrebno uokviriti $\text{bin}^k(\vec{x})$ dvojkama, i pozvati funkciju **streak1** s odgovarajućim argumentom.

Propozicija 4.70. *Za svaki $n \in \mathbb{N}_+$ postoji primitivno rekurzivna funkcija \arg_n , takva da za sve $\vec{x} = (x_1, \dots, x_k) \in \mathbb{N}^+$, za sve $i \in \mathbb{N}_+$ vrijedi $\arg_i(\text{bin}(\vec{x})) = \begin{cases} x_i, & i \leq k \\ 0, & \text{inače} \end{cases}$.*

Dokaz. Za svaki $n \in \mathbb{N}_+$ vrijedi $n - 1 \in \mathbb{N}$, pa definiramo

$$\arg_n(x) := \text{streak1}(2 \hat{>} x \hat{>} 2, n - 1). \quad (4.129)$$

Neka je $\vec{x} \in \mathbb{N}^k$ proizvoljan ($k \in \mathbb{N}_+$). Tada u $\beta(\vec{x})$ ima točno $k - 1$ separatora, pa je $\text{count2}(\text{bin}(\vec{x})) = k - 1$, odnosno za $y := 2 \hat{>} \text{bin}(\vec{x}) \hat{>} 2$ vrijedi $\text{count2}(y) = k - 1 + 2 = k + 1$. Sada za svaki $i \in [1..k]$ vrijedi $\arg_i(\text{bin}(\vec{x})) = \text{streak1}(y, i - 1)$, što je upravo x_i (recimo, za $i = 2 < k$ imat ćemo $\text{streak}(\langle \bullet^{x_1} / \bullet^{x_2} / \dots / \bullet^{x_k} \rangle, 1) = \text{pos2}(y, 2) - \text{Sc}(\text{pos2}(y, 1)) = (1 + x_1 + 1 + x_2) - \text{Sc}(1 + x_1) = x_2$). Za $i > k$ imat ćemo $i - 1 \geq k = \text{count2}(y) - 1$, pa će po lemi 4.69 biti $\text{streak1}(y, i - 1) = 0$, kao što i treba. \square

Teorem 4.71. *Za svaku parcijalno rekurzivnu funkciju f , βf je Turing-izračunljiva.*

Dokaz. Označimo s $k \in \mathbb{N}_+$ mjesnost od f . Cilj nam je dokazati da je $\mathbb{N}\beta f$ parcijalno rekurzivna. Tu funkciju možemo dobiti (iz dijagrama) kao $\text{bin}^1 \circ f \circ (\arg_1, \dots, \arg_k)$, restringiranu na sliku od bin^k . Ta slika je primitivno rekurzivan skup, jer samo treba provjeriti da njegov element ima točno $k - 1$ dvojku ($k - 1 \in \mathbb{N}$ je konstanta). Dakle,

$$\mathbb{N}\beta f(x) \simeq \text{bin}^1(f(\arg_1(x), \dots, \arg_k(x))), \text{ ako je } \text{count2}(x) = k - 1 \quad (4.130)$$

iz čega slijedi, po korolaru 3.63, da je $\mathbb{N}\beta f$ parcijalno rekurzivna. Sada je po teoremu 2.36, $\mathbb{N}\beta f$ RAM-izračunljiva, te je po teoremu 4.43 funkcija βf Turing-izračunljiva. \square

Poglavlje 5.

Neodlučivost

Dokazali smo brojne teoreme koji pokazuju ekvivalentnost različitih modela izračunljivosti. Za početak ih sve objedinimo na jednom mjestu. Najvažniji je svakako onaj koji govori o brojevnim funkcijama, odnosno karakterizira skup Comp .

Teorem 5.1 (Teorem ekvivalencije za brojevne funkcije). *Neka je $k \in \mathbb{N}_+$, te $f: \mathbb{N}^k \rightarrow \mathbb{N}$ brojevena funkcija mjesnosti k . Tada su sljedeće tvrdnje ekvivalentne:*

- (r) f je RAM-izračunljiva ($f \in \text{Comp}_k$);
- (m) f je makro-izračunljiva (postoji makro-algoritam koji računa f);
- (p) f je parcijalno rekurzivna (f je dobivena iz inicijalnih pomoću konačno mnogo primjena kompozicije, primitivne rekurzije i minimizacije);
- (i) f ima indeks (postoji $e \in \mathbb{N}$ takav da je $f = \{e\}^k$);
- (t) βf je Turing-izračunljiva (postoji Turingov stroj koji računa βf).

Dokaz. Sve bitno u ovom teoremu smo već dokazali. Ponovimo samo glavne ideje.

- (r \Rightarrow m) Ovo je trivijalno: $\text{Prog} \subset \text{MProg}$ (svaki RAM-program je ujedno i makro-program) — napomena 1.20.
- (m \Rightarrow r) Svaki makro-program $Q \in \text{MProg}$ ekvivalentan je svojem spljoštenju $Q^b \in \text{Prog}$ — teorem 1.26.
- (p \Rightarrow r) *Postorder*-obilaskom njene simboličke definicije, kompajliramo funkciju f u RAM-program — teorem 2.36.
- (r \Rightarrow i) Ako RAM-program $P \in \text{Prog}$ računa f , tada je njegov kod $e := \ulcorner P \urcorner \in \text{Prog}$ jedan indeks za f — propozicija 3.57.
- (i \Rightarrow p) Funkcija f^k s indeksom e dobivena je kompozicijom comp_k , konstante C_e^k i koordinatnih projekcija — korolar 3.54.

($t \Rightarrow p$) Funkciju f možemo dobiti kompozicijom iz parcijalno rekurzivnih funkcija blh , $\mathbb{N}\beta f$ i bin^k — teorem 4.65.

($p \Rightarrow t$) Funkciju $\mathbb{N}\beta f$ možemo dobiti restrikcijom na \mathcal{I}_{bin^k} kompozicije funkcija bin^1 , f i arg_1, \dots, arg_k — teorem 4.71.

Dijagramatski, usmjeren graf  je jako povezan. □

Za jednomjesne funkcije imamo korolar 4.50, koji je zgodniji zbog jednočlane abecede i bijektivnosti kodiranja. Za dokaz ekvivalencije parcijalne rekurzivnosti i Turing-izračunljivosti koristili smo ekvivalenciju brojevnog i jezičnog modela za jezične funkcije.

Teorem 5.2 (Teorem ekvivalencije za jezične funkcije). *Neka je $\Sigma \neq \emptyset$ abeceda, te $\varphi: \Sigma^* \rightarrow \Sigma^*$ jezična funkcija nad njom. Tada su sljedeće tvrdnje ekvivalentne:*

(T) φ je Turing-izračunljiva;

(P) $\mathbb{N}\varphi$ je parcijalno rekurzivna;

(R) $\mathbb{N}\varphi$ je RAM-izračunljiva.

Dokaz. Napomenimo, druga i treća stavka zapravo predstavljaju po $(\text{card } \Sigma)!$ tvrdnji — po jednu za svako kodiranje od Σ — ali sve su one ekvivalentne po korolaru 4.45.

Opet, sve bitno je već dokazano; slijedi rekapitulacija glavnih ideja.

($T \Rightarrow P$) Kodirajući sve dijelove Turing-izračunavanja s rječju zadanog koda, dobijemo parcijalnu rekurzivnost funkcije $\mathbb{N}\varphi$ — teorem 4.24.

($P \Rightarrow R$) Jednostavno kompajliramo funkciju $\mathbb{N}\varphi$ u RAM-program — teorem 2.36.

($R \Rightarrow T$) Transpiliranjem RAM-programa koji računa $\mathbb{N}\varphi$, u pet faza dobijemo Turingov stroj koji računa φ — teorem 4.43. □

Karakteristična funkcija je totalna, dakle za izračunljive jezike mora biti rekurzivna.

Teorem 5.3 (Teorem ekvivalencije za jezike). *Neka je $\Sigma \neq \emptyset$ abeceda, te $L \subseteq \Sigma^*$ jezik nad njom. Tada su sljedeće tvrdnje ekvivalentne:*

(R) relacija $\langle L \rangle := \{\langle w \rangle \mid w \in L\}$ je rekurzivna;

(O) postoji Turingov odlučitelj $(Q, \Sigma, \Gamma, \sqcup, \delta, q_0, q_{true}, q_{false})$ koji prepoznaje L .

Dokaz. Kao i u prethodnom teoremu, stavka (R) je zapravo $(\text{card } \Sigma)!$ tvrdnji. One su ekvivalentne po tranzitivnosti, jer je svaka od njih ekvivalentna sa stavkom (O).

- ($\mathbb{R} \Rightarrow 0$) Malom modifikacijom kraja konstrukcije transpiliranog Turingovog stroja koji računa $\mathbb{N}^{-1}\chi_{\langle L \rangle}$ dobijemo odlučitelj — teorem 4.52.
- ($0 \Rightarrow \mathbb{R}$) Malom modifikacijom kodiranja stanja, funkcije prijelaza i Turing-izračunavanja odlučitelja dobijemo rekurzivnost od $\langle L \rangle$ — teorem 4.51. \square

Još uvijek ostaje otvoreno pitanje što je s jezicima koje prepoznaju općeniti Turingovi strojevi, koji mogu i zapeti u beskonačnoj petlji (štoviše, kažemo da je ulazna riječ u jeziku Turingovog stroja ako i samo ako se to ne dogodi). Prema definiciji 4.5, ti jezici su zapravo *domene izračunljivih jezičnih funkcija*. Njihov status razriješit ćemo u točki 7.4.

Napomena 5.4. Formalno, morali bismo još osigurati da im traka na kraju bude oblika $v\sqcup\dots$ za $v \in \Sigma^*$, ali nije teško vidjeti, tehnikama koje se obrade u proučavanju formalnih jezika, da za svaki Turingov stroj (koji ne računa nužno neku funkciju) postoji Turingov stroj koji stane za točno iste ulaze, ali s praznom trakom ($v = \varepsilon$).

Skica dokaza: uvedemo novi znak $/$ u radnu abecedu i proglasimo ga prazninom (znak \sqcup time postaje običan znak radne abecede), te za svaki prijelaz koji čita bivšu prazninu $\delta(p, \sqcup) = (q, \gamma, d)$ dodamo prijelaz $\delta(p, /) = (q, \gamma, d)$ (pisanja praznina ostavimo na \sqcup). Semantički, i \sqcup i $/$ su sada praznine, ali stroj ne može napisati $/$ na traku — odnosno, jedine pojave znaka $/$ na traci su tamo gdje stroj još nije bio. Kako su pomaci mogući samo na susjedne ćelije, nosač takve trake je nužno povezan.

Sada na bivše završno stanje dodamo fragment koji odlazi desno do prvog znaka $/$ (zna da su nadalje samo praznine), i vraća se lijevo, zamjenjujući sve znakove $s /$. Kada u tom stanju pročita $/$, stane jer je došao do lijevog ruba trake (taj $/$ sigurno nije mogao prije zapisati). \triangleleft

Već smo rekli da domene izračunljivih brojevničkih funkcija ne moraju biti izračunljive — recimo, $\text{HALT} = \mathcal{D}_{\text{univ}}$ nije izračunljiva, iako to još nismo dokazali — pa je za očekivati da to vrijedi i za općenite Turing-prepoznatljive jezike. Preciznije ćemo taj fenomen opisati kad uvedemo rekurzivno prebrojive relacije.

5.1. Church–Turingova teza

Nakon toliko dokazanih implikacija oblika „ako je funkcija izračunljiva u nekom modelu, onda je izračunljiva i u drugom modelu”, počinjemo uočavati određene obrasce u definicijama i dokazima.

Algoritam \mathcal{A} je uglavnom opisan pomoću stroja (diskretnog dinamičkog sustava) s prebrojivim skupom mogućih konfiguracija C , tako da se svaka konfiguracija $c \in C$ može opisati s konačno (ali ne unaprijed ograničeno) mnogo bitova.

Ako s A označimo skup mogućih ulaznih podataka, a s B skup mogućih izlaznih podataka algoritma \mathcal{A} , imamo injekciju $\text{start}: A \rightarrow C$, koja svakom ulazu $x \in A$ pridružuje početnu konfiguraciju s ulazom x . Imamo još zadan podskup $E \subseteq C$ završnih konfiguracija, i surjekciju $\text{result}: E \rightarrow B$ (često zadanu na čitavom skupu C) koja svakoj završnoj konfiguraciji pridružuje rezultat izračunavanja.

Također imamo dvomjesnu relaciju \leadsto na skupu C , koja za determinističke algoritme (kakve jedino promatramo) ima funkcijsko svojstvo — pa je možemo promatrati i kao funkciju $\text{nextconf}: C \rightarrow C$, koja svakoj konfiguraciji stroja pridružuje sljedeću konfiguraciju po trenutnom koraku algoritma \mathcal{A} . Završne konfiguracije pritom ostaju nepromijenjene: restrikcija nextconf na skup E mora biti identiteta.

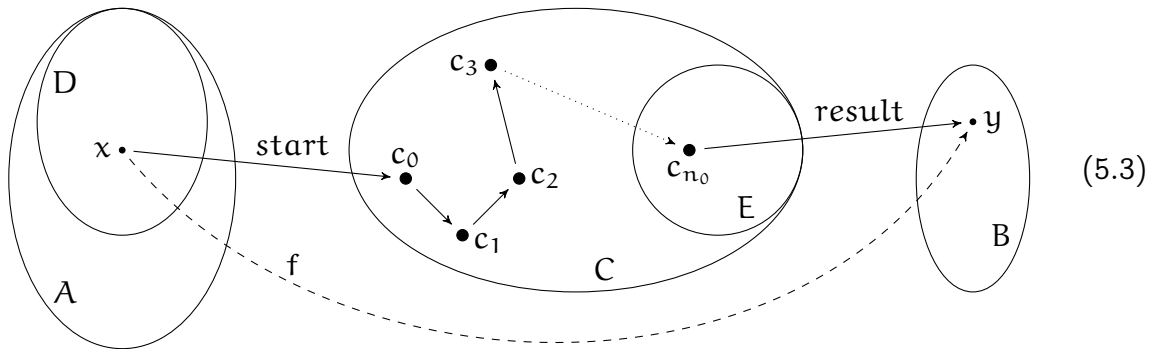
Iteriranjem funkcije nextconf na vrijednosti funkcije start dobivamo izračunavanje. Precizno, za $x \in A$ definiramo

$$c_0 := \text{start}(x), \quad (5.1)$$

$$c_{n+1} := \text{nextconf}(c_n), \quad (5.2)$$

i time smo dobili niz $(c_n)_n: \mathbb{N} \rightarrow C$ koji zovemo \mathcal{A} -izračunavanje s x .

Ako postoji n_0 takav da je $c_{n_0} \in E$, izračunavanje $(c_n)_n$ stane, i $y := \text{result}(c_{n_0})$ je njegov rezultat (to ne ovisi o izboru n_0 , jer je $\text{nextconf}|_E = \text{id}_E$, pa je završna konfiguracija jedinstvena). Ako s $D \subseteq A$ označimo skup svih ulaznih podataka s kojima \mathcal{A} -izračunavanje stane, time smo dobili funkciju $f: D \rightarrow B$ koju \mathcal{A} računa.



Ako sad između istih skupova A i B stavimo još neki drugi skup konfiguracija C' , sa svojom funkcijom $\text{nextconf}'$ i podskupom završnih konfiguracija E' , možemo zamisliti funkciju start kao zadatak f' za taj novi stroj, sa $A' := A$ i $B' := C$. No kako taj stroj zna računati samo funkcije iz A u B , moramo prethodno kodirati C nekom funkcijom $\text{Code}_1: C \rightarrow B$. To je zapravo najzanimljiviji dio, jer o njemu ovisi koliko će laki biti kasniji koraci. Dakle $f' = \text{Code}_1 \circ \text{start}$, i kako su te dvije funkcije obično prilično jednostavne, za očekivati je da će f' biti izračunljiva u novom modelu.

Napomena 5.5. Zanimljiv i relativno novi uvid Jurija Gureviča s kraja prošlog stoljeća je da se „univerzalni” skup C može naći u obliku interpretacija za strukture prvog reda.

Tada možemo i formalno, koristeći *lokalnost*, definirati što znači „prilično jednostavna” funkcija: to je ona koja ovisi o konačno mnogo „elemenata” (vrijednosti zatvorenih terma) u interpretaciji. Taj uvid vodi na formalizaciju *strojeva s apstraktnim stanjima* (ASM), koji u nekom smislu predstavljaju „univerzalnu podlogu” za opis algoritama. Lijep uvod u ASM-teoriju, pisan specifično za računarce, može se naći u [6]. \triangleleft

Analogni postupak možemo napraviti za *result*, samo s druge strane; i za *nextconf*, kodiranjem s obje strane. Svaka od tako dobivenih funkcija sama za sebe je dovoljno jednostavna (u smislu prethodne napomene) da je izračunljiva u novom modelu.

No to zapravo znači da sustav C' može simulirati svaki korak \mathcal{A} -izračunavanja, pa može simulirati i čitavo to izračunavanje. Samo treba unutar tog sustava imati neki način za:

- određivanje c_n iz n , iteriranjem funkcije *nextconf* (ograničena petlja);
- otkrivanje n_0 ako postoji (neograničena petlja); te
- dobivanje funkcije f iz tako dobivene funkcije $c_0 \mapsto c_{n_0}$, start i *result* (slijed).

Sada jasnije vidimo i zašto je skup \mathbb{N} „poseban”: dok A , B i C mogu biti svakakvi prebrojivi skupovi, domena izračunavanja (kao niza konfiguracija) mora biti \mathbb{N} . Drugim riječima, za bilo kakvo meta-programiranje (koje je nužno za univerzalnu funkciju) model u kojem radimo, pored reprezentacije ulaznih i izlaznih podataka, mora moći reprezentirati i prirodne brojeve. Onda je svakako najjednostavnije da ulazni i izlazni podaci također budu prirodni brojevi.

Odgovarajućim kodiranjima možemo dobiti i simulaciju za promijenjene skupove A' i B' — baš kao što smo to već činili kad je jedan od tih skupova, ili oba, bio C (za funkcije *start*, *result* i *nextconf*).

Zato nije čudno da sve formalizacije algoritama koje se mogu svesti na paradigmu shematski prikazanu u (5.3), mogu simulirati jedna drugu. Možemo ići toliko daleko da kažemo da je **prikazivost u obliku takve sheme** ono esencijalno što algoritme čini **algoritmima**.

Iako je to intuitivno jasno, iz toga nipošto ne slijedi da je svaki model jednostavno svesti na tu shemu. Recimo, možete se zabaviti pokušavajući konstruirati stroj — sustav $(C, \text{start}, \text{nextconf}, E, \text{result})$ — koji računa parcijalno rekurzivne funkcije zadane simboličkim definicijama (ili još kompliciranije, točkovnim definicijama). Koliko god taj zadatak bio prekriven debelim slojem tehničkih detalja, vjerojatno ćete se složiti da su to *samo* tehnički detalji — izuzetno biste se iznenadili da nekim slučajem ispadne da je takav stroj nemoguće konstruirati, zar ne? Upravo ta intuicija je ono što opravdava Church–Turingovu tezu.

5.1.1. Fiksiranje pojma algoritma

Zapravo, postoji širok (i višedimenzionalan!) „prostor kompromisa” (*tradeoffs*) prilikom dizajniranja takvih sustava. Na jednom kraju su sasvim jednostavni „konkretni” sustavi za koje je odmah jasno što su konfiguracije i kako je funkcija `nextconf` zadana — recimo, RAM-strojevi. Na drugom kraju su sasvim apstraktni sustavi kod kojih su tri nabrojene esencijalne operacije (ograničena i neograničena petlja, te slijed) aksiomatski propisane — recimo, u sustavu parcijalno rekurzivnih funkcija, redom kao primitivna rekurzija, minimizacija i kompozicija.

Osim s obzirom na apstraktnost, možemo gledati i kompromise s obzirom na druge dimenzije: smanjivanje broja različitih koncepata u sustavu (*Occamova britva*) vodi na sustave u kojima je gotovo sve prikazano prirodnim brojevima (jer, kao što smo rekli, prirodne brojeve moramo imati za reprezentaciju samog izračunavanja). Intuitivna bliskost onom što doista činimo kad provodimo algoritam vodi na sustave koji emuliraju naše stanje uma, i mentalne operacije (pamćenja simbola i jednostavnih odluka na nižoj razini, ili uočavanja obrazaca i vizualizacije njihove supstitucije na višoj razini apstrakcije) — kao što su jezični modeli izračunavanja.

algoritam	konkretna	apstraktna	(5.4)
jezični (praktični)	Turingov stroj	λ -račun	
brojevnii (teorijski)	RAM-stroj	parcijalna rekurzivnost	

Naravno, tablica (5.4) je vrlo pojednostavljena — em postoji puno više dimenzija po kojima možemo uspoređivati algoritamske formalizme (strukturalnost, primjenjivost na stvarne probleme, entropija, količina pretpostavljenog znanja, ...), em prikazane dimenzije nisu binarne. Pored jezičnog i brojevnog modela postoje razni drugi, recimo geometrijski (Wangove domine) ili algebarski (diofantski skupovi), pa čak i topološki (homotopska teorija tipova); dok je konkretno–apstraktno „podjela” zapravo spektar duž kojeg se mogu smjestiti razni koncepti. Recimo, makro-stroj je mrvicu apstraktniji od RAM-stroja, dok je logika prvog reda malo konkretnija od λ -računa. Čak ni sustavi koje smo obradili nisu ekstremi na tom spektru: postoje i sustavi poput Gödel–Herbrandovih jednadžbi, koji su još apstraktniji od parcijalne rekurzivnosti, kao i Minskijevi brojači (*counter machines*) koji su još konkretniji od RAM-strojeva. Već smo spomenuli ASM u napomeni 5.5 — taj model se proteže duž cijelog spektra konkretno–apstraktno, jer je njegova osnovna odlika da hvata algoritme na njihovoj prirodnoj razini apstrakcije, koja god ona bila.

Ono što je sada bitno je da između ćelija takve tablice možemo slobodno prolaziti kako nam je drago, dok imamo odgovarajuće teoreme ekvivalencije. Za pojedini redak (vrstu ulaznih odnosno izlaznih podataka) to se može sasvim formalizirati, u smislu da su izračunljive funkcije doslovno iste bez obzira na to koji stupac odaberemo. Za različite retke, moramo se dogovoriti oko prirodnog kodiranja, no to je najčešće prilično

jednostavno i dosta robusno s obzirom na tehničke detalje (recimo, zapis broja u bazi je vrlo prirodna veza brojevnog i jezičnog modela, i pritom za samu izračunljivost čak nije bitno — iako za složenost jest — odaberemo li unarni zapis ili zapis u bazi $b \geq 2$).

Church–Turingova teza. *Svaka dovoljno općenita formalizacija pojma algoritma na istoj vrsti podataka računa iste funkcije, a na različitim vrstama podataka računa funkcije koje su vezane prirodnim kodiranjima između tih vrsta podataka. To su upravo izračunljive funkcije, one za koje u intuitivnom smislu postoje algoritmi.*

Teoremi ekvivalencije mogu se sada izreći kao: sve ćelije u tablici (5.4) jesu dovoljno općenite formalizacije pojma algoritma. (Nismo definirali λ -račun, ali i za njega vrijedi teorem ekvivalencije. Zainteresirani čitatelj može naći detalje u [7].)

Povijesno, Alonzo Church je autor teze, i prvi koji je primijetio da je teza potrebna kako bi se dokazali rezultati o nepostojanju algoritma. Ipak, njegova formalizacija (λ -račun) je bila preapstraktna da bi zadovoljila vodeće eksperte za izračunljivost toga doba. To je između ostalog potaklo Turinga na detaljnu analizu intuitivnog pojma algoritma, te je njegova formalizacija, kad je dokazana ekvivalentnom ostalima do tad poznatima, vrlo brzo prihvaćena kao „mjera” (*Turing completeness*) pomoću koje se procjenjuju algoritamski sustavi (npr. programski jezici) sve do danas.

Church: *“Turing’s notion made the identification with effectiveness in the ordinary (not explicitly defined) sense evident immediately.”*

Gödel: *“The correct definition of mechanical computability was established beyond any doubt by Turing.”*

Church–Turingova teza nije uobličena kao teorem, pa čak niti kao tvrdnja koju bi se u principu moglo dokazati. U moderno vrijeme postoje i inicijative [14] da se ona uobliči kao *definicija*, fiksirajući jedan koncept izračunljivosti.

Definicija 5.6 (Soare). Za jezičnu funkciju φ kažemo da je *izračunljiva* ako postoji Turingov stroj koji je računa. Za brojevnju funkciju f kažemo da je *izračunljiva* ako je njena binarna reprezentacija βf izračunljiva. ◁

U tom svjetlu, teoremi ekvivalencije bili bi *karakterizacije* upravo definiranog pojma izračunljivosti. Ipak, u uvodnom kursu je vjerojatno bolje vidjeti interakciju raznih formalizacija, prije nego što se jedna od njih odabere kao definicija. Iako je modernom fizičaru sasvim prirodna definicija „metar je udaljenost koju svjetlost u vakuumu prijeđe za $\frac{1}{299\,792\,458}$ s”, to nije dobra definicija u osnovnoškolskoj nastavi fizike, jer pretpostavlja puno znanja i iskustva o prirodi svjetlosti, inercijalnim sustavima, i teoriji relativnosti. Bez tog znanja definicija se također može prihvatiti, ali djeluje forsirano. Autorovo mišljenje je da bi jednako tako djelovala i definicija 5.6 prije dokaza teorema ekvivalencije. Smijete se usprotiviti tom uvjerenju — ono nije matematičko, već psihološko.

5.2. Neizračunljive funkcije

Intuitivni pojam izračunljivog kao „onog za što postoji algoritam”, zajedno s Church–Turingovom tezom, napokon daju mogućnost da dokažemo da za neke probleme ne postoje algoritmi. To ćemo učiniti dokazom da neka funkcija g nije parcijalno rekurzivna — tada po teoremu ekvivalencije njena binarna reprezentacija nije Turing-izračunljiva, po definiciji 5.6 g nije izračunljiva, te možemo reći da za nju ne postoji algoritam.

Doduše, obično se problemi za koje se utvrđuje postojanje algoritma iskazuju u obliku *problema odluke*: za dani skup A i njegov podskup D , postoji li algoritam koji odlučuje, za proizvoljni $x \in A$, je li $x \in D$? Jednom kada imamo kodiranje $\mathbb{N}A$ skupa A , to gledamo kao pitanje izračunljivosti skupa (jednomjesne relacije) $\mathbb{N}A[D] \subseteq \mathbb{N}$, odnosno njene karakteristične funkcije $\chi_{\mathbb{N}A[D]}$. Sve te ideje smo već vidjeli (za $A := \Sigma^*$) kad smo promatrali odlučive jezike.

Definicija 5.7. Za brojevni problem D kažemo da je *odlučiv* ako je njegova karakteristična funkcija χ_D rekurzivna. Za problem $D \subseteq A$ takav da postoji prirodno kodiranje $\mathbb{N}A: A \rightarrow \mathbb{N}$, kažemo da je *odlučiv* ako je brojevni problem $\mathbb{N}A[D]$ odlučiv. \triangleleft

„Brojevni problemi” nisu ništa drugo nego relacije, ali postavljeni u obliku pitanja, odnosno ispitivanja nekog svojstva. Recimo, $\text{Halt}_2 = \mathcal{D}_{\text{comp}_2}^3$ možemo promatrati kao problem „za zadane $x, y, e \in \mathbb{N}$, (postoji li i) stane li program koda e s ulazom (x, y) ?”, odnosno „ima li izraz $\{e\}(x, y)$ smisla?”. Već smo nagovijestili da taj problem nije odlučiv. Kako bismo to dokazali? Uzmimo za početak nešto lakše. U uvodu smo već spomenuli paradoks sličan Russellovom, koji nastaje ako pretpostavimo da su svi algoritmi totalni: poredali smo sve jednomjesne algoritme u niz $(\mathcal{A}_n)_n$, i onda za ulaz n primijenili \mathcal{A}_n na n , i vratili sljedbenik izlaznog podatka. Pomoću indeksa možemo to doslovno napraviti.

Definicija 5.8. *Russellova funkcija* je jednomjesna funkcija [Russell](#)¹ zadana s

$$\text{Russell}(n) := \text{comp}_1(n, n) + 1. \quad (5.5)$$

Russellov skup je domena Russellove funkcije: $K^1 := \mathcal{D}_{\text{Russell}}$. \triangleleft

Korolar 5.9. *Russellova funkcija je parcijalno rekurzivna.*

Dokaz. Točkovna definicija (5.5) se lako zapiše u simboličkom obliku kao kompozicija $\text{Russell} = \text{Sc} \circ \text{comp}_1 \circ (I_1^1, I_1^1)$, pa tvrdnja slijedi po propoziciji 3.48. \square

Paradoks iz uvoda sada se može iskazati ovako: imamo algoritam za [Russell](#), ali nijedan algoritam koji računa vrijednosti od [Russell](#) nije totalan. Drugim riječima, Russellova funkcija je izračunljiva, ali ne postoji njeno izračunljivo totalno proširenje.

Lema 5.10. *Ne postoji rekurzivna funkcija r takva da je $r|_K = \text{Russell}$.*

Dokaz. Pretpostavimo da takva funkcija postoji. Tada je ona parcijalno rekurzivna, pa po teoremu ekvivalencije ima indeks, označimo ga s e (vidjeti napomenu 3.56); i totalna je, pa postoji $q := r(e) \in \mathbb{N}$. Sada je sljedbenik od q jednak

$$1 + q = 1 + r(e) = 1 + \{e\}(e) = 1 + \text{comp}(e, e), \quad (5.6)$$

iz čega slijedi da je $e \in K$ i $\text{Russell}(e) = 1 + q$. No $r(e) = q \neq 1 + q$, pa r ne može biti proširenje od Russell , kontradikcija. \square

Teorem 5.11. *Russellov skup K nije rekurzivan.*

Dokaz. Pretpostavimo suprotno, i označimo s $\widetilde{\text{Russell}} := \{K : \text{Russell}, Z\}$ proširenje Russellove funkcije nulom. Točkovno, imamo

$$\widetilde{\text{Russell}}(n) := \begin{cases} \text{Russell}(n), & n \in K \\ 0, & \text{inače} \end{cases}. \quad (5.7)$$

Po korolaru 5.9 Russell je parcijalno rekurzivna, po pretpostavci suprotnog je K rekurzivna relacija, a Z je inicijalna pa je trivijalno parcijalno rekurzivna. Prema teoremu 3.61, iz toga bi slijedilo da je $\widetilde{\text{Russell}}$ parcijalno rekurzivna funkcija.

S druge strane, po definiciji 2.43 je $R_0 := K^c$, pa je

$$\mathcal{D}_{\widetilde{\text{Russell}}} = (\mathcal{D}_Z \cap R_0) \cup (\mathcal{D}_{\text{Russell}} \cap K) = (\mathbb{N} \cap K^c) \cup (K \cap K) = K^c \cup K = \mathbb{N}, \quad (5.8)$$

dakle $\widetilde{\text{Russell}}$ je totalna funkcija. To, zajedno s parcijalnom rekurzivnošću, značilo bi da je $\widetilde{\text{Russell}}$ rekurzivna funkcija, što je u kontradikciji s lemom 5.10 (očito je $\widetilde{\text{Russell}}|_K = \text{Russell}$). \square

Time smo napokon dokazali da postoji izračunljiva funkcija s neizračunljivom domenom, i vidimo u čemu je problem s našom intuicijom: iako proširenje nulom doživljavamo kao trivijalnu operaciju na brojevnoj funkciji, za izračunljivost \tilde{f} je potrebno ustanoviti kad ulazni podatak x nije u domeni \mathcal{D}_f (da bismo ga preslikali u 0) — a to zapravo zahtijeva da ustanovimo da *nijedna* konfiguracija stroja koji računa f na x nije završna.

To je osnovna nesimetrija u definiciji algoritma: dok je zaustavljanje algoritma moguće karakterizirati neograničenom egzistencijalnom kvantifikacijom (projekcijom), što jest domena izračunljive funkcije (minimizacije), njegovo *nezaustavljanje* nije moguće tako karakterizirati. Ukratko, prirodni brojevi imaju lijepo svojstvo *početne konačnosti*: od svakog prirodnog broja ima konačno mnogo manjih, i ako postoji prirodni broj s nekim svojstvom, prateći sljedbenike od nule stići ćemo do njega u konačno mnogo koraka. S druge strane (doslovno!), prirodnih brojeva ima beskonačno mnogo: od svakog broja ima beskonačno mnogo većih, i ako trebamo ustanoviti da neko svojstvo vrijedi za sve prirodne brojeve, to nikako ne možemo učiniti provjeravajući

ih jedan po jedan. Čak i da ih ne provjeravamo po redu, do svakog trenutka smo provjerili samo konačno mnogo njih, te od svih njih postoji najveći, a većih od njega ima jednako mnogo kao i svih prirodnih brojeva.

Naravno, to nije *dokaz* da ne postoji možda neka druga metoda kojom bismo ustanovili da $x \notin \mathcal{D}_f$ — samo pokazuje da stvari nisu tako jednostavne.

5.2.1. Svođenje i problemi zaustavljanja

Problem kojim smo se bavili — za zadani n , je li n u domeni od $\{n\}^1$ — čini se vrlo umjetnim: nije da bismo ikad bili doista zainteresirani za općeniti odgovor na to pitanje. Ipak, jednom kad imamo neki neodlučiv problem, možemo se dočepati i ostalih.

Metoda je ista ona pomoću koje iz već poznatih odlučivih problema (izračunljivih funkcija) dobivamo nove: *svođenje* (redukcija). Ako možemo neku funkciju f svesti na neku drugu g — napisati f pomoću kompozicije, i eventualno primitivne rekurzije, funkcije g s nekim izračunljivim funkcijama, tada znamo da ako je g izračunljiva, tada je i f takva. To smo koristili mnogo puta u funkcijskom programiranju. Doduše, tamo nam je bila bitna primitivna rekurzivnost, pa smo govorili samo o totalnim funkcijama, i primitivna rekurzija je igrala bitnu ulogu. Kad govorimo o (ne)odlučivim problemima, primitivna rekurzivnost postaje luksuz koji si ne možemo uvijek priuštiti, pa ćemo zapravo promatrati samo kompoziciju. Kako se radi o problemima, dakle karakterističnim funkcijama relacija koje su totalne, zbog marljive evaluacije proizlazi da sve funkcije u kompoziciji moraju također biti totalne.

Napomenimo samo da u literaturi postoje razne vrste svođenja problema, pa i općenitih funkcija. Mi ćemo odabrati najjednostavniju koja je dovoljno dobra za naše potrebe.

Definicija 5.12. Neka su $k, l \in \mathbb{N}_+$, te R^k i P^l relacije. Kažemo da je R *svediva* na P , i pišemo $R \preceq P$, ako postoje rekurzivne funkcije $G_1^k, G_2^k, \dots, G_l^k$ takve da vrijedi $\chi_R = \chi_P \circ (G_1, G_2, \dots, G_l)$. \triangleleft

Zadnja jednakost se može točkovno zapisati kao $R(\vec{x}) \Leftrightarrow P(G_1(\vec{x}), \dots, G_l(\vec{x}))$. Primijetimo da je to zapravo obična kompozicija $\chi_R = \chi_P \circ G$, samo uzevši u obzir da je kodomena od G jednaka \mathbb{N}^l , pa je prikazujemo kroz l koordinatnih funkcija — što je već standardno kad se o kompoziciji radi.

Također primijetimo da nismo ništa dokomponirali na χ_P slijeva: nije dozvoljen *post-processing* povratne vrijednosti od χ_P , samo *pre-processing* njenih ulaznih podataka. Kako se radi o karakterističnim funkcijama čije su povratne vrijednosti iz $\text{bool} = \{0, 1\}$, jedini netrivialni *post-processing* bi moglo biti negiranje (komplementiranje), ali kako smo upravo vidjeli da komplementiranje općenito nije tako jednostavna operacija na problemima (npr. na domenama izračunljivih funkcija, odnosno problemima zaustavljanja), s razlogom ga nećemo htjeti ovdje.

Propozicija 5.13. *Relacija \preceq (na skupu $\bigcup_{k \in \mathbb{N}_+} \mathcal{P}(\mathbb{N}^k)$ svih brojevnih relacija) je refleksivna i tranzitivna.*

Dokaz. Refleksivnost je jednostavna: stavimo $G_1 := I_1^1, G_2 := I_2^1, \dots, G_l := I_l^1$ (koordinatne funkcije identitete $\text{id}_{\mathbb{N}^l}$). Očito je $\chi_P \circ (I_1^1, \dots, I_l^1) = \chi_P$, pa je $P \preceq P$.

Tranzitivnost je kompliciranija, ali samo notacijski, zbog toga što funkcije s više izlaza promatramo kroz koordinatne funkcije. Neka su $k, l, m \in \mathbb{N}_+$, te neka su R^k, Q^m i P^l relacije takve da je $R \preceq Q \preceq P$. Prvo svođenje znači da postoje rekurzivne funkcije G_1^k, \dots, G_m^k takve da je $\chi_R = \chi_Q \circ (G_1, \dots, G_m)$, dok drugo znači da postoje rekurzivne funkcije H_1^m, \dots, H_l^m takve da je $\chi_Q = \chi_P \circ (H_1, \dots, H_l)$. Sada je (možemo pisati jednakosti jer su sve funkcije ili karakteristične ili rekurzivne, dakle totalne, a kompozicija totalnih je totalna po propoziciji 2.6)

$$\begin{aligned} \chi_R(\vec{x}) &= \chi_Q(G_1(\vec{x}), \dots, G_m(\vec{x})) = (\chi_P \circ (H_1, \dots, H_l))(G_1(\vec{x}), \dots, G_m(\vec{x})) = \\ &= \chi_P(H_1(G_1(\vec{x}), \dots, G_m(\vec{x})), \dots, H_l(G_1(\vec{x}), \dots, G_m(\vec{x}))) = \\ &= \chi_P((H_1 \circ (G_1, \dots, G_m))(\vec{x}), \dots, (H_l \circ (G_1, \dots, G_m))(\vec{x})) = \\ &= (\chi_P \circ (H_1 \circ (G_1, \dots, G_m), \dots, H_l \circ (G_1, \dots, G_m))) (\vec{x}), \end{aligned} \quad (5.9)$$

dakle ako za svaki $i \in [1..l]$ definiramo $F_i := H_i \circ (G_1, \dots, G_m)$, tada je svaka F_i rekurzivna po lemi 2.31. Sada (5.9) možemo zapisati kao $\chi_R = \chi_P \circ (F_1, \dots, F_l)$, odakle slijedi $R \preceq P$. \square

Relacija $R \preceq P$ znači da ako imamo algoritam za rješavanje problema P , pomoću njega možemo riješiti i problem R . Intuitivno, R je „barem toliko odlučiv” koliko i P .

Propozicija 5.14. *Relacija svediva na rekurzivnu relaciju je i sama rekurzivna.*

Dokaz. Neka su $k, l \in \mathbb{N}_+$, te R^k relacija i P^l rekurzivna relacija, takve da je $R \preceq P$. To znači $\chi_R = \chi_P \circ (G_1, \dots, G_l)$, a rekurzivnost od P znači da je χ_P rekurzivna, pa je po lemi 2.31 i χ_R rekurzivna, odnosno R je rekurzivna. \square

Korolar 5.15. *Ako je $R \preceq P$ i R nije odlučiv, tada ni P nije odlučiv.*

Dokaz. Ovo je samo kontrapozicija propozicije 5.14, iskazana jezikom problema. \square

Korolar 5.15, kao što smo već nagovijestili, pruža način da pomoću jednog neodlučivog problema dođemo do ostalih.

Propozicija 5.16. *Problemi Halt_1 i HALT nisu odlučivi.*

Dokaz. Tvrdimo da vrijedi $K(n) \Leftrightarrow \text{Halt}_1(n, n)$. Doista,

$$\begin{aligned} K &= \mathcal{D}_{\text{Russell}} = \mathcal{D}_{\text{Sc} \circ \text{comp}_1 \circ (I_1^1, I_1^1)} = \mathcal{D}_{\text{comp}_1 \circ (I_1^1, I_1^1)} = \\ &= \{n \mid (I_1^1(n), I_1^1(n)) \in \mathcal{D}_{\text{comp}_1}\} = \{n \mid (n, n) \in \text{Halt}_1\}, \end{aligned} \quad (5.10)$$

gdje smo koristili definiciju (2.2), i njenu laganu posljedicu da je $\mathcal{D}_{F \circ G} = \mathcal{D}_G$ ako je F totalna. Iz toga slijedi $K \preceq \text{Halt}_1$, pa je Halt_1 neodlučiv po korolaru 5.15 i teoremu 5.11.

Slično, po definiciji iz korolara 3.47, vrijedi $\text{Halt}_1(x, e) \Leftrightarrow \text{HALT}(\langle x \rangle, e)$, a funkcija Code^1 je (primitivno) rekurzivna, te je $\text{Halt}_1 \preceq \text{HALT}$, iz čega je i HALT neodlučiv. \square

U literaturi se Halt_1 odnosno HALT (kako gdje) zove *halting problem* (problem zaustavljanja), jer postavlja pitanje stane li određeno izračunavanje (zadano pomoću RAM-programa i ulaza za njega). Mogli bismo dokazati i neodlučivost problema Halt_k za $k \geq 2$ (pokušajte formalno dokazati svođenje $\text{Halt}_1(x, e) \Leftrightarrow \text{Halt}_k(x, 0, 0, \dots, 0, e)$), ali to će lako slijediti jednom kad dokažemo teorem o parametru.

Dakle, ono što zasad znamo, po Church–Turingovoj tezi, je da ne postoji algoritam koji bi za svaki par RAM-stroja P i ulaza u za njega odlučivao stane li P -izračunavanje $s u$ — odnosno, za svaki algoritam \mathcal{A} postoji RAM-program P i prirodni broj u takav da \mathcal{A} ne daje ispravan odgovor na pitanje stane li P -izračunavanje $s u$. Ali znamo i više: možemo postići da P ne ovisi o \mathcal{A} . To je u skladu s univerzalnošću: postoji „najkompliciraniji” RAM-stroj, čiji je problem zaustavljanja najteži.

Korolar 5.17. *Postoji RAM-program P_0 , takav da nijedan algoritam ne može točno odrediti, za svaki $u \in \mathbb{N}$, stane li P_0 -izračunavanje $s u$.*

Dokaz. Po korolaru 5.9, Russellova funkcija je parcijalno rekurzivna. Po teoremu 2.36, ona je i RAM-izračunljiva. Označimo s P_0 jedan RAM-program koji je računa. Po definiciji 1.10, za svaki $u \in \mathbb{N}$, P_0 -izračunavanje $s u$ stane ako i samo ako je $u \in K$.

Kad bi postojao algoritam za odlučivanje problema zaustavljanja P_0 -izračunavanja s proizvoljnim u , po Church–Turingovoj tezi K bi bio odlučiv problem, što je u kontradikciji s teoremom 5.11. \square

Lema 5.18. *Postoji Turingov stroj \mathcal{T}_0 nad unarnom abecedom $\Sigma_\bullet = \{\bullet\}$, takav da nijedan algoritam ne može točno odrediti, za svaku riječ $w \in \bullet^*$, stane li \mathcal{T}_0 -izračunavanje $s w$.*

Dokaz. Po korolaru 4.50, jezična funkcija $\rho := \bullet\text{Russell}$ je Turing-izračunljiva, te vrijedi

$$\rho(\bullet^t) \simeq \bullet^{\text{Russell}(t)}. \quad (5.11)$$

To znači da za \mathcal{T}_0 možemo uzeti Turingov stroj koji računa ρ . Očito \mathcal{T}_0 jest nad unarnom abecedom, i stane točno za riječi oblika \bullet^t za $t \in K$.

Sad, kada bi postojao algoritam koji bi za svaku riječ nad Σ_\bullet odlučivao hoće li \mathcal{T}_0 -izračunavanje s njome stati, pomoću njega bismo mogli odlučiti K , sljedećim algoritmom: za ulaz $u \in \mathbb{N}$, konstruiramo riječ \bullet^u , i vratimo stane li \mathcal{T}_0 -izračunavanje s njom. Po definiciji 4.5, to će biti ako i samo ako je $\bullet^u \in \mathcal{D}_\rho$, što će prema definiciji unarne reprezentacije biti ako i samo ako je $u \in K$. Po Church–Turingovoj tezi, to bi značilo da je K rekurzivna, što je u kontradikciji s teoremom 5.11. \square

Propozicija 5.19 (Neodlučivost problema zaustavljanja za Turingove strojeve). *Za svaku abecedu Σ postoji Turingov stroj nad njom, čiji je problem zaustavljanja neodlučiv.*

Dokaz. Ako je $\bullet \in \Sigma$, možemo direktno primijeniti lemu 5.18, na Turingov stroj \mathcal{T}_0 s abecedom Σ . Kad bi postojao algoritam koji za svaku riječ $w \in \Sigma^*$ točno određuje stane li \mathcal{T}_0 -izračunavanje s w , to bi specijalno moralo vrijediti za sve $w \in \bullet^* \subseteq \Sigma^*$, što je u kontradikciji s lemom 5.18.

Ako pak $\bullet \notin \Sigma$, uzmimo neki konkretni znak $\alpha \in \Sigma$ (po definiciji abecede je $\Sigma \neq \emptyset$, pa α postoji), i zamijenimo ga s \bullet : gledamo Turingov stroj \mathcal{T}'_0 koji je kao \mathcal{T}_0 iz prethodnog odlomka, osim što umjesto \bullet ima α — kako u Σ , tako i u Γ i u svim prijelazima od δ . S obzirom na to da znakovi radne abecede nemaju nikakvu imanentnu semantiku (osim praznine, ali za to pogledajte sljedeći odlomak), \mathcal{T}'_0 -izračunavanje s α^u stane ako i samo ako \mathcal{T}_0 -izračunavanje s \bullet^u stane — pa kad bi neki algoritam za svaku $w \in \Sigma^*$ točno određivao stane li \mathcal{T}'_0 -izračunavanje s w , tada bi to specijalno vrijedilo i za sve riječi oblika α^u , $u \in \mathbb{N}$, pa bismo opet imali kontradikciju s neodlučivošću od K kao u prethodnom odlomku.

Jedan detalj još moramo uzeti u obzir: što ako je praznina stroja \mathcal{T}_0 upravo \bullet ? Tada ona po definiciji nije u Σ , ali je ne možemo jednostavno zamijeniti s α jer α jest u Σ . U tom slučaju odaberemo neki novi znak $\beta \notin \Gamma$, proglasimo ga prazninom, i \bullet zamijenimo s β u Γ i δ . Nakon toga primijenimo postupak iz prethodnog odlomka. \square

5.3. Neodlučivost u logici prvog reda

U uvodnom kursu matematičke logike obično se za logiku sudova dokažu teoremi adekvatnosti i potpunosti (koji povezuju sintaksni pojam dokazivosti sa semantičkim pojmom valjanosti), te se navedu neki sustavi (npr. glavni test) koji u potpunosti određuju je li neka formula valjana ili nije. Drugim riječima, postoji algoritam za utvrđivanje valjanosti. Postoji li *polinomni* algoritam za to, je zanimljivije pitanje i vodi na slavni milijun dolara vrijedan problem $P \stackrel{?}{=} NP$, ali barem znamo da je valjanost odlučiva. Štoviše, nije prevelik problem (i dobra je vježba), koristeći kodiranje opisano u primjeru 3.23, formalno dokazati da je skup $PValid \subset PF$, svih kodova valjanih formula logike sudova, primitivno rekurzivan. Praktički jedino na što treba misliti je da je ne možemo kodirati (totalne) interpretacije jer ih ima neprebrojivo mnogo, ali parcijalne interpretacije (s konačnom domenom) možemo.

U logici prvog reda, također vrijede teoremi adekvatnosti i potpunosti (dakle, postoji dokaz za formulu ϕ koji koristi samo aksiome i pravila zaključivanja logike prvog reda, ako i samo ako je ϕ istinita u svim σ -strukturama prvog reda, gdje je σ signatura koja sadrži sve nelogičke simbole u ϕ), i također imamo glavni test, i znamo da je korektan (odgovor koji daje je uvijek točan), ali nije totalan — mogu postojati beskonačne grane, u kojima algoritam ne daje odgovor. I to nije nedostatak specifičnog algoritma: jedan od

prvih i najvažnijih rezultata o neodlučivosti je Churchov teorem da **problem valjanosti za logiku prvog reda nije odlučiv**. Skraćeno kažemo „logika prvog reda nije odlučiva”, i doista, valjanost tu nije esencijalna. Ekvivalentno možemo promatrati probleme ispunjivosti, oborivosti, proturječnosti, ili čak zaključivanja (logičke posljedice iz konačnog skupa formula). Mi ćemo promatrati problem koji prirodno ima oblik problema zaključivanja, pa je prirodno usredotočiti se na njegovu vezu s problemom valjanosti.

Definicija 5.20. Označimo s $F1$ skup svih formula nad jezikom logike prvog reda.

Problem valjanosti pita: za proizvoljnu formulu $\phi \in F1$, je li ϕ valjana?

Problem zaključivanja pita: za proizvoljni $k \in \mathbb{N}$, za proizvoljne $\phi_1, \phi_2, \dots, \phi_k \in F1$, za proizvoljnu $\phi \in F1$, je li ϕ logička posljedica skupa $\{\phi_1, \dots, \phi_k\}$? \triangleleft

Propozicija 5.21. *Problemi valjanosti i zaključivanja svedivi su jedan na drugi.*

Dokaz. Možemo samo neformalno pričati o svođenju, jer još nismo precizirali abecedu i jezik logike prvog reda. Taj jezik jest detaljno prikazan u [17], ali nad beskonačnom abecedom, što nije dovoljno dobro za naše potrebe (pokušajte otkriti na kojim smo sve mjestima dosad koristili konačnost abecede). Uz nekoliko jednostavnih modifikacija, prilagodit ćemo taj jezik teoriji formalnih jezika, ali zasad samo opišimo ideje.

U jednom smjeru, zamislimo da imamo instancu problema valjanosti, i algoritam („crnu kutiju”) za odluku problema zaključivanja. Pitamo se što uvrstiti u taj algoritam, da dobijemo odgovor na pitanje valjanosti. Naravno, ako zadana instanca pita je li ϕ valjana, odgovor na to pitanje je isti kao i na pitanje je li ϕ posljedica praznog skupa formula — dakle stavimo $k = 0$ (i ne moramo zadavati formule ϕ_i jer ih nema).

U drugom smjeru je malo zanimljivije. Pretpostavimo da imamo algoritam za utvrđivanje valjanosti, i da su nam zadane formule $\phi_1, \phi_2, \dots, \phi_k$ i ϕ , te se pitamo je li ϕ logička posljedica ovih prethodnih. Svođenje možemo opisati indukcijom po $k \in \mathbb{N}$. Baza je ista kao gore: za $k = 0$, zapravo se pitamo je li ϕ valjana. Pretpostavimo da je problem zaključivanja za $k = l$ svediv na problem valjanosti, i pogledajmo problem zaključivanja za $k = l + 1$. Također, zbog jakog teorema potpunosti svejedno je koristimo li relaciju logičke posljedice \models ili izvedivosti \vdash .

Ako je zadnja premisa ϕ_{l+1} *rečenica* (zatvorena formula, bez slobodnih varijabli), možemo primijeniti teorem dedukcije (za jedan smjer — drugi smjer je trivijalan pomoću pravila zaključivanja *modus ponens*):

$$\phi_1, \dots, \phi_l, \phi_{l+1} \vdash \phi \quad \text{ako i samo ako} \quad \phi_1, \dots, \phi_l \vdash (\phi_{l+1} \rightarrow \phi), \quad (5.12)$$

zbog kojeg je problem zaključivanja s $l + 1$ premisom svediv na problem zaključivanja s l premisa, a onda po pretpostavci indukcije i po tranzitivnosti na problem valjanosti.

Ako formula ϕ_{l+1} nije rečenica, umjesto nje možemo promotriti njeno *univerzalno zatvorenje* $\overline{\phi_{l+1}}$, dobiveno univerzalnim kvantificiranjem svih slobodnih varijabli u ϕ_{l+1} . Lako se vidi da $\mathfrak{N} \models \phi_{l+1}$ ako i samo ako $\mathfrak{N} \models \overline{\phi_{l+1}}$, dakle odgovor na problem

zaključivanja neće se promijeniti, a $\overline{\phi_{l+1}}$ jest rečenica, pa možemo primijeniti teorem dedukcije kao u prethodnom odlomku. \square

Za dovršetak odnosno formalizaciju dokaza još bi trebalo vidjeti da je funkcija koja preslikava $(\phi_1, \dots, \phi_l, \phi_{l+1}, \phi)$ u $(\phi_1, \dots, \phi_l, (\overline{\phi_{l+1}} \rightarrow \phi))$ rekursivna (zadana s $l+1$ koordinatnih funkcija na kodovima, prvih l od kojih su inicijalne), što zasad ne možemo jer nemamo kodiranje F1 — ali je intuitivno jasno da to možemo učiniti samo mehaničkim manipulacijama simbolima od kojih se formule sastoje.

5.3.1. Reprezentacija RAM-konfiguracija formulama prvog reda

Osnovna ideja dokaza Churchovog teorema je svesti problem zaustavljanja za RAM-strojeve na problem zaključivanja. Iz toga će onda Churchov teorem odmah slijediti po propoziciji 5.16 i korolaru 5.15.

Dakle, neka su $k \in \mathbb{N}_+$, $P \in \mathcal{P}\text{rog}$, te $\vec{u} \in \mathbb{N}^k$ (u logici prvog reda obično individualne varijable označavamo s x_i , pa ćemo ovdje koristiti u_i kao uobičajenu oznaku za ulazne podatke). Trebamo konstruirati skup formula Γ i formulu ζ , koji ovise o P i \vec{u} , tako da P -izračunavanje s \vec{u} stane ako i samo ako vrijedi $\Gamma \models \zeta$. Štoviše, zbog korolara 5.17, zapravo možemo cijelu konstrukciju provesti za fiksni P (i za $k = 1$), pa ga nećemo pisati (iako smo svjesni da će Γ i ζ ovisiti o P).

Logičko zaključivanje je, u osnovi, traženje „puta” od premisa do zaključka, i osnovni razlog zašto je to težak problem je što znamo da put (*izvod*) postoji kad ga nađemo, ali dok ga nismo našli, ne znamo je li to zato što ne postoji, ili samo nismo tražili dovoljno dugo. To vrlo podsjeća na traženje „puta” od početne do završne konfiguracije pri rješavanju problema zaustavljanja — možemo li nekako formalizirati tu vezu?

Očito, u tom smislu, u Γ bi se trebala nalaziti neka formula $\pi_{\vec{u}}$ koja opisuje početnu konfiguraciju s ulazom \vec{u} , a ζ bi trebala biti formula koja opisuje neku (bilo koju) završnu konfiguraciju — kako nas ne zanima rezultat izračunavanja, nego samo zaustavljanje, ζ očito može biti egzistencijalno kvantificirana po stanju registara, i samo fiksirati vrijednost programskog brojača, a onda ne mora ovisiti o \vec{u} .

U tom smislu, instrukcije programa P mogli bismo shvatiti kao pravila zaključivanja — recimo, 3. INC \mathcal{R}_1 kaže da iz formule koja predstavlja konfiguraciju $(x, y, \bar{z}, 0, \dots, 3)$ možemo izvesti formulu koja predstavlja konfiguraciju $(x, y + 1, \bar{z}, 0, \dots, 4)$, za sve x , y i \bar{z} — ali s time postoji jedan bitan problem: logika prvog reda ne dopušta nam imati vlastita (nelogička) pravila zaključivanja, možemo imati samo *modus ponens* i generalizaciju. Srećom, *modus ponens* je vrlo općenito pravilo, koje može simulirati ostala pravila putem odgovarajućih nelogičkih aksioma — primjerice, ako bismo htjeli imati pravilo kojim iz A i B izvodimo C , dovoljno je u nelogičke aksiome dodati formulu $(A \rightarrow (B \rightarrow C))$. Tada možemo iz tog aksioma i A izvesti $(B \rightarrow C)$, pa onda iz toga i B izvesti C , koristeći samo *modus ponens*.

Taj pristup ćemo koristiti ovdje, odnosno u Γ ćemo imati po jednu formulu ι_i za svaku instrukciju programa P (s rednim brojem i). Još je preostalo objasniti što ćemo s ovim trotočkama u konfiguracijama (odnosno kako reprezentirati konačni nosač), i što s aritmetičkim operacijama (+ za INC, – za DEC).

Konfiguracije prikazujemo određenim atomarnim formulama. Sadržaj programskog brojača, budući da je jedan od konačno mnogo njih (za fiksni RAM-program P), kodirat ćemo statički, kroz supskript relacijskog simbola odgovarajuće atomarne formule — a sadržaj pojedinih registara preko terma u toj formuli. Ipak, kako svaka (atomarna) formula može imati samo konačno mnogo terma, morat ćemo se ograničiti samo na relevantne registre.

Aritmetiku bismo mogli prikazati nekom varijantom Peanove aritmetike, ali zapravo možemo i lakše: budući da je sve što nam treba sljedbenik i prethodnik, dovoljno je imati jedan konstantski simbol (koji predstavlja nulu) i jedan jednomjesni funkcijski simbol (koji predstavlja funkciju sljedbenik). Zatvoreni termi onda na očit način predstavljaju prirodne brojeve: broj u je predstavljen termom koji predstavlja u -struku primjenu tog funkcijskog simbola na taj konstantski simbol.

Napomenimo samo da ne promatramo logiku prvog reda s jednakošću — nećemo uopće imati potrebu promatrati jednakost, a kamoli kao relacijski simbol s nekim posebnim svojstvima. Usporedba s nulom za instrukciju DEC bit će sintaksna.

Vrijeme je da počnemo pričati formalnije: krenimo od RAM-programa P , i mjesnosti k (drugim riječima, od RAM-algoritma P^k). Označimo s $n := n_P$ duljinu programa P , i s $m := m_{P^k}$ širinu algoritma P^k . Signatura σ , nad kojom ćemo graditi formule za naš problem zaključivanja, imat će:

- jedan konstantski simbol 0;
- jedan jednomjesni funkcijski simbol s ;
- $n + 1$ m -mjesnih relacijskih simbola $R_0^m, R_1^m, \dots, R_n^m$.

Za svaki $u \in \mathbb{N}$, zatvoreni term $s(s(\dots(s(0))\dots))$ u kojem ima u pojava funkcijskog simbola s , skraćeno označavamo s \bar{u} — dakle $\bar{0} = 0$, $\bar{1} = s(0)$, $\bar{2} = s(s(0))$ itd. Za proizvoljni $\vec{u} = (u_1, u_2, \dots, u_k) \in \mathbb{N}^k$, definiramo formulu

$$\pi_{\vec{u}} := R_0(0, \bar{u}_1, \bar{u}_2, \dots, \bar{u}_k, 0, 0, \dots, 0), \quad (5.13)$$

gdje nakon \bar{u}_k ima još $m - k - 1$ simbola 0 (po definiciji je $m = m_{P^k} = \max\{m_P, k + 1\} \geq k + 1$, pa je $m - k - 1 = m - (k + 1) \in \mathbb{N}$).

Za svaki $i \in [0..n]$, definiramo formulu ι_i ovisno o tipu instrukcije s rednim brojem i u programu P :

▷ Ako je to i . INC \mathcal{R}_j , definiramo

$$\iota_i := (R_i(x_0, \dots, x_{m-1}) \rightarrow R_{i+1}(x_0, \dots, x_{j-1}, s(x_j), x_{j+1}, \dots, x_{m-1})). \quad (5.14)$$

▷ Ako je to i. DEC \mathcal{R}_j, l , definiramo

$$\iota_i := ((R_i(x_0, \dots, x_{j-1}, 0, x_{j+1}, \dots, x_{m-1}) \rightarrow R_l(x_0, \dots, x_{j-1}, 0, x_{j+1}, \dots, x_{m-1})) \wedge \\ \wedge (R_i(x_0, \dots, x_{j-1}, s(x_j), x_{j+1}, \dots, x_{m-1}) \rightarrow R_{i+1}(x_0, \dots, x_{m-1}))). \quad (5.15)$$

▷ Ako je to i. GO TO l , definiramo

$$\iota_i := (R_i(x_0, \dots, x_{m-1}) \rightarrow R_l(x_0, \dots, x_{m-1})). \quad (5.16)$$

Definiramo skup

$$\Gamma_{\vec{u}} := \{\pi_{\vec{u}}\} \cup \{\iota_i \mid i \in [0..n]\} \quad (5.17)$$

i formulu

$$\zeta := \exists x_0 \cdots \exists x_{m-1} R_n(x_0, \dots, x_{m-1}). \quad (5.18)$$

5.3.2. Zaključivanje kao zaustavljanje

Neka je P^k RAM-algoritam, \vec{u} ulaz za njega, i pomoću njih konstruirajmo $\Gamma_{\vec{u}}$ i ζ kako je opisano u prethodnoj točki. Također, neka je $(c_t)_{t \in \mathbb{N}}$ P -izračunavanje s \vec{u} . Promotrimo σ -strukturu $\mathfrak{N} := (\mathbb{N}, \phi)$, čija je interpretacija zadana s

- $\phi(0) := 0$,
- $\phi(s) := Sc$, te
- za svaki $i \in [0..n]$, $\phi(R_i) := \{(c_t(\mathcal{R}_0), \dots, c_t(\mathcal{R}_{m-1})) \mid t \in \mathbb{N} \wedge c_t(PC) = i\}$.

Indukcijom po r lako dokažemo $\phi(\bar{r}) = r$ za sve $r \in \mathbb{N}$, pa $\mathfrak{N} \models R_{pc}(\bar{r}_0, \bar{r}_1, \dots, \bar{r}_{m-1})$ — odnosno $\mathfrak{N} \models_v R_{pc}(x_0, x_1, \dots, x_{m-1})$, gdje je v valuacija takva da je $v(x_j) = r_j$ za sve $j \in [0..m]$ — neformalno znači da RAM-stroj s programom P i ulazom \vec{u} može doći u konfiguraciju $(r_0, r_1, \dots, r_{m-1}, 0, 0, \dots, pc)$ (nakon t koraka izračunavanja).

Lema 5.22. *Vrijedi $\mathfrak{N} \models \pi_{\vec{u}}$.*

Dokaz. Gledajući (5.13), samo treba vidjeti da postoji $t \in \mathbb{N}$ takav da je $c_t(PC) = 0$ i $c_t(\mathcal{R}_j) = \begin{cases} u_j, & j \in [1..k] \\ 0, & \text{inače} \end{cases}$; drugim riječima, c_t je početna konfiguracija s ulazom \vec{u} , a ona se sigurno postiže za $t = 0$ (možda i za neke druge t , ali jedan nam je dovoljan). \square

Lema 5.23. *Za svaki $i \in [0..n]$ vrijedi $\mathfrak{N} \models \iota_i$.*

Dokaz. Očekivano, imat ćemo slučajeve ovisno o tipu i -te instrukcije programa P .

Ako je to INC, dakle i. INC \mathcal{R}_j za neki j , tada mora biti $j < m_p \leq m$, i trebamo vidjeti da u \mathfrak{N} vrijedi formula (5.14). Ta formula je kondicional, pa uzmimo proizvoljnu

valuaciju v i pretpostavimo da u \mathfrak{M} uz valuaciju v vrijedi njen antecedens: $\mathfrak{M} \models_v R_i(x_0, \dots, x_{m-1})$. To znači da postoji t takav da je $c_t = (v(x_0), \dots, v(x_{m-1}), 0, \dots, i)$, no takva konfiguracija po definiciji 1.8(1) prelazi u

$$(v(x_0), \dots, v(x_{j-1}), v(x_j) + 1, v(x_{j+1}), \dots, v(x_{m-1}), 0, \dots, i + 1). \quad (5.19)$$

S druge strane, po definiciji 1.10, $c_t \rightsquigarrow c_{t+1}$, pa po lemi 1.9 imamo da je (5.19) upravo c_{t+1} (to ćemo koristiti i kasnije). Drugim riječima, stroj može postići konfiguraciju (5.19), iz čega slijedi (primijetimo da je $v(x_j) + 1 = \text{Sc}(v(x_j)) = \phi(s)(v(x_j)) = v(s(x_j))$)

$$\mathfrak{M} \models_v R_{i+1}(x_0, \dots, x_{j-1}, s(x_j), x_{j+1}, \dots, x_{m-1}), \quad (5.20)$$

odnosno u \mathfrak{M} uz valuaciju v vrijedi i konsekvens formule (5.14), što smo trebali.

Ako je instrukcija i. DEC \mathcal{R}_j, l za neke $j < m$ i $l \leq n$, tada trebamo dokazati da u \mathfrak{M} (uz proizvoljnu valuaciju v) vrijedi formula (5.15). Ta formula je konjunkcija dva kondicionala, pa trebamo dokazati da vrijede oba, metodom sličnom kao u prethodnom odlomku. Za prvi, pretpostavimo da vrijedi $\mathfrak{M} \models_v R_i(x_0, \dots, x_{j-1}, 0, x_{j+1}, \dots, x_{m-1})$. To znači da postoji t takav da je $c_t = (v(x_0), \dots, v(x_{j-1}), 0, v(x_{j+1}), \dots, v(x_{m-1}), 0, \dots, i)$ (jer je $v(0) = \phi(0) = 0$), no takva konfiguracija po definiciji 1.8(3) prelazi u

$$c_{t+1} = (v(x_0), \dots, v(x_{j-1}), 0, v(x_{j+1}), \dots, v(x_{m-1}), 0, \dots, l), \quad (5.21)$$

koja je time dostižna (u $t+1$ koraka), pa vrijedi $\mathfrak{M} \models_v R_l(x_0, \dots, x_{j-1}, 0, x_{j+1}, \dots, x_{m-1})$.

Za drugi, pretpostavimo $\mathfrak{M} \models_v R_i(x_0, \dots, x_{j-1}, s(x_j), x_{j+1}, \dots, x_{m-1})$. Kao u INC-slučaju, $v(s(x_j)) = v(x_j) + 1$, što je pozitivno zbog $v(x_j) \in |\mathfrak{M}| = \mathbb{N}$, pa to znači da postoji t takav da je $c_t = (v(x_0), \dots, v(x_{j-1}), v(x_j) + 1, v(x_{j+1}), \dots, v(x_{m-1}), 0, \dots, i)$. Po definiciji 1.8(2), ta konfiguracija prelazi u $c_{t+1} = (v(x_0), \dots, v(x_{m-1}), 0, \dots, i + 1)$, te vrijedi $\mathfrak{M} \models_v R_{i+1}(x_0, \dots, x_{m-1})$, čime smo dokazali i drugi kondicional.

Ako je instrukcija i. GO TO l za neki $l \leq n$, tada opet uzmemo proizvoljnu valuaciju v , i trebamo dokazati da uz nju u \mathfrak{M} vrijedi formula (5.16). To direktno slijedi iz definicije 1.8(4), jer se stanje registara u ovom slučaju uopće ne mijenja, samo se vrijednost programskog brojača promijeni iz i u l . \square

Propozicija 5.24. *Za svaki $\vec{u} \in \mathbb{N}^k$, ako $\Gamma_{\vec{u}} \models \zeta$, tada P-izračunavanje s \vec{u} stane.*

Dokaz. Pretpostavka $\Gamma_{\vec{u}} \models \zeta$ znači da u svakoj strukturi u kojoj vrijede sve formule iz $\Gamma_{\vec{u}}$, vrijedi i formula ζ . Leme 5.22 i 5.23 pokazuju da je \mathfrak{M} takva struktura, pa vrijedi $\mathfrak{M} \models \zeta$. Čitajući (5.18), vidimo da to znači da za svaku valuaciju v (pa specijalno, recimo, za onu koja sve varijable preslika u 0) postoji valuacija v' , koja se podudara s v na svim individualnim varijablama $x_i, i \geq m$, takva da vrijedi

$$(v'(x_0), \dots, v'(x_{m-1})) \in \phi(R_n). \quad (5.22)$$

Valuacije nam zapravo ovdje uopće nisu bitne — jedino što je bitno je da iz toga slijedi $\phi(R_n) \neq \emptyset$, pa postoji $t \in \mathbb{N}$ takav da je $c_t(PC) = n$. No to znači da je c_t završna konfiguracija, pa P-izračunavanje s \vec{u} stane nakon t koraka. \square

Sada nam je cilj dokazati obrat propozicije 5.24. Bitna razlika je što sad moramo *dokazati* da je ζ logička posljedica od $\Gamma_{\vec{u}}$, pa više ne možemo koristiti „intendiranu interpretaciju” s prirodnim brojevima, nego moramo provesti argumentaciju za proizvoljnu σ -strukturu \mathfrak{M} u kojoj vrijedi $\Gamma_{\vec{u}}$. Ta struktura ne mora biti izomorfna s \mathfrak{N} , ne mora čak ni zadovoljavati jednostavne aksiome poput injektivnosti sljedbenika (iskreno, ne možemo ih ni *iskazati* jer nemamo jednakost u teoriji), ali svejedno ćemo moći pokazati da u njoj vrijedi ζ .

Pa neka je $\mathfrak{M} = (M, \phi) \cong (M, \Theta, g, S_0, S_1, \dots, S_n)$ proizvoljna σ -struktura (redom imamo $\phi(0) =: \Theta \in M$, $\phi(s) =: g: M \rightarrow M$, a za svaki $i \in [0..n]$, $\phi(R_i) =: S_i \subseteq M^m$), i neka vrijedi $\mathfrak{M} \models \Gamma_{\vec{u}}$.

Neka je $(c_t)_{t \in \mathbb{N}}$ P-izračunavanje s \vec{u} . Za proizvoljne $t \in \mathbb{N}$ i $j \in [0..m]$, označimo $a_{jt} := \phi(\overline{c_t(R_j)}) = g(g(\dots g(\Theta) \dots))$, gdje ima $c_t(R_j)$ poziva funkcije g .

Lema 5.25. *Za svaki $t \in \mathbb{N}$, vrijedi $\mathfrak{M} \models R_{c_t(PC)}(\overline{c_t(R_0)}, \dots, \overline{c_t(R_{m-1})})$, odnosno $(a_{0t}, a_{1t}, \dots, a_{(m-1)t}) \in S_{c_t(PC)}$.*

Dokaz. Indukcijom po t . Za $t = 0$, tražena formula je upravo $\pi_{\vec{u}}$, koja je u $\Gamma_{\vec{u}}$ pa po pretpostavci vrijedi u \mathfrak{M} . Pretpostavimo da tvrdnja vrijedi za $t = b$, i pogledajmo tvrdnju za $t = b + 1$. Ako je c_b završna konfiguracija, tada je $c_{b+1} = c_b$ jer završne konfiguracije prelaze (jedino) u same sebe, pa je formula za c_{b+1} ista kao formula za c_b , i vrijedi u \mathfrak{M} po pretpostavci indukcije. Inače, $i := c_b(PC) < n$, pa postoji instrukcija programa P s rednim brojem i , i također je $\iota_i \in \Gamma_{\vec{u}}$, dakle $\mathfrak{M} \models \iota_i$.

Ako je ta instrukcija tipa INC, recimo i . INC R_j za neki $j < m$, tada u \mathfrak{M} vrijedi (5.14), odnosno uz valuaciju $v(x_j) := a_{jb}$, vrijedi

$$(a_{0b}, \dots, a_{(j-1)b}, g(a_{jb}), a_{(j+1)b}, \dots, a_{(m-1)b}) \in S_{i+1}, \quad (5.23)$$

uz pretpostavku $(a_{0b}, \dots, a_{(m-1)b}) \in S_i = S_{c_b(PC)}$. No ta pretpostavka je upravo pretpostavka indukcije, pa onda vrijedi i (5.23). A formula (5.23) je upravo formula koja odgovara konfiguraciji c_{b+1} , jer je $i + 1 = c_b(PC) + 1 = c_{b+1}(PC)$, i

$$\begin{aligned} a_{j(b+1)} &= \phi(\overline{c_{b+1}(R_j)}) = \phi(\overline{1 + c_b(R_j)}) = \phi(s(\overline{c_b(R_j)})) = \\ &= \phi(s)(\phi(\overline{c_b(R_j)})) = g(a_{jb}). \end{aligned} \quad (5.24)$$

Ako je ta instrukcija tipa GO TO, s odredištem $l \leq n$, uz istu valuaciju imamo da $\vec{d} \in S_i$ povlači $\vec{d} \in S_l$. Ova prva tvrdnja je pretpostavka indukcije, a druga je upravo ona koja nam treba za korak (stanje registara ostaje isto, odnosno $a_{jb} = a_{j(b+1)}$ za svaki $j \in [0..m]$).

Ako je ta instrukcija tipa DEC, recimo DEC \mathcal{R}_j, l , promotrimo slučajeve s obzirom na to je li $c_b(\mathcal{R}_j) = 0$ (krivo je reći „s obzirom na to je li $a_{jb} = \Theta$ ”, jer nitko ne brani da bude npr. $g(\Theta) = \Theta$, odnosno $\phi(\bar{1}) = \phi(\bar{0})$). Ako jest, imamo istu argumentaciju kao u prethodnom odlomku, jer se stanje registara tada ne mijenja: $\vec{a} \in S_i$ povlači $\vec{a} \in S_l$ po prvom konjunkt u (5.15). Ako nije, tada je sigurno $c_b(\mathcal{R}_j) \geq 1$, pa promotrimo (5.15)

(odnosno njen drugi konjunkt) uz valuaciju $v'(x_h) := \begin{cases} a_{hb} = \phi(\overline{c_b(\mathcal{R}_h)}), & h \neq j \\ \phi(\overline{c_b(\mathcal{R}_j) - 1}), & h = j \end{cases}$. Uz

tu valuaciju vrijedi antecedens, jer je $v'(s(x_j)) = a_{jb}$ — pa onda vrijedi i konsekvens, odnosno formula s istim sadržajem registara, osim što je $c_{b+1}(\mathcal{R}_j)$ za jedan manji, i $c_{b+1}(\text{PC})$ za jedan veći, kao što i treba biti. \square

Propozicija 5.26. *Ako P-izračunavanje s \vec{u} stane, tada $\Gamma_{\vec{u}} \models \zeta$.*

Dokaz. Pretpostavimo da izračunavanje $(c_t)_t$ stane, odnosno postoji t_0 takav da je $c_{t_0}(\text{PC}) = n$. Da bismo dokazali $\Gamma_{\vec{u}} \models \zeta$, uzmimo proizvoljnu σ -strukturu \mathfrak{M} u kojoj vrijedi $\Gamma_{\vec{u}}$. Za tu strukturu vrijedi lema 5.25, pa specijalno za $t = t_0$ vrijedi $(a_{0t_0}, \dots, a_{(m-1)t_0}) \in S_n$. To znači da za svaku valuaciju v možemo naći v' koja se podudara na v u svim varijablama x_j za $j \geq m$, te je $v'(x_j) := a_{jt_0}$ za $j < m$, takvu da vrijedi $\mathfrak{M} \models_{v'} R_n(x_0, \dots, x_{m-1})$. No to upravo znači $\mathfrak{M} \models \zeta$. \square

5.3.3. Formule prvog reda kao formalni jezik

Teorem 5.27 (Church). *Ne postoji algoritam koji za proizvoljnu formulu logike prvog reda odlučuje je li valjana.*

Dokaz. Pretpostavimo da imamo takav algoritam \mathcal{A} , i opišimo sljedeći neformalni algoritam za odluku Russellovog skupa:

Kao u dokazu korolara 5.17 (kompajliranjem), dobijemo RAM-program P_0 koji računa Russellovu funkciju. Za taj program uvedimo oznake m i n , te formule $\iota_i, i < n$ i ζ nad signaturom σ (koja također ovisi o P_0 preko m).

Za ulaz $u \in \mathbb{N}$ (koji dalje promatramo kao $(u) \in \mathbb{N}^1$):

1. Konstruiramo formulu π_u .
2. Konstruiramo konačan skup Γ_u .
3. Konstruiramo instancu problema zaključivanja $\Gamma_u \models^? \zeta$.
4. Pretvorimo je (koristeći dokaz propozicije 5.21) u instancu problema valjanosti neke formule φ_u .
5. Provjerimo (koristeći \mathcal{A}) je li tako dobivena formula valjana.

Taj algoritam odlučuje Russellov skup, jer za svaki $u \in \mathbb{N}$ vrijedi:

$$u \in K \iff P_0\text{-izračunavanje s } u \text{ stane} \iff (\Gamma_u \models \zeta) \iff \models \varphi_u. \quad (5.25)$$

Prva ekvivalencija je definicija 1.10, jer je $K = \mathcal{D}_{\text{Russell}}$, a P_0 računa Russell. Druga ekvivalencija je u jednom smjeru propozicija 5.26, a u drugom smjeru propozicija 5.24. Treća ekvivalencija je dobivena primjenom propozicije 5.21.

Dakle, pod pretpostavkom da \mathcal{A} ispravno odlučuje koje su formule prvog reda valjane a koje nisu, upravo opisani neformalni algoritam ispravno odlučuje koji su brojevi elementi Russellovog skupa a koji nisu. Po Church–Turingovoj tezi, to bi značilo da je relacija K rekurzivna, što je u kontradikciji s teoremom 5.11. \square

Time je dokazan Churchov teorem, ali na neformalnoj razini — „dokazali” smo da ne postoji neformalni algoritam, tako što smo se pozvali na Church–Turingovu tezu. Možemo li bez toga?

Rekli smo da probleme obično shvaćamo kao jezike: odlučivost problema je tu rekurzivnost (karakteristične funkcije kodiranja) jezika čiji elementi su riječi koje predstavljaju instance problema na koje je odgovor potvrđan. U ovom slučaju, promotrili bismo jezik *Valid* nad nekom abecedom koja je dovoljno bogata da može izraziti svaku formulu logike prvog reda, i u njemu bi se nalazile samo valjane formule. Ako uspijemo dokazati da *Valid* nije rekurzivan, „odgurati” ćemo primjenu Church–Turingove teze na sam kraj dokaza, i pseudokod u dokazu teorema 5.27 moći ćemo zamijeniti pravom rekurzivnom funkcijom koja svodi *Valid* (preciznije $\langle \text{Valid} \rangle$) na K .

Hoćemo li time dobiti išta vrednije nego u prethodnoj točki? Da, jer imamo garanciju da smo bar svođenje napisali egzaktno — a ako prihvatimo definiciju 5.6, argument postaje sasvim matematički bez pozivanja na intuiciju pojma algoritma.

Za početak, moramo uvesti tu dovoljno bogatu abecedu (označimo je Σ_{\log}). Rekli smo da koristimo nešto vrlo blisko standardnom pristupu, s najvećom razlikom što je naša abeceda konačna. (To je prilično bitan uvjet: bez njega, na primjer, funkcija prijelaza ne bi bila konačna, te bi dokaz leme 4.20 bio puno kompliciraniji.) Recimo, standardno, alfabet logike prvog reda ima beskonačno mnogo individualnih varijabli x_0, x_1, x_2, \dots — ali kad vidimo kako ih doista pišemo, posebno kasnije ($x_9, x_{10}, x_{11}, \dots, x_{283}, \dots$), vidimo da su sve one zapravo elementi jezika nad abecedom $\{x, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ s 11 elemenata. Drugim riječima, tradicionalni alfabet našeg jezika je beskonačan jer zapravo nije atomaran — sâm se može shvatiti kao (regularni) jezik nad elementarnijom abecedom, koja jest konačna.

To nije ništa neuobičajeno: moderni programski jezici imaju nekoliko međuovisnih struktura, koje im omogućuju da njihova teorija ostane utemeljena u teoriji formalnih jezika, a da ipak budu dovoljno izražajni. Recimo, po analogiji s prethodnim odlomkom, najčešće imaju prebrojivo mnogo varijabli (imena za podatke), no sasvim je jasno da su sve varijable sastavljene od konačno mnogo mogućih znakova, koristeći određena

jednostavna pravila. Konkretno, u programskom jeziku C, imena varijabli čine jezik nad abecedom $\{_, a, b, c, \dots, z, A, B, C, \dots, Z, 0, 1, 2, \dots, 9\}$ koja se sastoji od 63 znaka, i taj je jezik zadan pravilom da prvi znak ne smije biti dekadski znamenka, te čitava varijabla ne smije biti nijedna od konačno mnogo ključnih riječi.

Ta *leksička* struktura prirodno se nalazi „ispod” sintaksne strukture programskog jezika, u smislu da se leksički analizator izvršava prije sintaksnog, i predaje mu samo tipove *tokena* koji su pronađeni u izvornom kodu. Recimo, iako su *abc* i *xy* dvije različite varijable, u sintaksno ispravnom programu možemo zamijeniti jednu drugom i sigurno nećemo uvesti sintaksnu grešku — iako ćemo možda, zapravo vjerojatno, uvesti neke semantičke greške poput nedeklariranih varijabli.

Vrlo je slična stvar i u logici prvog reda: x_2 i x_{58} su različite individualne varijable, ali ako sintaksa kaže da na nekom mjestu (recimo neposredno iza kvantifikatora) mora doći varijabla, tada može doći bilo koja varijabla, i formula će ostati sintaksno ispravna. Ili, na početku atomarne formule mora stajati relacijski simbol, ali možemo staviti bilo koji relacijski simbol R_i — doduše, mora biti odgovarajuće mjesnosti, ali to samo znači da se mjesnost može zaključiti iz ostatka atomarne formule, pa je ne treba pisati. Recimo, $(R(x) \rightarrow R(x, x))$ je formula u kojoj postoje dva relacijska simbola: R_0^1 i R_0^2 .

Mogli bismo dakle doslovno tako raditi, osim što je dekadski zapis supskripta nespretno. Već smo mnogo puta jezično prikazivali prirodne brojeve u unarnom zapisu, zašto ne i ovdje? Štoviše, tradicionalna matematika već poznaje taj zapis u obliku (recimo za varijable) x, x', x'', x''', \dots — što nije ništa drugo nego prebrojiv skup individualnih varijabli koji smo već promatrali, samo nad jednostavnijom abecedom $\{x, '\}$. Isti pristup upotrijebit ćemo za konstantske (c, c', c'', \dots), funkcijske (f, f', \dots) i relacijske (R, R', \dots) simbole.

Ostatak alfabeta logike prvog reda je konačan, i možemo ga jednostavno ubaciti u našu abecedu, ali radi jednostavnosti uzet ćemo minimalan skup veznika i kvantifikatora $\{\neg, \rightarrow, \forall\}$ (tzv. *osnovni jezik* — pogledajte [17] za obrazloženje). Još moramo imati zarez i zagrade za konstrukciju složenih terma i atomarnih formula, a zagrade ćemo (vanjske) koristiti i pri pisanju kondicionala — negacija i univerzalna kvantifikacija su prefiksni operatori, i smatramo ih operatorima višeg prioriteta od kondicionala, tako da njima ne trebamo pisati zagrade.

U abecedu ćemo dodati i separator $\#$, koji ćemo koristiti u radu s konačnim nizovima formula — najvažniji primjer bit će *dokazi*, odnosno izvodi iz nekog konačnog (ili bar odlučivog) skupa aksioma.

5.3.4. Kodiranje formula prvog reda

Definicija 5.28. *Logička abeceda* je zadana prvim retkom tablice

$$\begin{array}{c|cccccccccccccc} \Sigma_{\log} & x & c & f & R & , & ' & (&) & \neg & \rightarrow & \forall & \# \\ \hline \mathbb{N}\Sigma_{\log} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{array}. \quad (5.26)$$

Njeno kodiranje zadano je drugim retkom te tablice. Njena pomaknuta baza je 12. \triangleleft

Primjer 5.29. Pogledajmo formulu $\varphi := \forall x \exists y (x < y)$. Kao što znamo, ona je ljepši zapis za nešto poput $\forall x_0 \exists x_1 R_1^2(x_0, x_1)$ (ako postoji jednakost — a ovdje vjerojatno postoji ako teorija priča o uređaju $<$ — obično se simbol R_0^2 rezervira za nju), a ona se može u osnovnom jeziku napisati kao $\forall x \neg \forall x' \neg R'(x, x')$, te je njen kod $\langle \varphi \rangle = (B19B16946715168)_{12} = 14318611220424608$. \triangleleft

Primjer 5.30. Kodirajmo formulu $1 + 1 = 2$. Prikladna teorija je recimo Peanova aritmetika, čija je formalna reprezentacija u alfabetu teorije prvog reda dana tablicom

naziv	neformalno	simbol	nad Σ_{\log}
nula	0	c_0	c
sljedbenik	x'	f_0^1	$f(x)$
zbrajanje	$x + y$	f_0^2	$f(x, x')$
množenje	xy	f_1^2	$f'(x, x')$
jednakost	$x = y$	R_0^2	$R(x, x')$
uređaj	$x < y$	R_1^2	$R'(x, x')$

(5.27)

Time formula u osnovnom jeziku postaje $R(f(f(c), f(c)), f(f(c)))$, a njen kod je $\langle 1 + 1 = 2 \rangle = (47373728537288537372888)_{12} = 2544115135095560147164520$. \triangleleft

Definicija 5.31. Nad logičkom abecedom promotrimo sljedeće jezike:

F1 Skup svih formula prvog reda.

Valid Skup svih valjanih formula prvog reda. \triangleleft

Propozicija 5.32. Jezik F1 je odlučiv.

Skica dokaza. Puni dokaz ovoga odveo bi nas predaleko u teoriju formalnih jezika. Tamo se promatraju razne klase jezika (koje čine tzv. *Chomskyjevu hijerarhiju*), te gramatike i automati za njih. Turingovi odlučitelji, koje smo spomenuli u točki 4.4.1, vrsta su takvih automata, no postoje i bitno jednostavniji automati, koji prihvataju jednostavnije jezike.

Specijalno, u sintaksoj analizi vrlo je bitna klasa *beskontekstnih* jezika, koje generiraju beskontekstne gramatike, a prepoznaju ih potisni automati. Potisni automat je specijalni slučaj (nedeterminističnog višetračnog) Turingovog stroja, a beskontekstna gramatika se može zapisati u *Chomskyjevoj normalnoj formi*, osiguravajući da njegovo izračunavanje uvijek stane — pa se može smatrati Turingovim odlučiteljem.

Jedna beskontekstna gramatika za jezik F1 zadana je pravilima

$$\text{Form} \rightarrow \text{Rel}(\text{Terms}) \mid \neg \text{Form} \mid (\text{Form} \rightarrow \text{Form}) \mid \forall \text{Var Form} \quad (5.28)$$

$$\text{Terms} \rightarrow \text{Term} \mid \text{Terms}, \text{Term} \quad (5.29)$$

$$\text{Term} \rightarrow \text{Var} \mid \text{Const} \mid \text{Func}(\text{Terms}) \quad (5.30)$$

uz tipove tokena

$$\text{Var} \rightarrow x \mid \text{Var}' \quad (5.31)$$

$$\text{Const} \rightarrow c \mid \text{Const}' \quad (5.32)$$

$$\text{Func} \rightarrow f \mid \text{Func}' \quad (5.33)$$

$$\text{Rel} \rightarrow R \mid \text{Rel}' \quad (5.34)$$

(Var, Const, Func i Rel su leksičke varijable, dok su Form, Term i Terms sintaksne varijable; Form je početna varijabla). Objašnjenje pojmova i dokaze tvrdnji koje smo naveli zainteresirani čitatelj može pronaći u [12]. \square

Iz prethodne propozicije slijedi (po teoremu 4.51) da je $\langle F1 \rangle$ rekurzivan skup. Drugim riječima, ako restringiramo $\mathbb{N}\Sigma_{\log}^*$ na formule prvog reda zapisane u osnovnom jeziku, imamo kodiranje formula. Za konstruktore tog skupa moramo prvo napraviti leksičke konstruktore za pojedine tokene, ali to doista nije teško. U nastavku često koristimo konkatenaciju u pomaknutoj bazi 12, koju zovemo jednostavno *konkatenacijom*, te umjesto $\hat{\cdot}_{12}$ pišemo samo \frown .

Lema 5.33. *Postoje primitivno rekurzivne funkcije Var, Const, Func i Rel koje preslikavaju $i \in \mathbb{N}$ redom u $\langle x_i \rangle$, $\langle c_i \rangle$, $\langle f_i \rangle$ i $\langle R_i \rangle$.*

Dokaz. Zapravo je jedini zanimljivi dio vidjeti da je „potenciranje” (uzastopna konkatenacija) znaka odnosno riječi, $\text{Repeat}(w, n) := w^n$, primitivno rekurzivno. Prateća funkcija se lako napiše primitivnom rekurzijom i konkatenacijom:

$$\text{Repeat}(x, 0) = \langle \varepsilon \rangle = 0, \quad (5.35)$$

$$\text{Repeat}(x, n + 1) = \text{Repeat}(x, n) \frown x. \quad (5.36)$$

Sada je $\text{Var}(i) := \langle x' \rangle \frown \dots \frown \langle x' \rangle = 1 \frown \text{Repeat}(6, i)$, i analogno $\text{Const}(i) := 2 \frown \text{Repeat}(6, i)$, $\text{Func}(i) := 3 \frown \text{Repeat}(6, i)$, te $\text{Rel}(i) := 4 \frown \text{Repeat}(6, i)$. \square

Introdukciju logičkih veznika mogli bismo obavljati na sličan način, samo ih treba izraziti u osnovnom jeziku. Recimo, primitivno rekurzivna funkcija

$$\text{Conjunction}(x, y) := \langle \neg \langle \rangle \frown x \frown \langle \rightarrow \neg \rangle \frown y \frown \langle \rangle \rangle = 115 \frown x \frown 129 \frown y \frown 8 \quad (5.37)$$

ima svojstvo da je $\text{Conjunction}(\langle \varphi \rangle, \langle \psi \rangle) = \langle \varphi \wedge \psi \rangle$ (preciznije $\langle \neg(\varphi \rightarrow \neg\psi) \rangle$ u osnovnom jeziku) za sve formule φ i ψ , te se može shvatiti kao introdukcija konjunkcije na kodovima. Za eliminaciju, možemo iskoristiti *brute force* pristup:

$$\text{IsConjunction}(x, y, z) : \Longleftrightarrow y \in \langle F1 \rangle \wedge z \in \langle F1 \rangle \wedge x = \text{Conjunction}(y, z), \quad (5.38)$$

$$\text{LeftConjunct}(x) := (\mu y < x)(\exists z < x) \text{IsConjunction}(x, y, z), \quad (5.39)$$

$$\text{RightConjunct}(x) := (\mu z < x)(\exists y < x) \text{IsConjunction}(x, y, z). \quad (5.40)$$

Naravno, koristili smo propoziciju 5.32 i teorem 4.51, te činjenicu da podriječ ima manji kod nego riječ u kojoj se nalazi (dokažite to!). Brojni drugi trikovi takvog tipa, uključujući i sasvim netrivialne stvari poput supstitucije terma za varijablu u formuli, mogu se vidjeti u [13].

Što se univerzalnog zatvorenja tiče, bitno je uočiti da zapravo uopće ne moramo nalaziti slobodne varijable u formuli — s obzirom na to da je $\forall x\varphi$ ekvivalentna s φ ako x nije slobodna u φ , dovoljno je naći neki nadskup skupa slobodnih varijabli. A kako za svaku varijablu x_n koja se pojavljuje (slobodno ili vezano) u φ vrijedi da je odgovarajuća riječ x', \dots , podriječ duljine $n + 1$ od φ , slijedi da mora biti $n < n + 1 \leq |\varphi| = \text{slh}(\langle \varphi \rangle, 12)$. Dakle, ako nam je zadan kod c formule φ , dovoljno je ispred „nalijepiti” samo one kvantifikatore $\forall x_i$ za $i < \text{slh}(c, 12)$, i možemo biti sigurni da smo dobili univerzalno zatvorenje.

$$\text{UnivQuant}(i) := \langle \forall x_i \rangle = \langle \forall \rangle \frown \langle x_i \rangle = 11 \frown \text{Var}(i) \quad (5.41)$$

$$\text{UnivQuants}(0) := \langle \varepsilon \rangle = 0 \quad (5.42)$$

$$\text{UnivQuants}(i + 1) := \langle \forall x_0 \forall x_1 \forall x_2 \dots \forall x_i \rangle = \text{UnivQuants}(i) \frown \text{UnivQuant}(i) \quad (5.43)$$

$$\text{UnivClosure}(c) := \text{UnivQuants}(\text{slh}(c, 12)) \frown c \quad (5.44)$$

Sada je funkcija **UnivQuant** primitivno rekurzivna po lemi 5.33 i propoziciji 4.61, a onda je **UnivQuants** primitivno rekurzivna po propoziciji 2.20, te je **UnivClosure** primitivno rekurzivna po propoziciji 4.61 i lemi 4.13.

Zapravo, kako u našem slučaju (reprezentacija problema zaustavljanja RAM-stroja preko valjanosti formula prvog reda) koristimo samo prvih m varijabli (m je konstanta), dovoljna će nam biti funkcija $c \mapsto \text{UnivQuants}(m) \frown c$.

5.3.5. Formalna izreka Churchovog teorema

Pogledajmo malo detaljnije kako u osnovnom jeziku izgleda formula φ_u iz dokaza teorema 5.27(4). To svakako ovisi o instrukcijama konkretnog programa P_0 koji računa Russellovu funkciju, ali dio koji ovisi o u (koji nam je jedini bitan za svođenje) je prilično malen i odnosi se samo na formulu π_u .

Preciznije, neka je $P_0 = \left[t. I_t \right]_{t < n}$. Za svaki $t < n$ imamo u Γ_u jednu formulu ι_t , tako da se njihova univerzalna zatvorenja, obrnutim redom, prebacuju s desne strane (5.12), gdje na početku stoji samo formula ζ . Na kraju nam na lijevoj strani ostane samo π_u (formulu koja je u tom trenutku na desnoj stani označimo s ψ), i nju možemo prebaciti lijevo bez univerzalnog zatvorenja jer je ona rečenica. Ključno je da formula ψ , koliko god ogromna bila, ne ovisi o u — ako konstruiramo funkciju od u , to će jednostavno biti neka jako velika konstanta.

Dakle, $\varphi_u := (\pi_u \rightarrow \psi)$, gdje je ψ oblika $(\iota_0 \rightarrow (\iota_1 \rightarrow (\iota_2 \rightarrow \dots (\iota_{n-1} \rightarrow \zeta) \dots)))$. Uz očite reprezentacije (0 kao c_0 odnosno \mathbf{c} , s kao f_0^1 odnosno \mathbf{f} , te R_i kao R_i^m odnosno riječ

koda $\text{Rel}(i)$), lako je svaku pojedinu atomarnu formulu koja se pojavljuje u ψ zapisati u osnovnom jeziku. Jedino što je još potrebno su kondicionali (koje imamo u osnovnom jeziku, samo moramo pisati vanjske zagrade), i konjunkcije za formule oblika (5.15) — koje se mogu reprezentirati kako smo to objasnili u (5.37), ili jednostavno proširujući Γ_u tako da za svaku instrukciju I_t tipa DEC ubacimo *dva* kondicionala (zvat ćemo ih ι_t^0 i ι_t^+) unutra. U praksi je lakše koristiti ovaj drugi pristup.

Primjer 5.34. Promotrimo problem zaustavljanja P-izračunavanja s \vec{u} , gdje je

$$P := \left[\begin{array}{l} 0. \text{ DEC } \mathcal{R}_1, 1 \\ 1. \text{ DEC } \mathcal{R}_1, 1 \end{array} \right], \quad (5.45)$$

a $\vec{u} = (1, 2)$. Lako vidimo da P-izračunavanje s \vec{u} ,

$$(0, 1, 2, 0, \dots, 0) \rightsquigarrow (0, 0, 2, 0, \dots, 1) \rightsquigarrow (0, 0, 2, 0, \dots, 1) \rightsquigarrow \dots, \quad (5.46)$$

ne stane, štoviše, stane ako i samo ako je $u_1 \geq 2$. Prevedeno na jezik logike prvog reda ($m = 3$, $n = 2$):

$$\zeta = \exists x_0 \exists x_1 \exists x_2 R_2(x_0, x_1, x_2) \quad (5.47)$$

$$\iota_0^0 = (R_0(x_0, 0, x_2) \rightarrow R_1(x_0, 0, x_2)) \quad (5.48)$$

$$\iota_0^+ = (R_0(x_0, s(x_1), x_2) \rightarrow R_1(x_0, x_1, x_2)) \quad (5.49)$$

$$\iota_1^0 = (R_1(x_0, 0, x_2) \rightarrow R_1(x_0, 0, x_2)) \quad (5.50)$$

$$\iota_1^+ = (R_1(x_0, s(x_1), x_2) \rightarrow R_2(x_0, x_1, x_2)) \quad (5.51)$$

$$\psi = (\iota_0^0 \rightarrow (\iota_0^+ \rightarrow (\iota_1^0 \rightarrow (\iota_1^+ \rightarrow \zeta)))) \quad (5.52)$$

$$\pi_{(1,2)} = R_0(0, s(0), s(s(0))) \quad (5.53)$$

$$\varphi_{(1,2)} = (\pi_{(1,2)} \rightarrow \psi) \quad (5.54)$$

odnosno u osnovnom jeziku

$$\begin{aligned} \varphi_{(1,2)} = & (R(c, f(c), f(f(c))) \rightarrow \\ & (\forall x \forall x' \forall x'' (R(x, c, x'') \rightarrow R'(x, c, x'')) \rightarrow \\ & (\forall x \forall x' \forall x'' (R(x, f(x'), x'') \rightarrow R'(x, x', x'')) \rightarrow \\ & (\forall x \forall x' \forall x'' (R'(x, c, x'') \rightarrow R'(x, c, x'')) \rightarrow \\ & (\forall x \forall x' \forall x'' (R'(x, f(x'), x'') \rightarrow R''(x, x', x'')) \rightarrow \\ & \neg \forall x \forall x' \forall x'' \neg R''(x, x', x''))))))) , \quad (5.55) \end{aligned}$$

čiji kod je broj s više od 200 znamenaka — ali dio koji ovisi o \vec{u} je samo ovaj prvi red; sve ostalo je konstanta za fiksni RAM-algoritam P^k . \triangleleft

Lema 5.35. Funkcija Fi , koja svakom prirodnom broju u pridružuje kod formule φ_u iz dokaza teorema 5.27(4), primitivno je rekurzivna.

Dokaz. Za početak, označimo s $\psi := \langle \psi \rangle$ kod formule ψ koja nastane prebacivanjem svih u_i na desnu stranu relacije \models . Taj broj ne ovisi o u .

Drugo, označimo s Pi funkciju koja preslikava $u \mapsto \langle \pi_u \rangle$. Ta funkcija je primitivno rekurzivna, jer je funkcija **Numeral**, koja preslikava $u \mapsto \langle \bar{u} \rangle$, primitivno rekurzivna.

$$\text{Numeral}(u) := \langle f(f(\dots f(c) \dots)) \rangle = \text{Repeat}(43, u) \frown 2 \frown \text{Repeat}(8, u) \quad (5.56)$$

$$\begin{aligned} \text{Pi}(u) &:= \langle \pi_u \rangle = \langle R_0(0, \bar{u}, 0, \dots, 0) \rangle = \langle R(c,) \frown \langle \bar{u} \rangle \frown \langle (, c)^{m-2} \rangle \frown \langle \rangle \rangle = \\ &= 7949 \frown \text{Numeral}(u) \frown \text{Repeat}(62, m-2) \frown 8 \end{aligned} \quad (5.57)$$

$$\text{Fi}(u) := \langle \varphi_u \rangle = \langle (\pi_u \rightarrow \psi) \rangle = 7 \frown \text{Pi}(u) \frown 10 \frown \psi \frown 8 \quad (5.58)$$

Repeat nam treba da bismo formalno zapisali ovo označeno trotočkom u π_u — preostalih $m-2$ (konstanta; znamo da je $m \geq 2$) registara su resetirani. Naravno, treba nam i za **Numeral** iz istog razloga: trotočke prije i nakon $f(c)$. \square

Propozicija 5.36 (Churchov teorem formalno). *Jezik Valid nije odlučiv.*

Dokaz. Lanac ekvivalencija (5.25) sada možemo proširiti još trima karikama:

$$\begin{aligned} u \in K &\iff P_0\text{-izračunavanje s } u \text{ stane} \iff (\Gamma_u \models \zeta) \iff \models \varphi_u \iff \\ &\iff \varphi_u \in \text{Valid} \iff \langle \varphi_u \rangle \in \langle \text{Valid} \rangle \iff \text{Fi}(u) \in \langle \text{Valid} \rangle. \end{aligned} \quad (5.59)$$

Predpredzadnja ekvivalencija je definicija skupa **Valid** (u donjem redu je φ_u zapisana u osnovnom jeziku). Predzadnja ekvivalencija je u jednom smjeru definicija (4.102), a u drugom injektivnost kodiranja (propozicija 4.14): ako je $\langle \varphi_u \rangle = \langle \varphi' \rangle$ za neku $\varphi' \in \text{Valid}$, tada je $\varphi_u = \varphi' \in \text{Valid}$. Zadnja ekvivalencija je definicija funkcije **Fi**.

Sada se čitav lanac može sažeti u $\chi_K = \chi_{\langle \text{Valid} \rangle} \circ \text{Fi}$, te je **Fi** (čak primitivno) rekurzivna, odnosno vrijedi $K \preceq \langle \text{Valid} \rangle$. Kada bi **Valid** bio odlučiv, po teoremu 4.51 bi jednomjesna relacija $\langle \text{Valid} \rangle$ bila rekurzivna. Prema propoziciji 5.14, tada bi i Russellov skup bio rekurzivan, što je kontradikcija s teoremom 5.11. \square

5.4. Prema Gödelovom prvom teoremu nepotpunosti

Argumentima koji su potrebni za dokaz Churchovog teorema već smo se poprilično približili dokazu Gödelovog prvog teorema, izrečenog u obliku: **Postoji istinita rečenica na prirodnim brojevima, nedokaziva u PA**. Za pravi dokaz trebalo bi ispeglati još puno tehničkih detalja, ali pogledajmo samo osnovnu ideju.

U prethodnoj točki uveli smo, za svaki prirodni broj u , formulu φ_u , koja je valjana ako i samo ako je u element Russellovog skupa. Štoviše, lako je vidjeti da je za svaki u , φ_u *rečenica*: nema slobodnih varijabli. Također smo uveli strukturu $\mathfrak{N} = (\mathbb{N}, 0, \text{Sc}, \dots)$, čiji konstantsko-funkcijski fragment se podudara sa standardnim modelom od PA (prirodni brojevi, nula i sljedbenik).

Definicija 5.37. *Gödelov skup* je skup formula $\mathcal{G}\ddot{o} := \{\neg\varphi_u \mid u \in K^c\}$. \triangleleft

Kako je negacija rečenice ponovo rečenica, Gödelov skup je skup rečenica.

Propozicija 5.38. *Vrijedi $\mathfrak{N} \models \mathcal{G}\ddot{o}$.*

Dokaz. Po kontrapoziciji propozicije 5.24, za svaki $u \in K^c$ formula φ_u nije valjana, a iz dokaza te propozicije slijedi da ne vrijedi upravo na strukturi \mathfrak{N} (jer tamo vrijedi π_u i sve ι_i , ali ne vrijedi ζ). Iz logike znamo da $\mathfrak{N} \not\models \varphi_u$ povlači $\mathfrak{N} \models \neg\varphi_u$, jer je φ_u rečenica. Drugim riječima, za svaki $u \in K^c$ vrijedi $\mathfrak{N} \models \neg\varphi_u$, što smo trebali dokazati. \square

Teorem 5.39. *Postoji rečenica $\gamma \in \mathcal{G}\ddot{o}$ (Gödelova rečenica) koja nije dokaziva u PA.*

Skica dokaza. Ključno je vidjeti da je jezik nad Σ_{\log} ,

$$\text{Proof} := \{ \psi_0 \# \psi_1 \# \dots \# \psi_n \mid (\psi_i)_{i=0}^n \text{ je dokaz u logici prvog reda} \}, \quad (5.60)$$

odlučiv. To sigurno nećemo napraviti u potpunosti, ali sve osnovne ideje već imamo. Prvo, slično kao u točki 4.4.4, možemo napraviti funkcije arg'_i , koje iz kodiranog $\#$ -separiranog niza formula ekstrahiraju pojedinu formulu (zapravo njen kod). Dinamizacijom možemo napraviti i $(\text{arg}')^2$, pomoću koje onda možemo pričati o svim formulama u dokazu (i osigurati da su dijelovi između separatora doista formule). Sad je jasno da, kako bi takav niz bio u Proof, svaka formula u njemu mora biti ili instanca sheme aksioma (A1), (A2), (A3), (A4) ili (A5) (iz Hilbertovog računa predikata), ili pak mora biti dobivena modus ponensom ili generalizacijom iz neke prethodne formule.

$$\begin{aligned} p \in \langle \text{Proof} \rangle \Leftrightarrow & \left(\forall i < 2 + (\#j < \text{slh}(p, 12)) (\text{sdigit}(p, j, 12) = 12) \right) \left(\text{arg}'(p, i) \in \langle F1 \rangle \wedge \right. \\ & (\text{lsA1}(\text{arg}'(p, i)) \vee \text{lsA2}(\text{arg}'(p, i)) \vee \text{lsA3}(\text{arg}'(p, i)) \vee \text{lsA4}(\text{arg}'(p, i)) \vee \text{lsA5}(\text{arg}'(p, i)) \\ & \vee (\exists j < i)(\exists k < i) \text{ModPon}(\text{arg}'(p, i), \text{arg}'(p, j), \text{arg}'(p, k)) \vee \\ & \left. \vee (\exists j < i) \text{Gen}(\text{arg}'(p, i), \text{arg}'(p, j)) \right) \end{aligned} \quad (5.61)$$

Sheme (A1), (A2) i (A3) lako je karakterizirati konkatencijom. Napisat ćemo je za (A1) — ostale su samo dulje ali ne bitno kompliciranije.

$$\begin{aligned} \text{lsA1}(f) : \Leftrightarrow & (\exists a < f) (\exists b < f) (a \in \langle F1 \rangle \wedge b \in \langle F1 \rangle \wedge \\ & \wedge f = \langle (\rangle \frown a \frown \langle \rightarrow (\rangle \frown b \frown \langle \rightarrow \rangle \frown a \frown \langle \rangle \rangle) \end{aligned} \quad (5.62)$$

Koristili smo relativno očitu činjenicu da za podriječ v od w vrijedi $\langle v \rangle \leq \langle w \rangle$ (za pravu podriječ stroga nejednakost), što nam omogućuje da ograničimo egzistencijalne kvantifikacije.

Za sheme (A4) i (A5) moramo još voditi računa o reprezentaciji terma, i uvjetima za supstituciju, što uvodi određene tehničke poteškoće, ali ništa što naši alati primitivno rekursivnih funkcija ne bi mogli riješiti. Za alternativnu aksiomatizaciju logike prvog reda koja izbjegava brojne tehničke poteškoće, možete pogledati [13].

Modus ponens i generalizacija su sintaksno vrlo jednostavni.

$$\text{ModPon}(a, b, c) : \Longleftrightarrow b = \langle \langle \rangle \wedge c \wedge \langle \rightarrow \rangle \wedge a \wedge \langle \rangle \rangle \quad (5.63)$$

$$\text{Gen}(a, b) : \Longleftrightarrow (\exists k < a)(a = \text{UnivQuant}(k) \wedge b) \quad (5.64)$$

Odnosno, α je dobivena modus ponensom iz β i γ ako i samo ako je $\beta = (\gamma \rightarrow \alpha)$. Također, α je dobivena generalizacijom iz β ako postoji k takav da je $\alpha = \forall x_k \beta$. Pri tome primijetimo da je $k < k + 1 = |x_k| < |\forall x_k| < |\alpha| \leq \langle \alpha \rangle = a$ po lemi 4.38, pa možemo ograničiti kvantifikaciju po k .

Kad smo se koliko-toliko uvjerali da je Proof odlučiv, trebamo isto provesti za Proof_{PA} , jezik koji sadrži dokaze u Peanovoj aritmetici. Nije tako strašno: među aksiome treba dodati i nelogičke, Peanove aksiome, te provjeriti za svaku formulu u dokazu je li instanca sheme aksioma matematičke indukcije. Ovo posljednje je također petljavo, ali iz istog razloga kao (A4): supstitucija je komplicirana. Za elegantan pristup koji zaobilazi supstituciju u slučaju PA, pogledajte [13].

Pogledajmo sada sljedeći *paralelni* algoritam:

Za ulaz $u \in \mathbb{N}$, prvo zapišemo $\neg \varphi_u$ tako da, osim nule i sljedbenika, koristi samo zbrajanje i množenje (nazovimo taj zapis $\neg \varphi'_u$). Nakon toga pokrenemo dvije dretve.

U prvoj, jednostavno simuliramo P_0 -izračunavanje s u , odnosno računanje funkcije Russell u točki u . Ako ova dretva stane, prekidamo program i vraćamo *true* (1).

U drugoj dretvi, generiramo sve elemente od Proof_{PA} redom po kodovima. Za svaki $p \in \text{Proof}_{\text{PA}}$, nađemo zadnju pojavu znaka $\#$ u p , i usporedimo dio p od te pojave (isključivo) do kraja s formulom $\neg \varphi'_u$ dobivenom na početku. Ako su jednake, prekidamo program i vraćamo *false* (0).

Očito, ako je $u \in K$, prva dretva će stati, pa će i cijeli algoritam stati i vratiti 1 — pod pretpostavkom da druga dretva ne zaustavi algoritam prije. No PA je konzistentna teorija, pa ne dokazuje ono što nije istina. Kako $\neg \varphi_u$ nije istinita u \mathfrak{N} , tako niti $\neg \varphi'_u$ nije istinita u \mathbb{N} , dakle ne može se dogoditi da neki $p \in \text{Proof}_{\text{PA}}$ završi njome.

S druge strane, ako $u \notin K$, prva dretva po definiciji neće stati, no hoće li druga? Ako bi PA bila i *potpuna* teorija, odnosno dokazivala sve što je istina u strukturi \mathbb{N} , tada bi s vremenom na red došao i dokaz za $\neg \varphi'_u$. To bi značilo da za sve $u \notin K$, naš algoritam također stane i vraća 0.

No to bi značilo da smo upravo napisali (vrlo neformalni) algoritam za računanje funkcije χ_K , što je po Church–Turingovoj tezi u kontradikciji s teoremom 5.11.

Jedina pretpostavka koja može biti kriva je da PA dokazuje sve formule oblika $\neg\varphi'_u, u \in K^c$ — drugim riječima, postoji formula iz \mathcal{G}_0 koja nije dokaziva u PA. \square

5.4.1. Problemi u skici dokaza Gödelovog teorema

U prethodnoj točki je mnogo toga napravljeno neformalno. Želja autora je bila prikazati da smo idejno jako blizu, ali naravno, tehnička zahtjevnost takvog rezultata je ogromna. Samo navedimo neke najočitije rupe, i ukratko opišimo kako se „krpaju”.

- Možemo li doista relacije R_n izraziti pomoću nule, sljedbenika, zbrajanja i množenja? Lako se vidi, koristeći funkciju [Reg](#), da je svaku od njih moguće izraziti kao projekciju primitivno rekurzivne relacije, no kodiranje koje pritom koristimo na vrlo bitan način ovisi o operaciji potenciranja. Pokazuje se [\[13\]](#) da je Gödelov teorem bitno jednostavnije dokazati ako u strukturi \mathbb{N} shvatimo i potenciranje kao osnovnu operaciju. Zapisati potenciranje *logički* pomoću zbrajanja i množenja (ne koristeći primitivnu rekurziju, jer ona ide preko teorije skupova, Dedekindovim teoremom rekurzije), ili alternativno, razviti kodiranje konačnih nizova koje umjesto množenja i potenciranja koristi zbrajanje i množenje, vrlo je teško. Gödelov originalni pristup (β -funkcija i kineski teorem o ostacima) je zapetljan, ali do danas nismo otkrili ništa bolje.
- Kako iz činjenice da je skup $\langle \text{Proof}_{\text{PA}} \rangle$ rekurzivan možemo dobiti njegovu izračunljivu enumeraciju? Ovo je zapravo najjednostavnije, i možete to riješiti za vježbu. Dakle, zadatak je dokazati: ako je S beskonačan rekurzivan skup, tada je enumeracija od S (funkcija koja broju i pridružuje i -ti po veličini element od S) rekurzivna. Potrebnu ideju vidjeli ste kod enumeracije skupa \mathbb{P} [\(3.10\)](#). Očito je Proof_{PA} beskonačan skup.
- Pokriva li Church–Turingova teza i paralelne algoritme? Svakako, iako će proći još neko vrijeme do trenutka kad ćemo to dokazati u poglavlju [7](#). Usprkos nemogućnosti dokaza same teze, dokazat ćemo da se paralelni algoritmi mogu izvršavati na RAM-stroju, odnosno opisati parcijalno rekurzivnim funkcijama, te se imaju jednako pravo zvati algoritmima kao i oni slijedni (neparalelni). Konkretno, tvrdnja koja nam ovdje treba formalizirana je kao Postov teorem, koji je dokazan u točki [7.3](#).
- Možemo li potpuno izbjeći Church–Turingovu tezu? Kako teorem govori o dokazivanju a ne o algoritmima, čini se da možemo — ili je bar možemo zamijeniti „Hilbertovom tezom” da je intuitivni pojam dokazivanja u nekoj teoriji upravo

onaj koji je opisan preciznom definicijom formalnog dokaza. Taj pristup je naravno potpuno izvan opsega ove knjige, kojoj su centralna tema algoritmi.

- Nekonstruktivnost dokaza svakako upada u oči. Dokazali smo da Gödelova rečenica postoji tako što smo zapravo dokazali da pretpostavka da su u nekom beskonačnom skupu sve rečenice dokazive vodi na kontradikciju. Tada nedokaziva rečenica u tom skupu postoji u klasično-logičkom smislu, ali svejedno o njoj ne znamo skoro ništa. Možemo li *konstruirati* Gödelovu rečenicu? Odgovor je potvrđan, iako to uopće nije jednostavno vidjeti. Ključna je ideja **samoreferiranja**: moguće je u Peanovoj aritmetici konstruirati formulu koja na određeni način govori o samoj sebi — konkretno, kaže da ona sama nije dokaziva. Iako nećemo pričati direktno o tome, napraviti ćemo sličnu stvar za algoritme, u teoremu rekurzije: algoritmi u svojoj definiciji mogu koristiti (najčešće *pozivati* pomoću funkcije comp_k , gdje je k mjesnost algoritma — ali mogući su i drugi oblici korištenja) same sebe. Osnovna ideja — specijalizacija/supstitucija i dijagonalna funkcija — je ista kao i za samoreferirajuće formule. To je tema sljedećeg poglavlja.

Poglavlje 6.

Metaprogramiranje

U ovom poglavlju dokazat ćemo nekoliko rezultata, kojima je svima zajednička jedna ideja: algoritmi mogu preko kodiranja raditi na raznim tipovima podataka, pa tako mogu raditi i na algoritmima. Recimo, u lemi 2.8, nismo mi samo dokazali da ako su $G_1, G_2, \dots, G_l, H \in \text{Comp}$ (odgovarajućih mjesnosti), tada je i $H \circ (G_1, \dots, G_l) \in \text{Comp}$ — već smo doista konstruirali funkciju $\text{compose}: \text{Prog}^+ \times \text{Prog} \times \mathbb{N}_+ \rightarrow \text{Prog}$, koja prima RAM-programe $P_{G_1}, P_{G_2}, \dots, P_{G_l}$ i P_H što računaju odgovarajuće funkcije (i mjesnost k), te od njih sastavlja RAM-program Q_F^b što računa njihovu kompoziciju. Funkcija compose (odnosno compose_{kl} , jer je parametrizirana mjesnostima) je izračunljiva, i ovdje ćemo to doista dokazati. Ipak, nećemo programirati direktno dokaz leme 2.8, jer on ide preko spljoštenja i funkcijskog makroa, što je dosta komplicirano za izvesti formalno — ići ćemo „zaobilaznim putem”, za koji će se poslije ispostaviti da je zapravo prilično koristan.

6.1. Specijalizacija

Kao što smo već nagovijestili, početak ćemo sa *specijalizacijom*, načinom da u određenom algoritmu fiksiramo određeni (zadnji) ulazni podatak na neki konkretni broj. Dosad smo vidjeli mnoge primjere specijalizacije:

- Funkcija blh iz leme 4.64 dobivena je specijalizacijom funkcije slh , fiksirajući njen zadnji (drugi) argument na 2. Skraćeno kažemo da je blh 2-specijalizacija slh , i pišemo $\text{blh} = \text{spec}(2, \text{slh})$.
- Operacija \frown je 12-specijalizacija funkcije sconcat (propozicija 4.61).
- Općenito u primitivnoj rekurziji (2.10), baza je zapravo 0-specijalizacija definirane funkcije: $G = \text{spec}(0, G \text{ } H)$.
- Za svaki konkretni $e \in \mathbb{N}$ i $k \in \mathbb{N}_+$, funkcija $\{e\}^k$ je e -specijalizacija funkcije comp_k .

- Baza G_3 za funkciju **Tape** (lema 4.22 — 0-specijalizacija **Tape**) je dobivena specijalizacijom funkcije **Recode**, fiksirajući njena dva zadnja argumenta na b' i b redom. Vidimo da možemo specijalizirati i s obzirom na više argumenata, što je samo višestruka primjena specijalizacije jednog argumenta.

$$G_3 = \text{spec}(0, \text{baza}) = \text{spec}(b', b, \text{Recode}) = \text{spec}(b', \text{spec}(b, \text{Recode})) \quad (6.1)$$

Ideja specijalizacije je time na neki način suprotna ideji dinamizacije, koja iz familije $f_i^k(\vec{x}), i \in \mathbb{N}$ konstruira $f^{k+1}(\vec{x}, i)$ — ovdje iz $f^{k+1}(\vec{x}, i)$ i konkretnog broja i_0 dobijemo funkciju $f_{i_0}^k$, koja preslikava \vec{x} u $f(\vec{x}, i_0)$. Iako dinamizacija općenito nije izračunljivo preslikavanje na funkcijama, specijalizacija jest, i to nam je cilj ovdje dokazati.

U literaturi se za specijalizaciju još koristi naziv *parcijalna evaluacija*, posebno kod algoritamske optimizacije — jer je ideja da ako imamo fiksnu vrijednost nekog argumenta, određeni dio izraza možemo izračunati unaprijed, i tako pojednostaviti izraz. Sličnu ideju imamo u konkretnoj matematici, kroz pojam *zatvorenog oblika*: npr. iako je teško izračunati općenite vrijednosti funkcije $f(n, k) := \sum_{j=1}^n j^k$, za $k = 2$ dobivamo jednostavnu formulu $f(n, 2) = \frac{n(n+1)(2n+1)}{6}$. U logičkom kontekstu, tome odgovara *supstitucija*, gdje iz formule s $k + 1$ slobodnih varijabli dobijemo formulu s k slobodnih varijabli, uvrštavajući za neku varijablu neki zatvoreni term (koji u standardnom modelu od PA odgovara upravo nekom fiksnom prirodnom broju). Vidimo da se slična ideja pojavljuje pod raznim imenima na mnogim mjestima, što upućuje na njenu korisnost.

Ono što je sasvim nesporno, i lako dokazati (i već smo to dokazali u svim specijalnim slučajevima navedenima na početku ove točke), je da specijalizacija čuva izračunljivost. Zbog teorema ekvivalencije, svejedno je u kom obliku konkretiziramo tu izračunljivost, a dokaz je najlakši za parcijalnu rekurzivnost.

Propozicija 6.1. *Neka su $k \in \mathbb{N}_+$, $y \in \mathbb{N}$, te f^{k+1} parcijalno rekurzivna funkcija.*

Tada je i funkcija $g^k = \text{spec}(y, f)$, zadana s $g(\vec{x}) \simeq f(\vec{x}, y)$, parcijalno rekurzivna.

Dokaz. Definiciju od g možemo zapisati u simboličkom obliku

$$g = f \circ (I_1^k, I_2^k, \dots, I_k^k, C_y^k), \quad (6.2)$$

što vrijedi jer imamo $\mathcal{D}_g = \{\vec{x} \in \mathbb{N}^k \mid (\vec{x}, y) \in \mathcal{D}_f\}$, što je upravo domena desne strane jer su sve koordinatne projekcije, kao i konstanta C_y^k , totalne. Sada tvrdnja slijedi iz zatvorenosti skupa svih parcijalno rekurzivnih funkcija na kompoziciju. \square

Korolar 6.2. *Ako je $k \in \mathbb{N}_+$, $y \in \mathbb{N}$, te f^{k+1} (primitivno) rekurzivna, tada je i $g^k = \text{spec}(y, f)$ (primitivno) rekurzivna.*

Dokaz. Isti kao gore — zapravo jednostavniji, jer ne moramo određivati domenu. \square

Ipak, ta tvrdnja nije glavna tema ove točke. Konkretizirajući izračunljivost kroz postojanje indeksa, ona samo kaže da ako f ima indeks, tada i g ima indeks — ne kaže, barem ne direktno, kako *izračunati* indeks od g pomoću indeksa od f . Primijetimo samo da nije veliki problem izračunati (neke) indekse od l_i^k za $i \in [1..k]$, i od C_y^k — te kad bismo imali *compose* kao izračunljivu funkciju na indeksima, mogli bismo kao specijalni slučaj dobiti i indeks kompozicije (6.2). Ipak, lakše je prvo realizirati specijalizaciju u obliku izračunljive funkcije S , a onda *compose* pomoću te funkcije.

Na neki način, $S = \mathbb{N}\text{spec}$, gdje funkcije kodiramo njihovim indeksima na način koji je detaljno objašnjen u primjeru 3.59. Ukratko, S prima y i *bilo koji* indeks za f , te vraća *neki* indeks za $\text{spec}(y, f)$.

Bilo bi super kad bismo mogli dobiti S kao jednu funkciju, neovisnu o mjesnosti k funkcije f — no lako se vidi da to, barem ovako kako smo zamislili indekse, nije moguće. Ta funkcija bi trebala primati y i kod RAM-programa e , i vraćati kod novog RAM-programa e' takvog da bude $\{e\}(\vec{x}, y) \simeq \{e'\}(\vec{x})$ za sve $\vec{x} \in \mathbb{N}^k$. No to znači da *registar* u kojem će pisati ulazni podatak y u odgovarajućem RAM-stroju (\mathcal{R}_{k+1}) ovisi o k , a k ne možemo zaključiti iz samog RAM-programa odnosno njegovog koda e .

Napomena 6.3. Zapravo, mogli bismo dokazati i „uniformnu” verziju, kad bismo fiksirali argumente s početka a ne s kraja. Tada bi y uvijek išao u \mathcal{R}_1 , a sve ostale registre (koji se stvarno spominju u programu, što *možemo* zaključiti iz e) bismo pomicali za jedno mjesto udesno. To se može izvesti (i može poslužiti kao osnova za malo ozbiljniji studentski rad), ali je puno kompliciranije, a nama će ova verzija biti dovoljna. \triangleleft

6.1.1. Teorem o parametru

Propozicija 6.4 (Teorem o parametru). *Za svaki $k \in \mathbb{N}_+$ postoji primitivno rekurzivna funkcija S_k^2 takva da za sve $y, e \in \mathbb{N}$ vrijedi $\{S_k(y, e)\}^k = \text{spec}(y, \{e\}^{k+1})$.*

Posljednja funkcijska jednakost može se zapisati i kao

$$\text{comp}_k(\vec{x}, S_k(y, e)) \simeq \text{comp}_{k+1}(\vec{x}, y, e), \text{ za sve } \vec{x} \in \mathbb{N}^k \quad (6.3)$$

(i to je razlog za „obrnuti” redoslijed pisanja argumenata od S).

Dokaz. Kao što smo već rekli, trebamo „hardkodirati” y u registar \mathcal{R}_{k+1} , i nakon toga pustiti RAM-program P da računa s ulaznim podacima \vec{x} (u registrima \mathcal{R}_1 do \mathcal{R}_k , kao ulazni podaci za $\{S_k(y, e)\}^k$) i y (u registru \mathcal{R}_{k+1}). Program P je onaj program čiji je e kod — no što ako takvog nema?

Prema propoziciji 3.57(2), tada ($e \notin \text{Prog}$) je $f := \{e\}^{k+1} = \emptyset^{k+1}$, pa je lako vidjeti da je tada i $\text{spec}(y, f) = \emptyset$ — naime, pomoću formule za domenu kompozicije, funkcija (6.2) ima praznu domenu. Tada nam za povratnu vrijednost od S treba neki

indeks od \emptyset^k , te je najjednostavnije upotrijebiti upravo e — kako je $e \notin \text{Prog}$, za svaku mjesnost k će biti $\{e\}^k = \emptyset^k$, opet prema propoziciji 3.57(2).

Sada promotrimo slučaj kad takav P postoji — zbog injektivnosti kodiranja programa taj P je jedinstven. Zapravo trebamo spljoštiti makro-program

$$Q := \left[\begin{array}{c} 0. \text{ INC } \mathcal{R}_{k+1} \\ 1. \text{ INC } \mathcal{R}_{k+1} \\ \vdots \\ (y-1). \text{ INC } \mathcal{R}_{k+1} \\ \hline y. P^* \end{array} \right] \quad (6.4)$$

koji se prirodno sastoji od dva dijela (razdvojena crtom) — dakle kôd spljoštenja bit će dobiven konkatenacijom dva koda. Prvi je vrlo jednostavan, i tehniku smo već vidjeli u primjeru 3.28 — s jedinom razlikom što umjesto konstante $6 = \lceil \text{INC } \mathcal{R}_0 \rceil$ imamo konstantu $\lceil \text{INC } \mathcal{R}_{k+1} \rceil = \text{codeINC}(k+1) = 2 \cdot 3^{k+2}$ (to doista jest konstanta jer smo fiksirali k). Dakle, prvi kod je $\overline{G}(y)$, za $G := C_{\text{codeINC}(k+1)}$.

Drugi kod se dobiva spljoštenjem makroa rednog broja y . Postupak je detaljno opisan u definiciji 1.22 — no kako je taj makro upravo zadnja instrukcija u Q , ne treba provoditi korak (2), već samo korake (1) i (3). Drugim riječima, umjesto jedne instrukcije $y. P^*$ treba stajati n_P instrukcijâ samog programa P — s tom razlikom što su im redni brojevi i odredišta svi pomaknuti za y . Za redne brojeve pobrinut će se konkatenacija, jer je $\text{lh}(\overline{G}(y)) = y$, ali kako pomaknuti odredišta?

Kod dokazivanja konkatenacije primitivno rekurzivnom (lema 3.18), vidjeli smo ideju „točkovne definicije konačnog niza”. Zbog toga, dovoljno je objasniti kako pomaknuti odredište pojedine instrukcije. Dakle, drugi kod će biti $\overline{H}(y, e, \text{lh}(e))$, gdje je $H(y, e, t) := \text{Shift}(y, e[t])$, i još trebamo definirati funkciju Shift koja pomiče odredište (ako postoji) instrukcije zadane kodom (drugi argument) za y (prvi argument).

Koristeći komponente dest i reg , te tipove instrukcija (leme 3.26 i 3.25), definiramo:

$$\text{Shift}(y, i) := \begin{cases} \text{codeDEC}(\text{reg}(i), \text{dest}(i) + y), & \text{InsDEC}(i) \\ \text{codeGOTO}(\text{dest}(i) + y), & \text{InsGOTO}(i) \\ i, & \text{inače} \end{cases} \quad (6.5)$$

Radi totalnosti, trebamo nekako definirati Shift i kad drugi argument nije u Ins , ali jer ćemo je pozivati samo na $e[t]$ za $\text{Prog}(e)$ i $t < \text{lh}(e)$, zapravo je nebitno kako tamo djeluje. Tada je Shift primitivno rekurzivna po teoremu 2.44, pa je takva i $H^3 = \text{Shift} \circ (I_1^3, \text{part} \circ (I_2^3, I_3^3))$. Naravno, G je primitivno rekurzivna po propoziciji 2.19, te su njihove povijesti \overline{G} i \overline{H} primitivno rekurzivne po lemi 3.15. Sada prema lemi 3.18 i teoremu 2.44 napokon možemo utvrditi primitivnu rekurzivnost funkcije

$$S_k(y, e) := \begin{cases} \overline{G}(y) * \overline{H}(y, e, \text{lh}(e)), & \text{Prog}(e) \\ e, & \text{inače} \end{cases} \quad (6.6)$$

Još treba vidjeti da k -mjesna funkcija s tim indeksom zadovoljava jednadžbu (6.3), no to je lako rastavljanjem na korake: gornja razina makro programa (6.4) je sasvim sekvencijalna. Dakle, za svaki $\vec{x} \in \mathbb{N}^k$ imamo

	\mathcal{R}_0	\mathcal{R}_1	\mathcal{R}_k	\mathcal{R}_{k+1}	$\mathcal{R}_{k+2}..$	
	0	x_1	x_k	0	0	
0. INC \mathcal{R}_{k+1}	0	x_1	x_k	1	0	
1. INC \mathcal{R}_{k+1}	0	x_1	x_k	2	0	
\vdots						
$(y-1)$. INC \mathcal{R}_{k+1}	0	x_1	x_k	y	0	
y . P^*	$\{e\}(\vec{x}, y)$?	?	?	?	,

(6.7)

pa je izlazni podatak doista $\{e\}^{k+1}(\vec{x}, y) \simeq \text{comp}_{k+1}(\vec{x}, y, e)$ — naravno, ako P -izračunavanje s (\vec{x}, y) uopće stane. Ako ne stane, onda ne stane ni Q -izračunavanje s \vec{x} , jer zapne na donjoj razini izvršavajući zadnju makro-instrukciju. \square

6.1.2. Teorem o parametrima

Jednom kad imamo teorem o parametru, njegovom iteracijom (ponovljenom primjenom) lako dobijemo specijalizaciju više zadnjih argumenata izračunljive funkcije odjednom. Primjer, koji može poslužiti i kao ideja dokaza, smo vidjeli u (6.1).

Korolar 6.5. *Za sve $k, l \in \mathbb{N}_+$ postoji primitivno rekurzivna funkcija S_{kl}^{l+1} takva da za sve $\vec{x} \in \mathbb{N}^k$, za sve $\vec{y} \in \mathbb{N}^l$, i za sve $e \in \mathbb{N}$, vrijedi*

$$\text{comp}_k(\vec{x}, S_{kl}(\vec{y}, e)) \simeq \text{comp}_{k+l}(\vec{x}, \vec{y}, e). \quad (6.8)$$

Dokaz. Fiksirajmo k , i dokažimo (za taj k) tvrdnju indukcijom po l . Kako je $l \in \mathbb{N}_+$, baza je $l = 1$, i naravno, $S_{k1} := S_k$ je primitivno rekurzivna po propoziciji 6.4.

Pretpostavimo sad da za neki l postoji primitivno rekurzivna funkcija S_{kl} sa svojstvom (6.8). Kako bismo definirali $S_{k(l+1)}$? Uzmimo proizvoljnu $(l+1)$ -torku $(\vec{y}, z) \in \mathbb{N}^{l+1}$, i računamo zdesna:

$$\text{comp}_{k+l+1}(\vec{x}, \vec{y}, z, e) \simeq \text{comp}_{k+l}(\vec{x}, \vec{y}, S_{k+l}(z, e)) \simeq \text{comp}_k(\vec{x}, S_{kl}(\vec{y}, S_{k+l}(z, e))) \quad (6.9)$$

— iz čega slijedi da je najprirodnije definirati

$$S_{k(l+1)}(\vec{y}, z, e) := S_{kl}(\vec{y}, S_{k+l}(z, e)). \quad (6.10)$$

Dakle, $S_{k(l+1)}$ je definirana kompozicijom iz S_{kl} (koja je primitivno rekurzivna po pretpostavci indukcije), S_{k+l} (koja je primitivno rekurzivna po propoziciji 6.4), i koordinatnih projekcija (koje su primitivno rekurzivne jer su inicijalne), pa je i sama primitivno rekurzivna. Time smo proveli korak indukcije, odnosno za sve pozitivne k i l smo definirali funkciju S_{kl} i dokazali da je primitivno rekurzivna. \square

Primjer 6.6. Drugi supskript funkcije S govori koliko zadnjih argumenata želimo fiksirati (i moramo navesti njihove vrijednosti u pozivu prije indeksa e), a prvi supskript govori koliko još argumenata naša funkcija $\{e\}$ prima (odnosno koliko će ih „preživjeti” specijalizaciju). Recimo, u slučaju funkcije mjesnosti 7 i indeksa e_1 , kojoj želimo fiksirati zadnja tri argumenta redom na brojeve 2, 8 i 5, imali bismo $e_2 := S_{43}(2, 8, 5, e_1) = S_4(2, S_5(8, S_6(5, e_1)))$, i za sve $x, y, z, t \in \mathbb{N}$ bi vrijedilo $\{e_2\}(x, y, z, t) \simeq \{e_1\}(x, y, z, t, 2, 8, 5)$. \triangleleft

Sada možemo i vidjeti da je $Ncompose$, barem za fiksne mjesnosti k i l , primitivno rekurzivna. Tehnika koju ćemo pritom koristiti može se općenito primijeniti na razne funkcije definirane na funkcijama, koje su kodirane preko njihovih indeksa.

Primijetimo da ovdje *ne* govorimo o primitivnoj rekurzivnosti kompozicije (rezultata komponiranja), niti o primitivnoj rekurzivnosti funkcija koje ulaze u kompoziciju. Radi ilustracije uzmimo $k = l = 1$: tada nas ne zanima jesu li G i H (pa time i $F := G \circ H$) primitivno rekurzivne. Sve što nas zanima je da su *parcijalno* rekurzivne, odnosno da imaju indekse (redom ih označimo s g, h i f). Govorimo o primitivnoj rekurzivnosti funkcije $compose_{11}$ koja uzima g i h , te vraća f — pogledajte primjer 3.59 ako ste zaboravili na što se tu misli.

Ugrubo, *kompajlirati* kompoziciju dvije funkcije koje već imamo kompajlirane, je daleko „pitomije” nego *izvršiti* (evaluirati) tu kompoziciju. Konkretno, izvršavanje ne mora stati, ako te funkcije nisu totalne. Sâmo kompajliranje kompozicije, s druge strane, mora stati — i to u unaprijed specificiranom broju koraka.

Propozicija 6.7. *Za sve $k, l \in \mathbb{N}_+$ postoji primitivno rekurzivna funkcija $compose_{kl}^{l+1}$, takva da za sve $g_1, g_2, \dots, g_l, h \in \mathbb{N}$ vrijedi*

$$\{compose_{kl}(g_1, \dots, g_l, h)\}^k = \{h\}^l \circ (\{g_1\}^k, \dots, \{g_l\}^k). \quad (6.11)$$

Dokaz. Zapravo ćemo programirati čitav postupak računanja kompozicije u proizvoljnom $\vec{x} \in \mathbb{N}^k$, ali ga nećemo pokrenuti — nego ćemo iz njegovog indeksa specijalizacijom dobiti indeks same kompozicije.

Dakle, definirajmo funkciju F_{kl} točkovno s

$$F_{kl}(\vec{x}^k, \vec{g}^l, h) := comp_l(comp_k(\vec{x}, g_1), \dots, comp_k(\vec{x}, g_l), h). \quad (6.12)$$

Funkcija F_{kl} je definirana kompozicijom iz parcijalno rekurzivnih funkcija $comp_k$ i $comp_l$, te koordinatnih projekcija — pa je parcijalno rekurzivna. Po korolaru 3.55, F_{kl} ima indeks; označimo ga s e (sjetite se napomene 3.56).

Sada definiramo $compose_{kl}$ kao funkciju koja prima g_1, \dots, g_l i h , te fiksira zadnjih $l + 1$ argumenata od F_{kl} na te vrijednosti:

$$compose_{kl}^{l+1}(\vec{g}, h) := S_{k(l+1)}(\vec{g}, h, e). \quad (6.13)$$

Dakle, $\text{compose}_{kl} = \text{spec}(e, S_{k(l+1)})$, pa je primitivno rekurzivna po korolaru 6.2. Sada za sve $\vec{x} \in \mathbb{N}^k$, za sve $\vec{g} \in \mathbb{N}^l$ i za sve $h \in \mathbb{N}$ vrijedi

$$\begin{aligned} \{\text{compose}_{kl}(\vec{g}, h)\}(\vec{x}) &\simeq \text{comp}_k(\vec{x}, \text{compose}_{kl}(\vec{g}, h)) \simeq \text{comp}_k(\vec{x}, S_{k(l+1)}(\vec{g}, h, e)) \simeq \\ &\simeq \text{comp}_{k+l+1}(\vec{x}, \vec{g}, h, e) \simeq \{e\}^{k+l+1}(\vec{x}, \vec{g}, h) \simeq F_{kl}(\vec{x}, \vec{g}, h) \simeq \\ &\simeq \text{comp}_l(\text{comp}_k(\vec{x}, g_1), \dots, \text{comp}_k(\vec{x}, g_l), h) \simeq \text{comp}_l(\{g_1\}(\vec{x}), \dots, \{g_l\}(\vec{x}), h) \simeq \\ &\simeq \{h\}(\{g_1\}(\vec{x}), \dots, \{g_l\}(\vec{x})) \simeq (\{h\}^l \circ (\{g_1\}^k, \dots, \{g_l\}^k))(\vec{x}), \quad (6.14) \end{aligned}$$

što smo trebali dokazati. \square

Često se tehnika koju smo upravo koristili ne mora ponovo provoditi, već se rezultat može jednostavno dobiti primjenom funkcije compose_{kl} (za neke k i l).

Primjer 6.8. Za svaki $k \in \mathbb{N}_+$ postoji primitivno rekurzivna funkcija plus_k takva da je za sve $e, f \in \mathbb{N}$, $\text{plus}_k(e, f)$ indeks zbroja funkcija $\{e\}^k + \{f\}^k$. \triangleleft

Dokaz. Kako je $F + G$ samo skraćeni zapis za $\text{add}^2 \circ (F, G)$, sve što nam treba je odgovarajuća funkcija compose , i neki indeks za add^2 . Nije problem napisati RAM-program za add^2 ,

$$P_{\text{add}^2} := \begin{bmatrix} 0. \text{DEC } \mathcal{R}_1, 3 \\ 1. \text{INC } \mathcal{R}_0 \\ 2. \text{GO TO } 0 \\ 3. \text{DEC } \mathcal{R}_2, 6 \\ 4. \text{INC } \mathcal{R}_0 \\ 5. \text{GO TO } 3 \end{bmatrix}, \quad (6.15)$$

kao niti izračunati njegov kod,

$$\alpha := \langle \langle 1, 1, 3 \rangle, \langle 0, 0 \rangle, \langle 2, 0 \rangle, \langle 1, 2, 6 \rangle, \langle 0, 0 \rangle, \langle 2, 3 \rangle \rangle = 2^{22501} \cdot 3^7 \cdot 5^{25} \cdot 7^{8437501} \cdot 11^7 \cdot 13^{649},$$

broj koji jest oveći (ima 7138041 znamenku), ali ipak sasvim točno određen. Sad samo treba napisati $\text{plus}(e, f) := \text{compose}_{k2}(e, f, \alpha)$, odnosno $\text{plus} = \text{spec}(\alpha, \text{compose}_{k2})$ je primitivno rekurzivna. \square

Kad smo uspjeli isprogramirati pisanje programa za kompoziciju, zašto ne bismo istu stvar pokušali i za primitivnu rekurziju, ili minimizaciju? Na prvi pogled, sve što nam treba je točkovni zapis izračunavanja pomoću funkcija comp (odgovarajuće mjesnosti), koji onda predstavlja parcijalno rekurzivnu funkciju, čiji indeks uzmemo i specijaliziramo ga s obzirom na indekse polaznih funkcija. Što može poći krivo?

Pokušajmo što vjernije slijediti postupak koji nas je doveo do funkcije compose_{kl} , za primitivnu rekurziju. Zanimarimo degenerirani slučaj — lako možete vidjeti da se u njemu pojavljuje sasvim isti problem.

Dakle, umjesto zadanih mjesnosti k i l , ovdje imamo samo zadanu mjesnost k . Umjesto $l + 1$ polaznih funkcija, ovdje imamo 2 polazne funkcije, G i H (još moraju biti totalne, ali to ćemo jednostavno shvatiti kao parcijalnu specifikaciju — ako ulazni indeksi nisu indeksi totalnih funkcija, ono što bi trebalo biti „indeks od $G \text{ } \mathbb{R} \text{ } H$ ” jednostavno može biti bilo što). Želimo primitivno rekurzivnu funkciju primRecurse_k , takvu da za sve indekse totalnih funkcija $g, h \in \mathbb{N}$ vrijedi

$$\{\text{primRecurse}_k(g, h)\}^{k+1} = \{g\}^k \text{ } \mathbb{R} \text{ } \{h\}^{k+2}. \quad (6.16)$$

Za `compose` smo uzeli parcijalnu jednakost (2.3), i zapisali je pomoću univerzalnih funkcija kao (6.12). Ako to učinimo s definicijom 2.9, dobit ćemo nešto poput

$$F_k(\vec{x}, y, g, h) \simeq \begin{cases} \text{comp}_k(\vec{x}, g), & y = 0 \\ \text{comp}_{k+2}(\vec{x}, \text{pd}(y), \text{comp}_{k+1}(\vec{x}, \text{pd}(y), f), h), & \text{inače} \end{cases}, \quad (6.17)$$

i sad bismo mogli upotrijebiti teorem 3.61, da nije jednog malog problema: jednadžba (2.11) ima traženu funkciju i s lijeve i s desne strane. Ovaj f koji se nalazi u grani „inače”, to je upravo onaj $\text{primRecurse}_k(g, h)$ kojeg želimo odrediti. Dakle, nemamo ga još, pa ga ne možemo koristiti u definiciji od F_k .

Ili možemo? Sjetimo se napomene 1.2 — ovaj f je upravo „implicitna varijabla” kakve smo rekli da trebamo prenijeti kao argumente. Dakle, na lijevoj strani (6.17) zapravo imamo $F_k(\vec{x}, y, g, h, f)$ — i po teoremu 3.61 to je parcijalno rekurzivna funkcija, te ima indeks ... i tako dalje. No sad bismo u F_k^{k+4} trebali fiksirati f , a to ne znamo kako učiniti jer još uvijek ne znamo koju vrijednost staviti za f .

Kad bismo samo znali odrediti f , dalje bi bilo lako: g i h bismo specijalizirali na isti način kao kod komponiranja. Primijetimo svakako da f ne možemo odrediti jednoznačno, jer naprosto nije jedinstven: $G \text{ } \mathbb{R} \text{ } H$, kao i bilo koja izračunljiva funkcija, ima beskonačno mnogo indeksa — a svi oni mogu poslužiti kao f . No ono što nam može biti bitno, je da *funkcija* $\{f\}^{k+1} = \{g\}^k \text{ } \mathbb{R} \text{ } \{h\}^{k+2}$ bude jednoznačno određena.

Zanemarujući g i h (jer njih znamo specijalizirati) i „utapajući” y među \vec{x} , te zovući tako dobivenu funkciju G_k , vidimo da je općenit oblik naše jednadžbe koju želimo riješiti $\{f\}(\vec{x}) \simeq G_k(\vec{x}, f)$. Takve *opće rekurzije* tema su iduće točke.

6.2. Opće rekurzije

Vidjeli smo da se u metaprogramiranju, a ponekad i u običnom programiranju, pojavljuje potreba za funkcijama čija simbolička definicija koristi njih same. Recimo, definicija funkcije `factorial` koju bismo mogli napisati u programskom jeziku C (zanemarujući iz didaktičkih razloga da za faktorijel postoje jednostavnije implementacije i u jeziku C i u jeziku primitivno rekurzivnih funkcija — primjer 2.22),

```

unsigned factorial(unsigned n){
    if(n) return n*factorial(n-1);
    else return 1;
}

```

može se točkovno zapisati kao

$$\text{factorial}(n) = \begin{cases} n \cdot \text{factorial}(\text{pd}(n)), & n > 0 \\ 1, & \text{inače} \end{cases} \quad (6.18)$$

ili simbolički, kao $\text{factorial} = \{\mathbb{N}_+ : \text{mul}^2 \circ (I_1^1, \text{factorial} \circ \text{pd}), \text{Sc} \circ Z\}$.

Kao *definicija* taj izraz nema smisla, jer je cirkularan — ali može poslužiti kao funkcijska *jednadžba* koju trebamo riješiti po „nepoznanici” *factorial*. Osnovna prednost definicija pred jednadžbama sastoji se u tome da (dobra) definicija uvijek jednoznačno određuje definirani objekt, dok jednadžba može nemati rješenja, ili ih može imati više.

Još jedan detalj treba razmotriti: u (6.18) smo koristili znak jednakosti jer otprije znamo da je faktorijel totalna funkcija — ali opće rekurzije koje koriste samo totalne funkcije mogu kao rješenja imati parcijalne funkcije koje nisu totalne. Recimo,

```

unsigned h(unsigned n){
    if(n==0) return 0;
    else if(n==1) return h(n)+1;
    else return h(n-2)+1;
}

```

— ovakva *h* je definirana samo na parnim brojevima, i svaki preslikava u njegovu polovinu (usporedite s jezičnom funkcijom φ_n iz primjera 4.6). Ovdje smo za $n = 1$ koristili kompoziciju sa sljedbenikom da forsiramo parcijalnost (kao kod Russellove funkcije), ali i da nismo napisali taj $+1$, C bi svejedno računao istu parcijalnu funkciju. Ipak, odgovarajuća točkovna funkcijska jednadžba

$$h(n) \simeq \begin{cases} 0, & n = 0 \\ h(n), & n = 1 \\ h(n \ominus 2) + 1, & \text{inače} \end{cases} \quad (6.19)$$

bi pored tog rješenja (nazovimo ga h_0) imala i razna druga, čak totalna rješenja. Jedno od njih je $h_1(n) := \begin{cases} h_0(n), & 2 \mid n \\ 42 + h_0(n-1), & \text{inače} \end{cases}$ — provjerite!

To samo znači ono što već znamo, da trebamo biti puno oprezniji s parcijalnim funkcijama nego s totalnima, pa će nam od posebnog značaja biti one opće rekurzije koje definiraju (isključivo) totalne funkcije.

Dakle, neformalno, *opća rekurzija* je funkcijska jednadžba $F = \mathcal{M}(\vec{G}, F)$, kojoj s desne strane stoji neka simbolička definicija koja pored nekih već poznatih funkcija G_1, G_2, \dots koristi i funkciju F . Kao i obično, konkretnu funkciju F koja zadovoljava tu funkcijsku jednadžbu zovemo *rješenjem* te jednadžbe. Za konkretnu opću rekurziiju, zanimat će nas tri pitanja:

(Egzistencija) Ima li uopće rješenje?

(Jedinstvenost) Ako ima, je li jedinstveno?

(Totalnost) Ako jest, je li totalno?

Pitanja moramo postavljati tim redom, jer npr. totalnost rješenja ne znači puno ako rješenje nije jedinstveno. Recimo, vidjeli smo jednadžbu (6.19) koja ima totalno rješenje h_1 , ali svejedno nijedan programski jezik ne bi pronašao to rješenje, već bi računao netotalnu funkciju h_0 .

Kao i drugdje gdje se gledaju funkcijske jednadžbe (recimo u teoriji običnih diferencijalnih jednadžbi), rijetko se gledaju za sasvim općenite funkcije. Kod diferencijalnih jednadžbi najčešće se bavimo glatkim funkcijama — ovdje, kod općih rekurzija, bavimo se izračunljivim funkcijama. Zbog teorema ekvivalencije, svejedno je kako manifestiramo tu izračunljivost — uglavnom ćemo to činiti kroz parcijalnu rekurzivnost i postojanje indeksa.

Dakle, zadane su nam parcijalno rekurzivne funkcije G_1, G_2, \dots preko svojih indeksa g_1, g_2, \dots — i tražimo indeks f funkcije F koja zadovoljava opću rekurziiju $F = \mathcal{M}(\vec{G}, F)$. Simboličku definiciju \mathcal{M} je, kao i obično, lakše napisati točkovno, u obliku jedne parcijalno rekurzivne funkcije G , koja prima ulazne podatke \vec{x} , indekse \vec{g} i indeks f ; te vraća vrijednost koja mora biti parcijalno jednaka $F(\vec{x}) \simeq \{f\}(\vec{x}) \simeq \text{comp}(\vec{x}, f)$.

No ako već pišemo točkovnu definiciju, indeksi \vec{g} nam ne trebaju — s obzirom na to da su G_i poznate funkcije, možemo ih jednostavno uklopiti u definiciju funkcije G . Jedina funkcija s kojom to ne možemo učiniti (jer nije još poznata) je funkcija F , tako da njen indeks moramo prenijeti kao dodatni argument funkciji G . Kad G želi pozvati F , na nekim argumentima \vec{y} (koji će najčešće biti na neki način „manji” od \vec{x}), koristit će izraz $\text{comp}_k(\vec{y}, f)$.

(Svakako, moguće su i „simultane opće rekurzije”, gdje imamo *više* nepoznatih funkcija koje sve ovise međusobno na opće-rekurzivan način — no to se može riješiti metodama koje smo već upoznali u dokazu propozicije 3.19.)

Definicija 6.9. Neka je $k \in \mathbb{N}_+$, te G^{k+1} parcijalno rekurzivna funkcija. *Opća rekurzija* je jednadžba (po e)

$$\text{comp}_k(\vec{x}, e) \simeq G(\vec{x}, e), \text{ za sve } \vec{x} \in \mathbb{N}^k. \quad (6.20)$$

Ako konkretni broj $e \in \mathbb{N}$ zadovoljava tu jednadžbu, funkciju $\{e\}^k$ zovemo *rješenjem* opće rekurzije. ◁

Ponekad neformalno govorimo i da je indeks e rješenje, ali bitno je da kažemo da je rješenje *jedinstveno* ako je funkcija jedinstvena, ne indeks. To je bitno jer u uobičajenim situacijama, gdje G koristi e samo za pozivanje F , inače nikad ne bismo imali jedinstvenost: čim postoji neki e , svi ostali indeksi za $\{e\}^k$ su također „rješenja”, a intuitivno znamo (i dokazat ćemo u ovom poglavlju) da ih ima beskonačno mnogo.

Pitanje totalnosti tada postaje pitanje *rekurzivnosti* — jer F ima indeks e , po teoremu ekvivalencije je parcijalno rekurzivna, a takve totalne funkcije zovemo rekurzivnima.

U prethodna dva odlomka, čini se, zanemarili smo pitanje egzistencije, odnosno pravili smo se kao da indeks e koji zadovoljava (6.20) uvijek postoji. Važan i netrivialan *teorem rekurzije* kaže da to doista vrijedi.

Naravno, netrivialan je samo u formalnom smislu: intuitivno, lako je objasniti kako izvršavati opću rekurzivnu funkciju (za zadanu funkciju G), i taj neformalni algoritam po Church–Turingovoj tezi računa parcijalno rekurzivnu funkciju. Ipak, to intuitivno objašnjenje može lako navesti na krivi trag (sjetite se funkcije h_1), a i tvrdnje o jedinstvenosti i rekurzivnosti lakše je dokazivati (najčešće matematičkom indukcijom) jednom kad imamo indeks, nego kad imamo neformalni postupak. Zato je dobro izbjeći Church–Turingovu tezu, i doista konstruirati indeks e s traženim svojstvom.

6.2.1. Dijagonalna funkcija

Najvažnije svojstvo funkcije $F = \{e\}^k$ je da mora „znati svoj indeks”, odnosno pozvana s argumentima \vec{x} , mora se ponašati kao funkcija G pozvana s \vec{x} i još indeksom e .

Recimo, u slučaju funkcije `factorial`, kompajler se brine da poziv te funkcije u njenom kodu doista pozove nju samu (sa za jedan manjim argumentom), ali da kojim slučajem kompajler ne podržava rekurziju, mogli bismo (pokazivač na) funkciju `factorial` prenijeti u samu sebe kao dodatni parametar.

Naravno, u ovom slučaju to uopće ne rješava problem, jer moramo *imati* funkciju `factorial` koju ćemo poslati kao argument, ali zasad pričamo samo o specifikaciji.

```
typedef unsigned funkcija(unsigned n);
unsigned G(unsigned n, funkcija f){
    if(n) return n*f(n-1);
    else return 1;
}
/* ... nekako definiramo factorial */
int main(void){
    return G(5, factorial);
} /* vrati 120 */
```

Matematički, „pokazivač na funkciju” nije ništa drugo nego njen indeks, što motivira sljedeću definiciju.

Definicija 6.10. Neka je $k \in \mathbb{N}_+$, te H^{k+1} parcijalno rekurzivna funkcija. *Dijagonala* funkcije H je k -mjesna funkcija dH , definirana s $dH(\vec{x}) \simeq H(\vec{x}, h)$, gdje je h jedan (fiksni) indeks funkcije H . \triangleleft

Kako funkcija može imati više indeksa, može imati i više dijagonala — odnosno formalno, dijagonala bi se trebala definirati za *indeks*, ne za funkciju. No nama će ionako biti bitna samo prateća funkcija $\mathbb{N}d$, koja je definirana na indeksima.

U „stvarnom životu” nemamo tih problema, jer je praktički jedino što ikad radimo s pokazivačem na funkciju, funkcijski poziv. Standard jezika C čak zabranjuje ikakvu aritmetiku na funkcijskim pokazivačima, ali neki kompajleri to ipak dozvoljavaju. Slično, ovdje bismo mogli h iskoristiti ne samo kao zadnji argument funkcije comp_k , nego raditi razne čudne stvari poput gledanja počinje li „strojni kod” funkcije H instrukcijom $\text{INC}(\text{InsINC}(h[0]))$, i sličnog — ali najčešće to ne činimo, a dok god h koristimo samo za pozivanje funkcije H , svejedno je koji njen indeks smo uzeli.

Lema 6.11. *Za svaki $k \in \mathbb{N}_+$ postoji primitivno rekurzivna funkcija D_k , koja preslikava indeks h funkcije H^{k+1} u indeks njene dijagonale dH , i to upravo one koja je definirana pomoću indeksa h .*

Dokaz. Samo treba raspisati definicije. Želimo da za sve $h \in \mathbb{N}$ i za sve $\vec{x} \in \mathbb{N}^k$, vrijedi

$$\{D_k(h)\}(\vec{x}) \simeq dH(\vec{x}) \simeq H(\vec{x}, h) \simeq \{h\}(\vec{x}, h), \quad (6.21)$$

odnosno u terminima univerzalne funkcije, $\text{comp}_k(\vec{x}, D_k(h)) \simeq \text{comp}_{k+1}(\vec{x}, h, h)$. Ali već znamo jednu takvu funkciju: svakako, $\text{comp}_{k+1}(\vec{x}, h, h) \simeq \text{comp}_k(\vec{x}, S_k(h, h))$. Dakle, možemo jednostavno definirati $D_k(h) := S_k(h, h)$, i ta funkcija je primitivno rekurzivna: $D_k = S_k \circ (I_1^1, I_1^1)$. \square

Dijagonaliziranje funkcije, kao što smo vidjeli, još uvijek nije dovoljno da bismo doista dobili funkciju koja je rješenje opće rekurzije. U programu s funkcijom G , svejedno smo trebali nekako drugačije dobiti *factorial* koju bismo poslali kao njen drugi argument. To se čini prilično beskorisnim (jer jednom kad imamo *factorial*, možemo je jednostavno pozvati bez petljanja s dijagonalom), ali zapravo je vrlo blizu rješenju.

Luda ideja: funkcije *factorial* i G , konceptualno gledano, služe istoj svrsi — računanju faktoriјele zadanog broja. Ako nemamo *factorial*, možemo li umjesto nje upotrijebiti G kao drugi argument od G (efektivno, dijagonalizirati G)?

Doslovno, to neće ići — tipovi ne pašu. Drugi argument funkcije G trebao bi biti pokazivač na jednomjesnu funkciju, dok je sama G , naravno, dvomjesna. C-kompajler će nam samo dati *warning*, jer svi pokazivači su ionako samo „ukrašeni prirodni brojevi”, baš kao i indeksi u svijetu parcijalno rekurzivnih funkcija — ali u strože tipiziranom jeziku, dobili bismo kompajlersku grešku.

C je ovdje izuzetno zanimljiv, jer na mnogim kompajlerima, ako zanemarite *warning*, zapravo ćete dobiti ispravno rješenje (sigurno radi na gcc 4.6.3 — pokušajte na svom omiljenom kompajleru). Kako je to moguće?! Zna li gcc nešto što mi ne znamo?

Ovdje treba znati neke tehničke pojedinosti o kompajliranju jezika C. (Nešto smo već rekli kad smo govorili o funkcijskom makrou.) Kad pozovemo funkciju, njeni argumenti stavljaju se na stog (novi okvir) obrnutim redom, i nakon toga se prenosi kontrola na samu funkciju, koja očekuje argumente na stogu i tamo stavlja povratnu vrijednost. Ovdje smo pozvali funkciju G s dva argumenta: 5 i G. Unutar koda funkcije G, pokazivač f pokazuje na funkciju G. Kad smo f pozvali s jednim argumentom (konkretno, 4), broj 4 je stavljen na stog, ali je ispod njega i dalje ostao pokazivač G, koji je funkcija f veselo koristila kao svoj drugi argument. Dakle, poziv $f(n-1)$ je zapravo interpretiran kao $f(n-1, G)$. Umjesto broja 4 jednom će (nakon još nekoliko takvih poziva) na tom mjestu na stogu završiti povratna vrijednost 24, ali ništa neće dirati ovaj G ispod (cjelobrojno množenje i oduzimanje jednice se obavlja u registrima procesora, ne koristeći stog).

Možemo li to iskoristiti da bismo eksplicitno napisali taj poziv u kodu, i izbjegli *warning*? Da, jer C nam dopušta da ne specificiramo tipove parametara funkcije.

```
typedef unsigned funkcija(); /* prima bilo što, vraća unsigned */
unsigned G(unsigned n, funkcija f){
    if(n) return n * f(n-1, f);
    else return 1;
}
int main(void){
    return G(5, G);
} /* vrati 120 bez warninga, barem na nekim kompajlerima */
```

Ali ako već želimo tako zaobilaziti statičke tipove, puno je jednostavnije prebaciti se na dinamički tipiziran jezik. U Pythonu, stvari rade točno kako smo zamislili.

```
>>> def H(n, f):
...     if n: return n * f(n-1, f)
...     else: return 1
>>> H(7, H)
5040
```

Refaktorirajmo ovu duplikaciju koda za dijagonaliziranje, u definiciji i pozivu H:

```
>>> def d(H):
...     def dH(x):
...         return H(x, H)
...     return dH
```

```
>>> def H(n, f):
...     if n: return d(f)(n-1)
...     else: return 1
>>> factorial = d(H)
>>> factorial(7)
5040
```

Dobra strana toga je da više ne moramo ručno proizvoditi funkciju H iz funkcije G . Prethodno smo to napravili tako što smo, tamo gdje je G pozivala f , u funkciji H pozvali df . Sad to možemo formalizirati, tako da definicija od H pozove G s df automatski.

```
>>> def G(n, f):
...     if n: return n * f(n-1)
...     else: return 1
>>> def H(x, f): return G(x, d(f))
```

I to je dokaz teorema rekurzije (za $k = 1$)! Ako pažljivo pogledamo, vidjet ćemo da nigdje nismo koristili rekurzivne pozive. Sad samo sve to treba napisati matematički.

Teorem 6.12 (Teorem rekurzije). *Neka je $k \in \mathbb{N}_+$, te G^{k+1} parcijalno rekurzivna funkcija. Tada postoji $e \in \mathbb{N}$ takav da za sve $\vec{x} \in \mathbb{N}^k$ vrijedi $\{e\}^k(\vec{x}) \simeq G(\vec{x}, e)$.*

Dokaz. Definiramo funkciju H^{k+1} pomoću $H(\vec{x}, f) \simeq G(\vec{x}, D_k(f))$. Ta funkcija je očito parcijalno rekurzivna (dobivena je kompozicijom iz G , D_k i koordinatnih projekcija), pa ima indeks: označimo ga s h (sjetite se napomene 3.56).

Tvrdimo da je dH rješenje opće rekurzije, odnosno njen indeks $e := D_k(h)$ je traženi broj. Doista, za sve $\vec{x} \in \mathbb{N}^k$ imamo

$$\{e\}(\vec{x}) \simeq dH(\vec{x}) \simeq H(\vec{x}, h) \simeq G(\vec{x}, D_k(h)) \simeq G(\vec{x}, e), \quad (6.22)$$

što smo trebali dokazati. □

6.2.2. Ackermannova funkcija

Sada bismo htjeli vidjeti kako pomoću teorema rekurzije možemo definirati rekurzivne funkcije. Okvirno, metoda je uvijek ista: funkcijsku jednadžbu koju želimo riješiti (po F^k) zapišemo u obliku opće rekurzije (nađemo funkciju G^{k+1}), te nam teorem rekurzije dađe parcijalno rekurzivno rješenje — štoviše, dađe nam njegov indeks e_0 . Sada je potrebno dokazati dvije stvari (ako vrijede): prvo, da je $F_0 := \{e_0\}^k$ jedinstveno rješenje, i drugo, da je F_0 totalna funkcija.

I jedno i drugo su univerzalne tvrdnje na prirodnim brojevima — recimo, jedinstvenost kaže da, kad god je e_1 „rješenje”, za sve $\vec{x} \in \mathbb{N}^k$ vrijedi $\text{comp}(\vec{x}, e_1) \simeq F_0(\vec{x})$;

totalnost kaže da za sve $\vec{x} \in \mathbb{N}^k$ vrijedi $\vec{x} \in \mathcal{D}_{F_0}$ — te ih je uobičajeno dokazivati indukcijom. Didaktički problem je u tome što ako je ta indukcija prejednostavna za provesti, to zapravo znači da je programabilna, pa recimo totalnost ne znači samo rekurzivnost već znači primitivnu rekurzivnost. A tada nam ne treba teorem rekurzije: primitivna rekurzija je dovoljna (kao što je uostalom slučaj i s funkcijom [factorial](#)).

Recimo, ako se totalnost od F_0^1 može dokazati običnom matematičkom indukcijom, to znači da imamo neku ogradu za broj koraka računanja $F_0(0)$, te neki izračunljiv način da iz ograde za broj koraka u računanju $F_0(n)$ dobijemo takvu ogradu za $F_0(n+1)$ — a to zapravo znači da F_0 možemo dobiti (degeneriranom) primitivnom rekurzijom, samo trebamo u funkciji [step](#) ograničiti minimizaciju tom nađenom ogralom. Ako umjesto obične indukcije moramo upotrijebiti jaku indukciju, to na isti način znači da se F_0 može dobiti rekurzijom s poviješću. U slučaju simultane indukcije, imamo simultanu rekurziju, i tako dalje.

Vjerojatno najjednostavniji obrazac indukcije koji nije moguće uloviti u teoriji primitivno rekurzivnih funkcija je *ugnižežđena* indukcija po dvije varijable, gdje iz pretpostavke da tvrdnja vrijedi za neki m i za *sve* n , slijedi da vrijedi za $m+1$ i za *sve* n . Jedan od prvih primjera takve funkcije (čija totalnost zahtijeva takav induksijski obrazac koji nije formalizabilan primitivnom rekurzijom) dao je Hilbertov asistent Wilhelm Friedrich Ackermann 1928. godine.

Ackermann je promatrao niz aritmetičkih operacija: sljedbenik, zbrajanje, množenje, potenciranje, ... nekako je jasno da ga možemo i nastaviti na sličan način; sljedeći član — tetracija — je čak korisna operacija u nekim kombinatornim situacijama. Mala nepravilnost je što je sljedbenik jednomjesna funkcija, a sve ostale su dvomjesne: zato stavimo $A_0^2 := \text{Sc} \circ I_2^2$. Ostale funkcije označimo na očit način: $A_1^2 := \text{add}^2$, $A_2^2 := \text{mul}^2$, $A_3^2 := \text{pow}$, i tako dalje. Lako je vidjeti da se osnovna ideja — svaka osim prve funkcije u tom nizu dobivena je iteracijom prethodne — može iskazati kao

$$A_{n+1}(x, y+1) = A_n(x, A_{n+1}(x, y)), \text{ za sve } x, y, n \in \mathbb{N}, \quad (6.23)$$

što zapravo znači $A_{n+1} = G_{n+1} \circ A_n \circ (I_1^3, I_3^3)$; razlikuju se samo početni uvjeti zadani s $G_n = \text{spec}(0, A_n)$. Prvih nekoliko je doista različito: $G_0(x) = \text{Sc}(0) = 1$, $G_1(x) = x + 0 = x$, $G_2(x) = x \cdot 0 = 0$, $G_3(x) = x^0 = 1$ — ali svi kasniji se obično fiksiraju na 1.

Dakle, za sve $n > 1$ vrijedi $A_{n+1} := C_1^1 \circ A_n \circ (I_1^3, I_3^3)$.

Propozicija 6.13. *Za svaki $n \in \mathbb{N}$, A_n^2 je primitivno rekurzivna.*

Dokaz. Običnom matematičkom indukcijom po n . Za $n = 0$, to je kompozicija inicijalnih funkcija. Za $n \in \{1, 2\}$ to smo već dokazali (primjer 2.11). Za $n \geq 2$, ako je A_n primitivno rekurzivna, tada je $A_{n+1} = C_1^1 \circ A_n \circ (I_1^3, I_3^3)$ simbolička definicija A_{n+1} , iz koje slijedi da je i ona primitivno rekurzivna. \square

Primijetimo sličnost s dokazom propozicije 2.19. No dok smo dinamizirani (zadan argumentom) broj iteracija operatora \circ mogli shvatiti kao primitivnu rekurziju, dinamizirani broj iteracija operatora \mathbb{R} više ne možemo tako shvatiti. Zato dinamizacija familije $A_n, n \in \mathbb{N}$,

$$A(x, y, z) := A_z(x, y), \quad (6.24)$$

nije primitivno rekurzivna. Formalni dokaz je kompliciran, ali ideja je ista kao i ideja dokaza da se npr. add^2 ne može dobiti samo kompozicijom iz inicijalnih funkcija: *raste prebrzo*. Svaka kompozicija s inicijalnom funkcijom može u najboljem slučaju povećati bilo koji argument za 1, dakle pomoću konačno mnogo operatora \circ možemo dobiti samo funkcije oblika $x + c$, gdje je c konstanta a x jedan od argumenata. No $x + y$ raste brže od toga.

Slično je i ovdje: svaka primjena primitivne rekurzije može povećati z samo za 1, pa s konačno mnogo operatora \mathbb{R} možemo postići najviše $A_c(x, y)$, gdje je c konstanta. No A^3 raste brže od toga.

Korolar 6.14. A^3 je totalna funkcija.

Dokaz. Neka je $(x, y, z) \in \mathbb{N}^3$ proizvoljna. Kako je A_z primitivno rekurzivna po propoziciji 6.13, ona je totalna, pa je $\mathcal{D}_{A_z} = \mathbb{N}^2$. Specijalno je $(x, y) \in \mathcal{D}_{A_z}$, pa izraz $A_z(x, y)$ ima smisla — označimo njegovu vrijednost s t . No tada je i $A(x, y, z) = t$, dakle izraz $A(x, y, z)$ također ima smisla, pa je $(x, y, z) \in \mathcal{D}_A$. \square

Svakako, budući da je svaka A_n izračunljiva, i funkcija A bi trebala biti izračunljiva u intuitivnom smislu: ako nam netko dađe (x, y, z) i traži od nas da izračunamo $A(x, y, z)$, zapravo traži da izračunamo $A_z(x, y)$, a to sigurno (za konkretan z) znamo učiniti. Po Church–Turingovoj tezi A bi trebala biti parcijalno rekurzivna, dakle rekurzivna, ali možemo li to dokazati bez primjene Church–Turingove teze? Svakako, i u tome će nam pomoći teorem rekurzije.

Napomena 6.15. Često se u literaturi navodi pojednostavljena, dvomjesna funkcija A^2 , zadana s

$$A(0, n) := n + 1, \quad (6.25)$$

$$A(m + 1, 0) := A(m, 1), \quad (6.26)$$

$$A(m + 1, n + 1) := A(m, A(m + 1, n)), \quad (6.27)$$

pod imenom „Ackermannova funkcija”, iako su je zapravo smislili Rószs Péter i Raphael Mitchel Robinson. Iako s manje specijalnih slučajeva i zato lakša za proučavanje, ta funkcija je daleko manje motivirana i teško je objasniti zašto joj definicija izgleda baš tako. Može se, iako nije baš jednostavno, dokazati da za sve $m, n \in \mathbb{N}$ vrijedi

$$A(m, n) = A(2, n + 3, m) \ominus 3, \quad (6.28)$$

iz čega slijedi da kad bi A^3 bila primitivno rekurzivna, takva bi bila i A^2 . No A^2 nije primitivno rekurzivna (dokaz možete vidjeti u [16]) pa kontrapozicijom slijedi da ni A^3 nije primitivno rekurzivna. \triangleleft

6.2.3. Rekurzivnost Ackermannove funkcije

Propozicija 6.16. A^3 je rekurzivna funkcija.

Dokaz. Očito, koristit ćemo teorem rekurzije. Prvo skupimo sve jednadžbe kojima je A^3 definirana na jedno mjesto. U to svakako ulazi (6.23), sada u obliku (6.29), koji kazuje što činiti kad su i drugi i treći argument pozitivni. Ako je treći argument 0, imamo jednostavno definiciju A_0 preko sljedbenika drugog argumenta (6.30), a u preostalim slučajevima imamo početni uvjet (6.31), koji je jednak prvom argumentu za zbrajanje, nuli za množenje, te je u svim ostalim slučajevima jednak 1.

$$A(x, y + 1, z + 1) = A(x, A(x, y, z + 1), z) \quad (6.29)$$

$$A(x, y, 0) = y + 1 \quad (6.30)$$

$$A(x, 0, z) = A_{\text{start}}(x, z) := \begin{cases} x, & z = 1 \\ 0, & z = 2 \\ 1, & \text{inače} \end{cases} \quad (6.31)$$

Slučaj kad su i drugi i treći argument 0 tako je pokriven i sa (6.30) i sa (6.31), ali to nije problem, jer je po obje te jednadžbe $A(x, 0, 0) = 1$.

Sada zapišimo taj sustav u obliku opće rekurzije. Prvo skupimo sve te jednadžbe u jedno grananje (koristit ćemo znak $=$ jer imamo korolar 6.14, ali općenito, trebali bismo koristiti \simeq dok još ne znamo je li funkcija totalna):

$$A(x, y, z) = \begin{cases} A(x, A(x, \text{pd}(y), z), \text{pd}(z)), & y > 0 \wedge z > 0 \\ y + 1, & z = 0 \\ A_{\text{start}}(x, z), & \text{inače} \end{cases}, \quad (6.32)$$

a zatim to napišimo u obliku opće rekurzije $\text{comp}_k(x, y, z, e) = G(x, y, z, e)$, gdje je očito $k = 3$, e je traženi indeks, a funkcija G^4 je zadana s

$$G(x, y, z, e) := \begin{cases} \text{comp}_3(x, \text{comp}_3(x, \text{pd}(y), z, e), \text{pd}(z), e), & y \in \mathbb{N}_+ \wedge z \in \mathbb{N}_+ \\ \text{Sc}(y), & z = 0 \\ A_{\text{start}}(x, z), & \text{inače} \end{cases} \quad (6.33)$$

(sad smo morali upotrijebiti znak \simeq , jer G nije totalna — npr. $(0, 1, 1, 0) \notin \mathcal{D}_G$).

Po teoremu 3.61, G je parcijalno rekurzivna. Po teoremu 6.12, postoji prirodni broj a takav da za sve $x, y, z \in \mathbb{N}$ vrijedi $\{a\}(x, y, z) \simeq G(x, y, z, a)$. Tvrdimo da je $\{a\}^3 = A^3$, odnosno da je rješenje jedinstveno.

Točkovno, trebamo dokazati $\{a\}(x, y, z) \simeq A(x, y, z)$ za sve $x, y, z \in \mathbb{N}$, i to činimo ugniježđenom indukcijom: vanjskom po z , unutarnjom po y .

Vanjska baza: za $z = 0$ vrijedi (za sve $x, y \in \mathbb{N}$)

$$\{a\}(x, y, 0) \simeq G(x, y, 0, a) = y + 1 = A(x, y, 0). \quad (6.34)$$

Vanjska pretpostavka: pretpostavimo da za $z = t$, za sve $x, y \in \mathbb{N}$, vrijedi

$$\{a\}(x, y, t) \simeq A(x, y, t). \quad (6.35)$$

Vanjski korak: neka je sada $z = t + 1$. Dokazujemo da za sve $x, y \in \mathbb{N}$ vrijedi $\{a\}(x, y, t + 1) \simeq A(x, y, t + 1)$, unutarnjom indukcijom po y .

Unutarnja baza: za $y = 0$ vrijedi (za sve $x \in \mathbb{N}$)

$$\{a\}(x, 0, t + 1) \simeq G(x, 0, t + 1, a) \simeq Astart(x, t + 1), \quad (6.36)$$

što je u slučaju $t = 0$ jednako $Astart(x, 1) = x = x + 0 = A_1(x, 0) = A(x, 0, 1)$, u slučaju $t = 1$ je jednako $Astart(x, 2) = 0 = x \cdot 0 = A_2(x, 0) = A(x, 0, 2)$, a u svim ostalim slučajevima ($t \geq 2$) je jednako $Astart(x, t + 1) = 1 = A_{t+1}(x, 0) = A(x, 0, t + 1)$. Dakle, uvijek je $\{a\}(x, 0, t + 1) \simeq A(x, 0, t + 1)$, pa je unutarnja baza dokazana.

Unutarnja pretpostavka: pretpostavimo da za $y = s$, za sve $x \in \mathbb{N}$, vrijedi

$$\{a\}(x, s, t + 1) \simeq A(x, s, t + 1). \quad (6.37)$$

Unutarnji korak: neka je sad $y = s + 1$, i $x \in \mathbb{N}$ proizvoljan. Moramo dokazati $\{a\}(x, s + 1, t + 1) \simeq A(x, s + 1, t + 1)$. Krenimo raspisivati s lijeve strane:

$$\begin{aligned} \{a\}(x, s + 1, t + 1) &\simeq G(x, s + 1, t + 1, a) \simeq \\ &\simeq \text{comp}_3(x, \text{comp}_3(x, \text{pd}(s + 1), t + 1, a), \text{pd}(t + 1), a) \simeq \\ &\simeq \text{comp}_3(x, \text{comp}_3(x, s, t + 1, a), t, a) \simeq \text{comp}_3(x, \{a\}(x, s, t + 1), t, a) \simeq \\ &\simeq \text{comp}_3(x, A(x, s, t + 1), t, a) \simeq \{a\}(x, A(x, s, t + 1), t) \simeq \\ &\simeq A(x, A(x, s, t + 1), t) \simeq A(x, s + 1, t + 1). \end{aligned} \quad (6.38)$$

Time su oba koraka provedena, pa tvrdnja vrijedi. To znači da A ima indeks a , pa je parcijalno rekurzivna, odnosno zbog korolara 6.14 rekurzivna funkcija. \square

Upravo provedeni dokaz predstavlja određenu „šablonu” pomoću koje možemo za razne rekurzivno zadane funkcije dokazati da su doista rekurzivne u precizno definiranom smislu (imaju indeks, i totalne su). Ključno je da moramo na neki „vanjski” način zaključiti totalnost, jer inače bismo zapravo mogli zaključiti da je promatrana funkcija primitivno rekurzivna.

6.3. Ekvivalentnost indeksa

Teorem rekurzije može se shvatiti na još jedan način, koji je povijesno bio prvi dokazan. Taj fenomen smo već vidjeli na početku ovog poglavlja. Teorem o parametru (propozicija 6.4) se prirodno može dobiti iz primitivne rekurzivnosti komponiranja (propozicija 6.7), jer po (6.2) možemo staviti

$$S'_k(y, e) := \text{compose}_{k(k+1)}(i_1, \dots, i_k, \overline{C_6}(y), e), \quad (6.39)$$

gdje je $i_n := \langle \text{codeDEC}(n, 3), \text{codeINC}(0), \text{codeGOTO}(0) \rangle$ indeks funkcije I_n^k , te je (primjer 3.28) $\overline{C_6}(y)$ indeks funkcije C_y^k — ali s obzirom na to da je lakše dokazati teorem o parametru, išli smo u suprotnom smjeru.

Tako je i ovdje: jednakost $\{e\}(\vec{x}) \simeq G(\vec{x}, e)$ iz teorema rekurzije možemo shvatiti kao $\{e\} = \text{spec}(e, G)$, odnosno na neki način $e \approx S_k(e, g)$, gdje je g neki indeks za G . Lako se vidi da ta „približna jednakost” ne može doista biti jednakost ni u kojem zanimljivom slučaju, jer za $g \in \text{Prog}$ uvijek vrijedi $S_k(e, g) > e$ (zgodna je vježba dokazati to). Što je onda relacija \approx ?

Ukratko, vrijeme je da malo preciznije formaliziramo ideju ekvivalentnih RAM-programa iz definicije 1.25: ono što bismo htjeli reći nije da su e i $S_k(e, g) =: F(e)$ isti brojevi, već da su $\{e\}^k$ i $\{F(e)\}^k$ iste funkcije. Nažalost, kao i u slučaju specijalizacije, imat ćemo različite relacije $\approx_k, k \in \mathbb{N}_+$ — nećemo ih moći upotrebljivo objediniti u jednu relaciju, jer u RAM-programu nigdje ne piše intendirana mjesnost.

Definicija 6.17. Neka je $k \in \mathbb{N}_+$, te $e, f \in \mathbb{N}$. Kažemo da su e i f k -ekvivalentni, i pišemo $e \approx_k f$, ako vrijedi $\{e\}^k = \{f\}^k$. \triangleleft

Primjer 6.18. Za svaki $k \in \mathbb{N}_+$, za sve $e, f \in \text{Prog}^c$ vrijedi $e \approx_k f$ (jer svi brojevi izvan Prog su indeksi prazne funkcije). Također, za svaki $e \in \text{Prog}$, za sve $j, k \in \mathbb{N}_+$, vrijedi $e * \langle \text{codeINC}(j) \rangle \approx_k e$ (dodavanje instrukcije $\text{INC } \mathcal{R}_j$ na kraj programa neće promijeniti uvjet zaustavljanja, niti će promijeniti izlaznu vrijednost jer je $j > 0$). \triangleleft

Propozicija 6.19. Za svaki $k \in \mathbb{N}_+$, \approx_k je relacija ekvivalencije, s prebrojivo mnogo klasa ekvivalencije koje su sve prebrojive.

Dokaz. Refleksivnost, simetričnost i tranzitivnost su trivijalne: recimo, $e \approx_k f \approx_k g$ znači $\{e\}^k = \{f\}^k = \{g\}^k$, pa je $\{e\}^k = \{g\}^k$, iz čega $e \approx_k g$. Inače, u refleksivnosti se, kroz samu oznaku $\{e\}^k$, krije korolar 1.13 koji kaže da za fiksni k , svaki RAM-program računa jedinstvenu k -mjesnu funkciju.

Prebrojivost broja klasa ekvivalencije jednostavno slijedi iz činjenice da postoji prebrojivo mnogo izračunljivih funkcija (teorem 1.15). Konkretno, za svaki $k \in \mathbb{N}_+$ su sve funkcije $C_y^k, y \in \mathbb{N}$, različite, pa je (primjer 3.28) $\overline{C_6}(y) \not\approx_k \overline{C_6}(z)$ za sve $y \neq z$. Iz toga slijedi da različitih klasa ima beskonačno, a ne može ih biti neprebrojivo

jer su neprazni disjunktne podskupovi od \mathbb{N} (recimo, $\min: \mathbb{N}/\approx_k \rightarrow \mathbb{N}$, koja svakoj klasi ekvivalencije pridružuje njen najmanji element, je injekcija, pa je $\text{card}(\mathbb{N}/\approx_k) \leq \text{card } \mathbb{N} = \aleph_0$).

Prebrojivost svake klase slijedi iz primjera 6.18: neka je $e \in \mathbb{N}$ proizvoljan. Ako je $e \in \text{Prog}^c$, tada je čitav Prog^c u njegovoj klasi. Na primjer, $n \mapsto 2n + 3$ je injekcija s \mathbb{N} u $\text{Seq}^c \subseteq \text{Prog}^c \subseteq [e]_{\approx_k} \subseteq \mathbb{N}$, pa je $[e]_{\approx_k}$ prebrojiva. Ako je pak $e \in \text{Prog}$, tada je $F: \mathbb{N}_+ \rightarrow [e]_{\approx_k}$, zadana s $F(j) := e * \langle \text{codeNC}(j) \rangle$, injekcija — jer j možemo rekonstruirati iz $f := F(j)$ kao $\text{regN}(\text{rpart}(f, 0))$. \square

Možda malo više iznenađujuća činjenica vezana uz familiju relacija $\approx_k, k \in \mathbb{N}_+$ je da je ona *padajuća*: vrijedi $(\approx_1) \supset (\approx_2) \supset (\approx_3) \supset \dots$. Da bismo to dokazali, prvo trebamo jedno tehničko svojstvo funkcija S_k .

Lema 6.20. *Za svaki $e \in \mathbb{N}$ i za svaki $k \in \mathbb{N}_+$ vrijedi $S_k(0, e) = e$.*

Dokaz. Direktnim uvrštavanjem u definiciju. Prvo, ako je $e \in \text{Prog}^c$, tada je po (6.6) $S_k(y, e) = e$ za sve $y \in \mathbb{N}$, pa tako i za $y = 0$. Još treba dokazati tvrdnju za $e \in \text{Prog}$.

Drugo, rastavom na slučajeve vidimo da je $\text{Shift}(0, i) = i$. Recimo, za $i \in \text{InsDEC}$, postoje $j, l < i$ takvi da je $i = \text{codeDEC}(j, l) = \langle 1, j, l \rangle$. No to znači da je $j = i[1] = \text{regN}(i)$ i $l = i[2] = \text{dest}(i)$, pa je $\text{Shift}(0, i) = \text{codeDEC}(j, l + 0) = i$. Ostali slučajevi su slični, zapravo još jednostavniji.

Treće, sada je $H(0, e, t) = \text{Shift}(0, e[t]) = e[t] = \text{part}(e, t)$, pa je desni operand operacije $*$ u (6.6) jednak $\overline{H}(0, e, \text{lh}(e)) = \overline{\text{part}}(e, \text{lh}(e)) = e$ jer iz $\text{Prog}(e)$ slijedi $\text{Seq}(e)$. Lijevi operand iste operacije je $\overline{G}(0) = 1$ jer je svaka povijest u nuli jednaka $\langle \rangle = 1$.

I četvrto, to onda znači da je (za $e \in \text{Prog}$) $S_k(0, e) = 1 * e$, što se lako vidi da je jednako e prema (3.22) i (3.23), jer vrijedi $\text{lh}(1) = 0$ i $\text{Seq}(e)$. \square

Korolar 6.21. *Za sve $k, l \in \mathbb{N}_+$, za sve $\vec{x} \in \mathbb{N}^k$ i za sve $e \in \mathbb{N}$, vrijedi*

$$\{e\}^{k+l}(\vec{x}, 0, 0, \dots, 0) \simeq \{e\}^k(\vec{x}). \quad (6.40)$$

Dokaz. Prvo, indukcijom po l , prateći dokaz korolara 6.5 i koristeći prethodnu lemu u svakom koraku, lako dokažemo $S_{kl}(0, 0, \dots, 0, e) = e$. Iz toga onda slijedi

$$\{e\}^{k+l}(\vec{x}, 0, 0, \dots, 0) \simeq \{S_{kl}(0, 0, \dots, 0, e)\}(\vec{x}) \simeq \{e\}^k(\vec{x}), \quad (6.41)$$

što smo i trebali dokazati. Primijetimo da je na neformalnoj razini ovo očito: za bilo koji RAM-program P , P -izračunavanje s \vec{x} je *isto* (isti niz istih konfiguracija) kao i P -izračunavanje s $(\vec{x}, 0, 0, \dots, 0)$, za bilo koji broj dodanih nula na kraj ulaza — jer je izračunavanje determinističko, a početna konfiguracija $c_0 = (0, \vec{x}, 0, 0, \dots, 0)$ je ista. Samo smo eksplicitno rekli da će još l registara nakon \mathcal{R}_k biti postavljeno na nule, no oni bi ionako bili postavljeni na nule u računanju s \vec{x}^k , jer tamo nisu ulazni. \square

Propozicija 6.22. *Za sve $k, l \in \mathbb{N}_+$ takve da je $k > l$ vrijedi $(\approx_k) \subset (\approx_l)$.*

Dokaz. Označimo $m := k - l \in \mathbb{N}_+$. Za (\subseteq) , neka je $e \approx_k f$. Tada je $\{e\}^k = \{f\}^k$ (a $k = l + m$), pa je prema prethodnom korolaru, za sve $\vec{x} \in \mathbb{N}^l$,

$$\{e\}^l(\vec{x}) \simeq \{e\}^{l+m}(\vec{x}, 0, 0, \dots, 0) \simeq \{f\}^{l+m}(\vec{x}, 0, 0, \dots, 0) \simeq \{f\}^l(\vec{x}). \quad (6.42)$$

Dakle vrijedi i $\{e\}^l = \{f\}^l$, pa je $e \approx_l f$.

Za (\neq) , moramo naći dva RAM-programa koji su ekvivalentni za sve moguće l -torke ulaznih podataka, ali kad počnemo stavljati ulaze i u registre $\mathcal{R}_{l+1} \dots \mathcal{R}_k$, više nisu. Za to nam mogu poslužiti 1 kao dobro poznati indeks nulfunkcije, i i_k kao indeks k -te koordinatne projekcije — pogledajte tekst nakon (6.2) za preciznu definiciju.

Tada svakako vrijedi $1 \approx_l i_k$, jer za svaki $\vec{x} \in \mathbb{N}^l$ vrijedi

$$\{i_k\}^l(\vec{x}) \simeq \{i_k\}^{l+m}(\vec{x}, 0, 0, \dots, 0) = I_k^k(\vec{x}, 0, 0, \dots, 0) = 0 = C_0^l(\vec{x}) = \{1\}^l(\vec{x}); \quad (6.43)$$

ali za $\vec{y} := (0, 0, \dots, 0, 1) \in \mathbb{N}^k$ vrijedi

$$\{i_k\}^k(\vec{y}) = I_k^k(0, 0, \dots, 0, 1) = 1 \neq 0 = C_0^k(\vec{y}) = \{1\}^k(\vec{y}), \quad (6.44)$$

iz čega slijedi $1 \not\approx_k i_k$. □

Kao što rekosmo na početku, indeks e iz teorema rekurzije može se shvatiti kao neka vrsta „fiksne točke” primitivno rekurzivne funkcije F^1 zadane s $F(e) := S_k(e, g)$, gdje je g neki (fiksni) indeks funkcije G čiji poziv stoji na desnoj strani opće rekurzije. Dakle, da imamo teorem koji nam kaže da svaka primitivno rekurzivna jednomjesna funkcija ima fiksnu točku, mogli bismo pomoću njega dokazati teorem rekurzije. Ipak, kako zasad stvari stoje, teorem rekurzije smo već dokazali, pa pokušajmo pomoću njega dokazati teorem o fiksnoj točki.

Prvo, opet, sasvim je jasno da ne možemo tražiti $e = F(e)$ — recimo, S_c je čak inicijalna funkcija, ali ne postoji e takav da je $e = S_c(e)$ (to smo koristili kod Russellove funkcije). Ali pokazat će se da možemo tražiti $e \approx_k F(e)$ za bilo koji unaprijed fiksirani $k \in \mathbb{N}_+$ (e će ovisiti o k).

Drugo, F ne mora biti primitivno rekurzivna — vidjet ćemo da je dovoljno da bude rekurzivna. Tada teorem svakako vrijedi i za primitivno rekurzivne funkcije, zbog korolara 2.33. Samo napomenimo da ne možemo još oslabiti pretpostavku zahtijevajući da je F parcijalno rekurzivna: $\emptyset^1 = \mu\emptyset^2$ je parcijalno rekurzivna, a očito ne postoji $e \in \mathbb{N}$ takav da su e i $\emptyset(e)$ k -ekvivalentni, jer izraz $\emptyset(e)$ nema smisla ni za koji e .

Treće, F mora biti jednomjesna: broj k je mjesnost funkcije $\{e\}^k$, koja je jednaka funkciji $\{F(e)\}^k$. Mjesnost od F mora biti 1 jer želimo u nju uvrstiti $e \in \mathbb{N}^1$.

I četvrto, F mora biti izračunljiva: nije dovoljno da zahtijevamo samo totalnost. Recimo, promotrimo karakterističnu funkciju klase ekvivalencije $\text{Emp} := [0]_{\approx_1}$. Kad

bi postojao broj e takav da je $e \approx_1 \chi_{\text{Emp}}(e)$, tada bi $e \in \text{Emp}$ povlačilo s jedne strane $e \approx_1 0$, a s druge $e \approx_1 \chi_{\text{Emp}}(e) = 1$, pa bi po tranzitivnosti bilo $0 \approx_1 1$, kontradikcija. ($\{0\}(0)$ nema smisla po propoziciji 3.57(2), a $\{1\}(0) = Z(0) = 0$, pa $\{0\}(0) \neq \{1\}(0)$, dakle $\{0\}^1 \neq \{1\}^1$.) No $e \notin \text{Emp}$ je također kontradikcija, jer bi to značilo $e \approx_1 \chi_{\text{Emp}}(e) = 0$, dakle $e \in [0]_{\approx_1} = \text{Emp}$. Dakle, χ_{Emp} nema fiksnu točku, iako je (kao i svaka karakteristična funkcija) totalna.

Primijetite sličnost upravo provedenog razmišljanja s Russellovim paradoksom. Primijetite također da će iz toga slijediti, jednom kad dokažemo teorem o fiksnoj točki, da skup Emp nije rekurzivan. No zapravo će to biti samo jedna jednostavna posljedica Riceovog teorema, koji ćemo dokazati kasnije.

Lema 6.23 (Teorem o fiksnoj točki). *Neka je $k \in \mathbb{N}_+$, te F^1 rekurzivna funkcija.*

Tada postoji $e \in \mathbb{N}$ takav da je $e \approx_k F(e)$.

Dokaz. Jednostavno, zapišimo traženi uvjet pomoću univerzalne funkcije:

$$\text{comp}_k(\vec{x}, e) \simeq \text{comp}_k(\vec{x}, F(e)), \text{ za sve } \vec{x} \in \mathbb{N}^k. \quad (6.45)$$

Pa to je opća rekurzija! Na desnoj strani je G^{k+1} zadana s $G(\vec{x}, e) := \text{comp}_k(\vec{x}, F(e))$, dakle dobivena kompozicijom iz parcijalno rekurzivne comp_k , rekurzivne F i inicijalnih koordinatnih projekcija, pa je parcijalno rekurzivna. Po teoremu 6.12, postoji $e \in \mathbb{N}$ koji zadovoljava tu funkcijsku jednadžbu. Jer je F rekurzivna, dakle totalna, postoji i $f := F(e) \in \mathbb{N}$. Sada se jednadžba može zapisati kao $\{e\}(\vec{x}) \simeq \{f\}(\vec{x})$ za sve $\vec{x} \in \mathbb{N}^k$, odnosno $\{e\}^k = \{f\}^k$, dakle $e \approx_k f = F(e)$, što smo i trebali. \square

6.4. Invarijantnost

Vidjeli smo u prethodnoj točki da iz teorema o fiksnoj točki zapravo možemo zaključiti da klasa ekvivalencije $\text{Emp} = [0]_{\approx_1}$ nije rekurzivna: njena karakteristična funkcija ne može imati fiksnu točku, jer to vodi na paradoks vrlo sličan Russellovom. Međutim, taj rezultat je „kap u moru” općenitog rezultata koji kaže da *nijedna* klasa ekvivalencije nijedne relacije \approx_k nije rekurzivna — štoviše, nijedna *unija* takvih klasâ ekvivalencije nije rekurzivna, osim trivijalnih unija $\bigcup \emptyset = \emptyset$ (unija nijedne klase) i $\bigcup (\mathbb{N}/\approx_k) = \mathbb{N}$ (unija svih klasa).

O čemu se tu zapravo radi? Reći da je neki broj e element klase Emp , zapravo znači reći da je $e \approx_1 0$, odnosno $\{e\}^1 = \{0\}^1 = \emptyset^1$. Drugim riječima, Emp je upravo skup svih indeksa prazne jednomjesne funkcije. Ako uzmemo neku drugu izračunljivu funkciju, dobit ćemo neku drugu klasu (kojoj pripada njen indeks), i obrnuto, neka druga klasa $[e]_{\approx_k}$ će biti skup svih indeksa funkcije $\{e\}^k$.

Neka unija klasa $S := \bigcup_{e \in A} [e]_{\approx_k}$ tada će odgovarati nekom *skupu* izračunljivih k -mjesnih funkcija $\mathcal{F} := \{\{e\}^k \mid e \in A\} \subseteq \text{Comp}_k$, te će praznom skupu S odgovarati

prazan skup \mathcal{F} , a skupu $S = \mathbb{N}$ odgovarat će $\mathcal{F} = \text{Comp}_k$. Kako je to preslikavanje injekcija, ostalim skupovima brojeva ($\emptyset \subset S \subset \mathbb{N}$) odgovarat će ostali skupovi funkcija ($\emptyset \subset \mathcal{F} \subset \text{Comp}_k$). Vrijeme je da to formaliziramo.

Definicija 6.24. Neka je $k \in \mathbb{N}_+$. Za svaki $\mathcal{F} \subseteq \text{Comp}_k$ definiramo *skup indeksa* (ili *skup k-indeksa*, ako želimo naglasiti k) kao

$$\|\mathcal{F}\| := \{e \in \mathbb{N} \mid \{e\}^k \in \mathcal{F}\}. \quad (6.46)$$

(U oznaci nigdje ne spominjemo k, jer se može rekonstruirati iz \mathcal{F} ako je \mathcal{F} neprazan. Naravno, $\|\emptyset\| = \emptyset$, bez obzira na k.) \triangleleft

Primijetite sličnost s definicijom (4.102) — kao što smo tamo kodirali riječi zapisom u bazi, ovdje kodiramo funkcije njihovim indeksima. Kao što smo već rekli, relacija $\text{index} \subseteq \mathbb{N} \times \mathbb{N}_+ \times \text{Comp}$, zadana s $\text{index}(e, k, F) \iff \{e\}^k = F$, nema funkcijsko svojstvo po prvoj varijabli, odnosno ne možemo govoriti o jedinstvenom indeksu neke konkretne funkcije — ali zato možemo govoriti o skupu indeksa nekog skupa funkcija.

Lema 6.25. Za svaki $k \in \mathbb{N}_+$, preslikavanje $\|\cdot\|: \mathcal{P}(\text{Comp}_k) \rightarrow \mathcal{P}(\mathbb{N})$ je injekcija.

Štoviše, iz dokaza će slijediti da $\|\cdot\|$ strogo raste: $\mathcal{F} \subset \mathcal{G}$ povlači $\|\mathcal{F}\| \subset \|\mathcal{G}\|$.

Dokaz. Neka su $\mathcal{F}, \mathcal{G} \subseteq \text{Comp}_k$ različiti. Tada postoji funkcija F takva da (bez smanjenja općenitosti) je $F \in \mathcal{G}$, ali $F \notin \mathcal{F}$. Tada $F \in \mathcal{G} \subseteq \text{Comp}_k$ znači da je F RAM-izračunljiva, pa postoji RAM-program P koji je računa. Tada je $e := \lceil P \rceil$ indeks funkcije F , odnosno $\{e\}^k \in \mathcal{G}$, pa je $e \in \|\mathcal{G}\|$.

No kad bi bilo i $e \in \|\mathcal{F}\|$, to bi značilo $\{e\}^k \in \mathcal{F}$, što je nemoguće jer $F \notin \mathcal{F}$. Drugim riječima, $e \in \|\mathcal{G}\| \setminus \|\mathcal{F}\|$, odnosno $\|\mathcal{F}\| \neq \|\mathcal{G}\|$. \square

Na neki način, do na nezgodnu činjenicu da je $\mathcal{P}(\text{Comp}_k)$ neprebrojiv, čini se kao da imamo „kodiranje” skupova izračunljivih funkcija. Onda bismo mogli reći, po uzoru na točku 4.4.1, da \mathcal{F} ima neko svojstvo izračunljivosti ako karakteristična funkcija $\chi_{\|\mathcal{F}\|}$ ima to svojstvo. Ili, neko *svojstvo* \wp izračunljivih k-mjesnih funkcija je *odlučivo* ako je skup $\|\{F \in \text{Comp}_k \mid \wp(F)\}\|$ rekurzivan. Ali zbog Riceovog teorema, takva definicija bi bila sasvim beskorisna: *jedini* rekurzivni skupovi indeksa su \emptyset i \mathbb{N} , odnosno jedina odlučiva svojstva izračunljivih funkcija su trivijalna svojstva \perp i \top .

Kako je to moguće? Pa \mathbb{N} ima hrpu rekurzivnih podskupova. Uzmimo recimo skup parnih brojeva $2\mathbb{N}$. Nije li to skup indeksa nekog netrivialnog skupa funkcija? Nije, jer se može vidjeti da *svaka* funkcija ima (neki) parni indeks: prazna funkcija ima indeks $0 \in 2\mathbb{N}$, nulfunkcija ima indeks $\lceil [0. \text{INC } \mathcal{R}_1] \rceil = \langle \text{codeINC}(1) \rangle = 2^{19} \in 2\mathbb{N}$, a sve ostale RAM-izračunljive funkcije imaju *samo* parne indekse, jer ih računaju neprazni RAM-programi (I_0, I_1, \dots) čiji kodovi su sigurno djeljivi s $2^{\lceil I_0 \rceil + 1}$. Drugim riječima, ovo svojstvo izračunljivih funkcija je trivijalno svojstvo \top .

Dobro, uzmimo onda neki manji skup, recimo jednočlan skup $\{1\}$. On je svakako rekurzivan (lema 2.45), ali opet nije skup indeksa. Recimo, za $k = 1$, u odgovarajućem skupu \mathcal{F} bila bi funkcija \mathbf{Z} , ali skup svih njenih indeksa je daleko veći od $\{1\}$ — iz primjera 6.18 vidimo da je beskonačan. Čim smo stavili 1 unutra, morali smo staviti i čitavu klasu $\|\{\mathbf{Z}\}\| = [1]_{\approx_1}$.

Dakle, skupovi indeksa nisu bilo kakvi skupovi prirodnih brojeva. Oni moraju imati specijalno svojstvo *invarijantnosti* na relaciju \approx_k : ako sadrže e , tada moraju sadržavati i sve f takve da je $e \approx_k f$. Na neki način, u takvom skupu se nalaze (kodirani) RAM-programi, ali skup je „neovisan o implementaciji” konkretnog algoritma. Recimo, mogli bismo zamisliti funkciju $\text{sort}^1: \text{Seq} \rightarrow \text{Seq}$ koja sortira konačne nizove, zadane kodovima. Na primjer (*unit test*), $\text{sort}(\langle 2, 9, 0, 2 \rangle) = \langle 0, 2, 2, 9 \rangle$. Nije preteško pokazati da je sort parcijalno rekurzivna, ali pritom moramo odabrati koji algoritam za sortiranje ćemo koristiti. Recimo, *selection sort* bi izgledao ovako nekako:

$$\text{tail}(s) := \mu t(\langle s[0] \rangle * t = s) \quad (6.47)$$

$$\text{min}(s) := \mu n(\exists i < \text{lh}(s))(s[i] = n) \quad (6.48)$$

$$\text{up}(s, x) \simeq \begin{cases} \langle \rangle, & s = \langle \rangle \\ \text{up}(\text{tail}(s), x), & s[0] < x \\ \langle s[0] \rangle * \text{up}(\text{tail}(s), x), & s[0] \geq x \end{cases} \quad (6.49)$$

$$\text{sort}(s) \simeq \begin{cases} \langle \rangle, & s = \langle \rangle \\ \langle \text{min}(s) \rangle * \text{sort}(\text{up}(s, \text{min}(s))), & \text{Seq}'(s) \end{cases} \quad (6.50)$$

(funkcije up i sort definirane su općim rekurzijama), dok bi *quicksort* bio nešto poput

$$\text{dn}(s, x) \simeq \begin{cases} \langle \rangle, & s = \langle \rangle \\ \text{dn}(\text{tail}(s), x), & s[0] \geq x \\ \langle s[0] \rangle * \text{dn}(\text{tail}(s), x), & s[0] < x \end{cases} \quad (6.51)$$

$$\text{sort}(s) \simeq \begin{cases} \langle \rangle, & s = \langle \rangle \\ \text{sort}(\text{dn}(\text{tail}(s), s[0])) * \langle s[0] \rangle * \text{sort}(\text{up}(\text{tail}(s), s[0])), & \text{Seq}'(s) \end{cases} \quad (6.52)$$

— što će nakon kompajliranja (i korištenja teorema rekurzije na nekoliko mjesta) rezultirati jako različitim RAM-programima, odnosno indeksima funkcije sort . Ako sa ss označimo indeks za *selection sort*, a sa qs označimo indeks za *quicksort*, očito vrijedi $ss \approx_1 qs$, jer što se specifikacije tiče, i jedan i drugi računaju istu funkciju: sortiraju konačan niz.

Možete se zabaviti pišući razne druge algoritme za sortiranje u jeziku parcijalno rekurzivnih funkcija ... dobit ćete raznorazne implementacije, odnosno indekse, za jednu te istu matematičku funkciju, i svi su oni u istoj klasi ekvivalencije $\|\{\text{sort}\}\|$. Ako

bismo prije navedeni *unit test* htjeli napisati u obliku jednadžbe po indeksu,

$$\{e\}(\langle 2, 9, 0, 2 \rangle) = \langle 0, 2, 2, 9 \rangle, \quad (6.53)$$

tada je jasno da i ss i qs , i svi drugi indeksi iz $\|\{sort\}\|$, moraju zadovoljavati tu jednadžbu. Iako taj *unit test* ni izdaleka nije dovoljan za specifikaciju funkcije *sort*, svejedno ga možemo gledati kao neko svojstvo koje izračunljive funkcije mogu a i ne moraju imati: $\wp(F) : \iff F(810\,152\,280) = 272\,386\,847\,250$. Važno je naglasiti da to svojstvo ne ovisi o implementaciji funkcije F , već samo o funkciji samoj.

Definicija 6.26. Neka je $k \in \mathbb{N}_+$. Za skup $S \subseteq \mathbb{N}$ kažemo da je *k-invarijantan* ako i samo ako za sve $e, f \in \mathbb{N}$, iz $e \in S$ i $e \approx_k f$ slijedi $f \in S$. \triangleleft

Svojstvo *k-invarijantnosti* pruža brojevnu karakterizaciju skupova indeksa, bez pozivanja na skupove funkcija.

Lema 6.27. Neka je $k \in \mathbb{N}_+$, te $S \subseteq \mathbb{N}$.

Tada je S *k-invarijantan* ako i samo ako je $S = \|\mathcal{F}\|$ za neki $\mathcal{F} \subseteq \text{Comp}_k$.

Dokaz. Za smjer (\Leftarrow), neka je $f \approx_k e \in S = \|\mathcal{F}\|$. Tada je po definiciji $\{f\}^k = \{e\}^k \in \mathcal{F}$, pa je i $f \in \|\mathcal{F}\| = S$. Za smjer (\Rightarrow), pretpostavimo da je S *k-invarijantan*, i tražimo \mathcal{F} . Očiti kandidat je $\mathcal{F} := \{\{e\}^k \mid e \in S\}$. Dakle, trebamo dokazati da je $\|\mathcal{F}\| = S$.

Za inkluziju (\supseteq), po definiciji \mathcal{F} iz $e \in S$ slijedi $\{e\}^k \in \mathcal{F}$, dakle $e \in \|\mathcal{F}\|$.

Za inkluziju (\subseteq), iz $f \in \|\mathcal{F}\|$ slijedi da postoji $F^k \in \mathcal{F}$ takva da je $\{f\}^k = F$. No $F \in \mathcal{F}$ po definiciji \mathcal{F} znači da postoji $e \in S$ takav da je $\{e\}^k = F$. Sada $\{e\}^k = F = \{f\}^k$ znači $e \approx_k f$, pa jer je S *k-invarijantan* i $e \in S$, zaključujemo $f \in S$, što smo trebali. \square

Još je jedna stvar tu na prvi pogled čudna: zašto svojstvo (6.53) nije parcijalno rekurzivno? Čini se da je njegova karakteristična funkcija dobivena kompozicijom iz primitivno rekurzivne funkcije $\chi_$, parcijalno rekurzivne funkcije comp_1 , te konstanti $C_{\langle 2, 9, 0, 2 \rangle}^1$ i $C_{\langle 0, 2, 2, 9 \rangle}^1$. Ipak, to nije istina, jer prema marljivoj evaluaciji, ta kompozicija nikako ne može biti totalna (jer comp_1 nije totalna), dok bi karakteristična funkcija morala biti totalna. Isti problem smo već imali prije — pogledajte napomenu 3.40.

6.4.1. Riceov teorem

Teorem 6.28 (Riceov teorem). Neka je $k \in \mathbb{N}_+$, i $S \subseteq \mathbb{N}$ *rekurzivan k-invarijantan skup*. Tada je $S = \emptyset$ ili $S = \mathbb{N}$.

Ideja dokaza je vrlo slična onom što smo napravili za Emp — samo, dok nam je tamo Russellov paradoks bio „serviran”, ovdje ćemo morati namjestiti scenu za njega.

Dokaz. Pretpostavimo da je S rekurzivan, i da S nije ni \emptyset ni \mathbb{N} . $S \neq \emptyset$ znači da postoji broj $s \in S$, a $S \neq \mathbb{N}$ znači da postoji $n \in S^c = \mathbb{N} \setminus S$. Tada je funkcija F^1 , zadana s

$$F(x) := \begin{cases} n, & x \in S \\ s, & \text{inače} \end{cases}, \quad (6.54)$$

rekurzivna. To je jednostavna primjena teorema 2.44 — simbolički, $F = \{S: C_n^1, C_s^1\}$.

Jer je F rekurzivna, ima fiksnu točku: postoji broj $e \in \mathbb{N}$ takav da je $e \approx_k F(e)$. No to je nemoguće: ako je $e \in S$, tada je po k -invarijantnosti i $F(e) \in S$ — što je u kontradikciji s činjenicom da je $F(e) = n$ za $e \in S$.

Ako pak $e \notin S$, tada je $F(e) = s \in S$, pa zbog simetričnosti relacije \approx_k imamo i $F(e) \approx_k e$. No to bi značilo da S nije k -invarijantan, jer smo našli $s \in S$ takav da je $s \approx_k e$, ali e nije u S .

U oba slučaja došli smo do kontradikcije, pa pod pretpostavkom da je S rekurzivan, jedino je moguće da je $S = \emptyset$ ili $S = \mathbb{N}$. \square

Riceov teorem smo izrekli u „pozitivnom” obliku — no on se češće koristi u „negativnom” obliku.

Korolar 6.29. *Neka je $S \subset \mathbb{N}$ neprazan skup koji je k -invarijantan za neki $k \in \mathbb{N}_+$. Tada S nije rekurzivan.*

Dokaz. Ovo je samo obrat po kontrapoziciji teorema 6.28. \square

Korolar 6.30. *Neka je $k \in \mathbb{N}_+$, i $\emptyset \subset \mathcal{F} \subset \text{Comp}_k$. Tada \mathcal{F} nije odlučiv.*

Dokaz. Označimo $S := \|\mathcal{F}\|$. Prema lemi 6.27, skup S je k -invarijantan. S druge strane, po lemi 6.25 je $S \neq \|\emptyset\| = \emptyset$ i $S \neq \|\text{Comp}_k\| = \mathbb{N}$ (svaki prirodni broj je indeks neke k -mjesne funkcije). Po prethodnom korolaru, S nije rekurzivan, odnosno \mathcal{F} nije odlučiv. \square

Korolar 6.31. *Neka je $k \in \mathbb{N}_+$, te neka je \wp bilo koje netrivialno svojstvo k -mjesnih izračunljivih funkcija. Tada nijedan algoritam ne može točno odrediti, za proizvoljnu $F \in \text{Comp}_k$, ima li F svojstvo \wp .*

Dokaz. Budući da je \wp netrivialno svojstvo, postoji k -mjesna funkcija koja ga ima i neka druga k -mjesna funkcija koja ga nema. To znači da za skup $\mathcal{F} := \{F \in \text{Comp}_k \mid \wp(F)\}$ vrijedi $\emptyset \subset \mathcal{F} \subset \text{Comp}_k$. Kad bi takav algoritam postojao, morao bi primati funkciju u nekom obliku (točkovna definicija, simbolička definicija, RAM-program, Turingov stroj, ...) — a svaki od njih znamo, neformalnim algoritmom, pretvoriti u indeks. Štoviše, imamo i (neformalne) algoritme pretvorbe u suprotnom smjeru, što znači da algoritam kojim bismo htjeli odlučiti \wp mora raditi za sve indekse traženih funkcija. No po Church–Turingovoj tezi, to znači da bi taj algoritam računao $\chi_{\|\mathcal{F}\|}$, što je u kontradikciji s korolarom 6.30. \square

6.4.2. Sintaksna i semantička svojstva RAM-programa

Vezano uz zadani RAM-program možemo postaviti mnoga zanimljiva pitanja: ima li više od milijun instrukcija, stane li s ulazom $(2, 5)$, je li mu registar \mathcal{R}_{15} relevantan, sadrži li instrukciju tipa `GO TO`, računa li totalnu funkciju, dekrementira li ikad u toku izračunavanja registar \mathcal{R}_7 , je li dobiven kompajliranjem simboličke definicije neke primitivno rekurzivne funkcije, zapiše li Turingov stroj dobiven njegovim transpiliranjem ikad prazninu na traku, može li se dobiti spljoštenjem makro-programa koji sadrži barem jedan funkcijski makro, napravi li paran broj koraka prije zaustavljanja s ulazom 27, i tako dalje.

Mnoga od tih svojstava (ali ne nužno sva) prirodno svrstavamo u dvije grupe: nazovimo ih *sintaksna* i *semantička* svojstva. Sintaksna svojstva su ona vezana uz konkretni „programski jezik” kojim je RAM-program pisan: govore o instrukcijama, registrima, ili o raznim sintaksnim postupcima pretvorbe (npr. iz makro-programa, ili u Turingov stroj), te o sintaksnim svojstvima tih drugih nositelja izračunavanja.

Semantička svojstva su ona vezana uz RAM-izračunavanje: govore o zaustavljanju, totalnosti, funkcijama koje se računaju, njihovim domenama, slikama, grafovima, i sličnom. Važno okvirno pravilo koje pruža dobru intuiciju je: **sintaksna svojstva su uglavnom odlučiva, semantička svojstva su uglavnom neodlučiva.**

Dobru empirijsku potvrdu prvog dijela pravila vidjeli smo u poglavlju 3, gdje smo hrpu sintaksnih svojstava pokazali odlučivima, i razvili alate koji nam omogućavaju za još veću hrpu to pokazati bez puno muke. Recimo, \mathcal{R}_{15} je relevantan za P ako i samo ako vrijedi $(\exists i < lh(e))(\text{regn}(e[i]) \geq 15)$ (gdje je $e := \ulcorner P \urcorner$), što je primitivno rekurzivno svojstvo od e .

S druge strane, Riceov teorem pruža dobar uvid u drugi dio tog pravila. Među semantičkim svojstvima izdvajaju se ona koja govore *samo* o računatoj funkciji, bez ikakvog spominjanja konkretne implementacije. Recimo, „ P -izračunavanje s $(2, 5)$ stane” se može tako zapisati, jer zapravo kaže $(2, 5) \in \mathcal{D}_F$, gdje je F funkcija koju P računa. Riceov teorem kaže da su sva takva svojstva sigurno neodlučiva — osim dva trivijalna, naravno. Primjerice, svojstvo „ovaj RAM-program rješava *halting problem*” jest odlučivo: za svaki RAM-program vratimo *false*. Također, „ovaj RAM-program računa parcijalno rekurzivnu funkciju” jest odlučivo: uvijek vratimo *true*.

S druge strane, parnost broja koraka ne možemo tako napisati. Najlakši način da se to vidi je invarijantnost: za svaki RAM-program koji napravi paran broj koraka prije zaustavljanja s ulazom 27, postoji RAM-program koji računa istu funkciju, ali napravi neparan broj koraka prije zaustavljanja s ulazom 27: standardni trik dodavanja instrukcije `INC \mathcal{R}_1` na kraj programa, viđen u primjeru 6.18, „upalit” će i ovdje.

Svakako, postoje i mnoga „hibridna” svojstva, i za utvrđivanje njihove odlučivosti je potrebna detaljna analiza — ali svejedno, Riceov teorem s jedne i Church–Turingova teza s druge strane pružaju dobar dio odgovora na takva pitanja.

Iako Riceov teorem govori samo o skupovima indeksa (dakle invarijantnim podskupovima od \mathbb{N}), svođenjem možemo dokazati i razne druge neodlučivosti odnosno nerekurzivnosti: sjetite se korolara 5.15. Neko sasvim generalno uputstvo za korištenje Riceovog teorema moglo bi se napisati u sljedećem obliku:

1. Utvrdimo da trebamo dokazati da neki skup S nije rekurzivan.
2. Pomoću skupa S nađemo skup \mathcal{F} u kojem se nalaze izračunljive funkcije neke fiksne mjesnosti k . Skup svih indeksa svih funkcija iz \mathcal{F} označimo s $T := \|\mathcal{F}\|$.
3. Nađemo neki element od T (najčešće tako da nađemo neku jednostavnu funkciju $F \in \mathcal{F}$, napišemo RAM-program koji je računa, i nađemo njegov kod).
4. Analogno, nađemo neki element od T^c . Skupa s korakom 3 imamo $\emptyset \subset T \subset \mathbb{N}$.
5. Dokažemo da je T k -invarijantan, gdje je k mjesnost koju smo fiksirali u koraku 2.
6. Formalno svedemo $T \preceq S$, pišući χ_T kao kompoziciju χ_S s nekim rekurzivnim funkcijama.
7. Zaključimo da kad bi S bio rekurzivan, bio bi takav i T , što je kontradikcija s Riceovim teoremom.

Recimo, za dokaz da $\text{Halt}_3^4 = \{(x, y, z, e) \mid (x, y, z) \in \mathcal{D}_{\{e\}^3}\}$ nije rekurzivna relacija:

- fiksiramo (x, y, z) , recimo na $(0, 0, 0)$ — dakle $\mathcal{F} := \{F \in \text{Comp}_3 \mid (0, 0, 0) \in \mathcal{D}_F\}$, odnosno $T := \{e \in \mathbb{N} \mid (0, 0, 0) \in \mathcal{D}_{\{e\}^3}\}$;
- nađemo funkcije $C_0^3 \in \mathcal{F}$ i $\emptyset^3 \notin \mathcal{F}$, iz čega dobivamo $1 \in T \wedge 0 \notin T$;
- dokažemo 3-invarijantnost: $T \ni e \approx_3 f$ povlači $(0, 0, 0) \in \mathcal{D}_{\{e\}^3} = \mathcal{D}_{\{f\}^3}$, pa $f \in T$;
- zaključimo da vrijedi $e \in T$ ako i samo ako je $(0, 0, 0, e) \in \text{Halt}_3$ — dakle $\chi_T = \chi_{\text{Halt}_3} \circ (Z, Z, Z, I_1^1)$, odnosno $T \preceq \text{Halt}_3$. (Raspišite detalje!)

Poglavlje 7.

Rekurzivna prebrojivost

Vidjeli smo već nekoliko puta da domena izračunljive funkcije ne mora biti izračunljiva, i čak smo stekli neki uvid u razlog toga: radi izbjegavanja Russellovog paradoksa moramo dopustiti i parcijalne izračunljive funkcije, ali relacije (kao što je domena funkcije) smatramo izračunljivima preko njihovih karakterističnih funkcija, koje moraju biti totalne. Zato su rekurzivne relacije „jače” u smislu izražajnosti od parcijalno rekurzivnih funkcija, i zapravo su jednako jake kao i *rekurzivne* funkcije. Na algoritamskoj razini (relacije kao karakteristične funkcije) to je istina po definiciji, a na skupovnoj razini (funkcije kao relacije s funkcijskim svojstvom), to slijedi iz teorema 3.44 (teorem o grafu za totalne funkcije).

Pokušajmo ustanoviti što bi bio pandan parcijalnim funkcijama za relacije. Tražiti „parcijalnu karakterističnu funkciju” kao funkciju iz \mathbb{N} u bool je vjerojatno preslabo: korisnost parcijalno rekurzivnih funkcija je u tome što iako nisu definirane svuda, tamo gdje jesu definirane mogu poprimiti proizvoljne vrijednosti. Kad bismo čak i u slučaju da je funkcija definirana mogli dobiti samo jedan bit informacije, imali bismo tri moguća ishoda (*true*, *false* i „ne znam”) koji bi odgovarali četirima mogućim situacijama (u slučaju izostanka izlaza, ulaz i dalje može biti ili ne biti u skupu). Odnosno, parcijalna karakteristična funkcija nije jednoznačno određena skupom.

Možemo li postići jednoznačnost, a da i dalje dozvolimo parcijalnost? Odgovor se nameće sam po sebi: specificiramo da ulaz jest/nije u skupu upravo ako algoritam stane/ne stane s tim ulazom. Sam izlaz nam onda uopće nije bitan, i možemo zamišljati da je *true*, iako ćemo vidjeti u jednom trenutku da će biti korisno pretpostaviti da je zapravo 0. Ono što jest bitno je da smo na taj način specificirali *poluodlučive* probleme: ako je odgovor na pitanje potvrđan, algoritam će nam ga dati (čim stane), ali ako je odgovor negativan, nećemo to nikada saznati izvršavajući algoritam, upravo jer neće nikada stati. Upravo napisanu ideju možemo formalizirati koristeći činjenicu da smo uvijek stajanje algoritma koji računa funkciju reprezentirali kroz domenom te funkcije — pogledajte definicije 1.10 i 4.5, te dijagram (5.3).

Definicija 7.1. Neka je $k \in \mathbb{N}_+$. Za brojevenu relaciju R^k kažemo da je *rekurzivno prebrojiva* ako postoji $F \in \text{Comp}_k$ takva da je $R = \mathcal{D}_F$. \triangleleft

Za početak pokažimo da intuicija „poluodlučivosti” doista smješta rekurzivno prebrojive relacije „između” odlučivih (rekurzivnih) i općenitih relacija, odnosno poklapa se s intuicijom „težine problema” koju smo hvatali relacijom \preceq . Od velike važnosti bit će formula (2.2) za domen u kompozicije.

Propozicija 7.2. *Svaka rekurzivna relacija je rekurzivno prebrojiva.*

Ideja je jednostavna: ako imamo rekurzivnu karakterističnu funkciju, samo trebamo „zaboraviti” vrijednosti 0 (ukloniti tu prasiliku iz domene), i ostaviti samo vrijednosti 1. Drugim riječima, trebamo restringirati χ_R na R .

Dokaz. Neka je $k \in \mathbb{N}$, i R^k rekurzivna relacija. Tada je prema korolaru 3.63, $\chi_R|_R$ parcijalno rekurzivna, te je očito $\mathcal{D}_{\chi_R|_R} = \mathcal{D}_{\chi_R} \cap R = \mathbb{N}^k \cap R = R$. \square

No kad već pričamo o restrikciji, funkcija koju restringiramo je nebitna: s bilo kojom rekurzivnom funkcijom dobit ćemo istu domen u. Ponekad je zgodno fiksirati njene vrijednosti na 0 — usporedite s napomenom 5.4.

Korolar 7.3. *Relacija je rekurzivno prebrojiva ako i samo ako je domena neke parcijalno rekurzivne funkcije koja ne poprima pozitivne vrijednosti.*

(Ne možemo reći „poprima samo vrijednost 0”, jer prazna relacija je domena samo prazne funkcije, koja ne poprima nikakve vrijednosti.)

Dokaz. Smjer (\Leftarrow) je trivijalan. Za smjer (\Rightarrow), neka je $R = \mathcal{D}_F$ rekurzivno prebrojiva. Prema formuli za domen u kompozicije, $\mathcal{D}_{Z \circ F} = \{\vec{x} \in \mathcal{D}_F = R \mid F(\vec{x}) \in \mathcal{D}_Z = \mathbb{N}\} = R$, a funkcija $Z \circ F$ je očito parcijalno rekurzivna i ne poprima pozitivne vrijednosti. \square

Lema 7.4. *Neka su $k, l \in \mathbb{N}_+$, te R^k i P^l relacije takve da je $R \preceq P$. Ako je P rekurzivno prebrojiva, tada je i R takva.*

Dokaz. Po definiciji, $P = \mathcal{D}_H$ za neku parcijalno rekurzivnu funkciju H^l , te postoje rekurzivne funkcije G_1^k, \dots, G_l^k takve da je $\chi_R = \chi_P \circ (G_1, \dots, G_l)$. Sada označimo $F := H \circ (G_1, \dots, G_l)$ — to je parcijalno rekurzivna funkcija kao kompozicija parcijalno rekurzivnih. No kako su sve G_i rekurzivne, dakle totalne, formula (2.2) daje

$$\begin{aligned} \vec{x} \in \mathcal{D}_F &\iff \vec{x} \in \bigcap_{i=1}^l \mathcal{D}_{G_i} \wedge (G_1(\vec{x}), \dots, G_l(\vec{x})) \in \mathcal{D}_H = P \iff \\ &\iff \vec{x} \in \bigcap_{i=1}^l \mathbb{N}^k \wedge \chi_P(G_1(\vec{x}), \dots, G_l(\vec{x})) = 1 \iff \\ &\iff \vec{x} \in \mathbb{N}^k \wedge (\chi_P \circ (G_1, \dots, G_l))(\vec{x}) = 1 \iff \chi_R(\vec{x}) = 1 \iff \vec{x} \in R, \end{aligned} \quad (7.1)$$

dakle po aksiomu ekstenzionalnosti, $R = \mathcal{D}_F$ je domena parcijalno rekurzivne funkcije, odnosno rekurzivno prebrojiva relacija. \square

Naravno, obrat propozicije 7.2 ne vrijedi — kanonski kontraprimjer je Russellov skup kao domena (parcijalno rekurzivne, korolar 5.9) Russellove funkcije, koji nije rekurzivan po teoremu 5.11.

Dakle, skup rekurzivnih relacija je pravi podskup skupa rekurzivno prebrojivih relacija, koji je pak pravi podskup skupa svih relacija — što se može vidjeti kardinalnim argumentom, kao u korolaru 1.16. Svakoј parcijalno rekurzivnoj funkciji odgovara jedinstvena domena, a svakom prirodnom broju e odgovara jedinstvena parcijalno rekurzivna k -mjesna funkcija $\{e\}^k$, dakle rekurzivno prebrojivih relacija ima prebrojivo mnogo — dok svih brojevnih relacija ima $\text{card}(\bigcup_{k \in \mathbb{N}_+} \mathcal{P}(\mathbb{N}^k)) = \aleph_0 \cdot 2^{\aleph_0} = \mathfrak{c} > \aleph_0$.

Kako se u formuli (2.2) nalazi presjek l domenâ, kompozicijom s nekom totalnom „vanjskom” l -mjesnom funkcijom možemo lako dobiti zatvorenost na presjeke. Svejedno je koju vanjsku funkciju odaberemo (može biti i inicijalna l_1^l), ali simetrije radi uzmimo add^l .

Propozicija 7.5. *Neka su $k, l \in \mathbb{N}_+$, te $R_1^k, R_2^k, \dots, R_l^k$ rekurzivno prebrojive relacije, sve iste mjesnosti. Tada je i $\bigcap_{i=1}^l R_i$ također rekurzivno prebrojiva relacija.*

Dokaz. Po definiciji, postoje parcijalno rekurzivne funkcije $G_1^k, G_2^k, \dots, G_l^k$ takve da je za svaki $i \in [1..l]$, $R_i = \mathcal{D}_{G_i}$. Funkcija $F := G_1 + \dots + G_l = \text{add}^l \circ (G_1, \dots, G_l)$ je parcijalno rekurzivna kao kompozicija parcijalno rekurzivnih, a njena domena je

$$\mathcal{D}_F = \left\{ \vec{x} \in \bigcap_{i=1}^l \mathcal{D}_{G_i} \mid (G_1(\vec{x}), \dots, G_l(\vec{x})) \in \mathcal{D}_{\text{add}^l} = \mathbb{N}^l \right\} = \bigcap_{i=1}^l \mathcal{D}_{G_i} = \bigcap_{i=1}^l R_i \quad (7.2)$$

po formuli (2.2), dakle to je rekurzivno prebrojiv skup. \square

Primijetimo da zahtjev da su sve relacije iste mjesnosti zapravo nije bitan: ako nisu, presjek je prazan (jer su \mathbb{N}^k i \mathbb{N}^l disjunktni za $k \neq l$), pa je sigurno rekurzivno prebrojiv kao domena prazne funkcije. Ista napomena vrijedi za sljedeći korolar (restrikcija $f^k|_{R_l}$ je prazna funkcija za $k \neq l$). Ali nastaviti ćemo promatrati prazne relacije i funkcije odvojeno po različitim mjesnostima, kao što smo i dosada činili.

Sličnim trikom možemo pokazati da restringiranjem na rekurzivno prebrojiv skup čuvamo izračunljivost — samo tada moramo paziti i na vrijednosti funkcije na čiju domenu restringiramo. Ideju smo vidjeli u napomeni 3.60.

Korolar 7.6. *Neka je $k \in \mathbb{N}_+$, F^k parcijalno rekurzivna funkcija i R^k rekurzivno prebrojiva relacija. Tada je i restrikcija $F|_R$ također parcijalno rekurzivna.*

Dokaz. Po korolaru 7.3 postoji parcijalno rekurzivna funkcija G takva da je $R = \mathcal{D}_G$ i $\mathcal{I}_G \subseteq \{0\}$. Tvrdimo da je tada $F|_R = F + G$. Doista, po formuli (2.2) je

$$\mathcal{D}_{F+G} = \mathcal{D}_F \cap \mathcal{D}_G = \mathcal{D}_F \cap R = \mathcal{D}_{F|_R}, \quad (7.3)$$

te za svaki \vec{x} iz te zajedničke domene vrijedi $(F + G)(\vec{x}) = F(\vec{x}) + G(\vec{x}) = F(\vec{x})$, jer je $G(\vec{x}) = 0$. Dakle, $F|_R = \text{add}^2 \circ (F, G)$ je parcijalno rekurzivna kao kompozicija takvih. \square

Dokazali smo zatvorenost skupa svih rekurzivno prebrojivih relacija (iste mjesnosti) na konačne presjeke. Što je sa zatvorenošću na konačne unije? Tu marljiva evaluacija nije pogodna. Ipak, pokazat ćemo da rekurzivno prebrojive relacije možemo gledati i na malo drugačiji „dualni” način, u kojem će unije biti vrlo prirodne.

7.1. Karakterizacija preko projekcije

U još jednom kontekstu smo vidjeli poluodlučivost: projekcija izračunljive relacije $\exists_* R$ ne mora biti izračunljiva, ali ako je $\vec{x} \in \exists_* R$, tada to možemo ustanoviti jednostavnim isprobavanjem, redom za sve $y \in \mathbb{N}$, vrijedi li $R(\vec{x}, y)$. Te dvije formalizacije (zapravo tri formalizacije, jer je svejedno formaliziramo li izračunljivost projicirane relacije kao primitivnu rekurzivnost ili kao rekurzivnost) su ekvivalentne.

Teorem 7.7. *Neka je R brojevena relacija. Tada su sljedeće tvrdnje ekvivalentne:*

- (1) R je rekurzivno prebrojiva;
- (2) R je projekcija neke rekurzivne relacije;
- (3) R je projekcija neke primitivno rekurzivne relacije.

Dokaz. Da (3) povlači (2) je trivijalno: svaka primitivno rekurzivna relacija je rekurzivna (primijenimo korolar 2.33 na njenu karakterističnu funkciju).

Da (2) povlači (1) je također jednostavno, samo umjesto kompozicije ovdje trebamo formulu za domenu *minimizacije*: $R = \exists_* P = \mathcal{D}_{\mu P}$ je domena funkcije μP prema (2.33), a ta funkcija je parcijalno rekurzivna jer je dobivena minimizacijom rekurzivne relacije.

Najzanimljiviji je dokaz da (1) povlači (3). Po definiciji rekurzivne prebrojivosti, postoji parcijalno rekurzivna funkcija F takva da je $R = \mathcal{D}_F$. Po korolaru 3.55, F ima indeks, označimo ga s e (sjetite se napomene 3.56). Označimo s k mjesnost od R odnosno F . Tada $R(\vec{x})$ možemo zapisati kao $\vec{x} \in \mathcal{D}_F = \mathcal{D}_{\{e\}^k}$, odnosno (e mora biti kod nekog RAM-programa čije izračunavanje s \vec{x} stane u nekom broju n koraka) $\exists n \text{ Final}(\langle \vec{x} \rangle, e, n)$. Dakle, dobili smo projekciju, sad samo treba ovo pod kvantifikatorom zapisati u odgovarajućem obliku: relacija zadana s

$$P(\vec{x}, n) : \Longleftrightarrow \text{Final}(\langle \vec{x} \rangle, e, n) \quad (7.4)$$

je primitivno rekurzivna jer joj je karakteristična funkcija dobivena kompozicijom primitivno rekurzivnih funkcija χ_{Final} , Code^k , C_e^{k+1} i koordinatnih projekcija (zapravo, $P \preceq \text{Final}$, uz dodatni uvjet da su „vezne funkcije” primitivno rekurzivne). \square

Jednom kad imamo karakterizaciju preko projekcije, lako je dokazati zatvorenost na konačne unije, jer se projekcija i unija (disjunkcija) obje mogu zapisati pomoću egzistencijalne kvantifikacije, a dva egzistencijalna kvantifikatora komutiraju.

Propozicija 7.8. *Neka su $k, l \in \mathbb{N}_+$, te $R_1^k, R_2^k, \dots, R_l^k$ rekurzivno prebrojive relacije, sve iste mjesnosti. Tada je i $\bigcup_{i=1}^l R_i$ također rekurzivno prebrojiva relacija.*

Dokaz. Po teoremu 7.7, za svaki $i \in [1..l]$ postoji rekurzivna relacija P_i^k takva da je $R_i = \exists_* P_i$. Sada je

$$\begin{aligned} \vec{x} \in \bigcup_{i=1}^l R_i = \bigcup_{i=1}^l \exists_* P_i &\iff (\exists i \in [1..l])(\vec{x} \in \exists_* P_i) \iff \\ &\iff (\exists i \in [1..l])(\exists y \in \mathbb{N}) P_i(\vec{x}, y) \iff (\exists y \in \mathbb{N})(\exists i \in [1..l])(\vec{x}, y) \in P_i \iff \\ &\iff (\exists y \in \mathbb{N}) \left((\vec{x}, y) \in \bigcup_{i=1}^l P_i \right) \iff \vec{x} \in \exists_* \left(\bigcup_{i=1}^l P_i \right), \quad (7.5) \end{aligned}$$

te je $P := \bigcup_{i=1}^l P_i$ rekurzivna po propoziciji 2.42 — a onda je $\bigcup_{i=1}^l R_i = \exists_* P$ rekurzivno prebrojiva po teoremu 7.7. \square

Primijetimo, projekcijska karakterizacija zapravo kaže da je ulaz \vec{x} u domeni funkcije koju (RAM-)algoritam računa ako i samo ako postoji n takav da nakon n koraka taj algoritam dođe u završnu konfiguraciju. U tom smislu, upravo dokazana propozicija daje jednostavnu implementaciju paralelnog računanja u l dretvi. Recimo, za $l = 2$,

$$\begin{aligned} \exists n \text{ Final}(\langle \vec{x} \rangle, \ulcorner P \urcorner, n) \vee \exists n \text{ Final}(\langle \vec{x} \rangle, \ulcorner Q \urcorner, n) &\iff \\ &\iff \exists n (\text{Final}(\langle \vec{x} \rangle, \ulcorner P \urcorner, n) \vee \text{Final}(\langle \vec{x} \rangle, \ulcorner Q \urcorner, n)) \quad (7.6) \end{aligned}$$

odgovara paralelnom pokretanju dva RAM-programa P i Q s istim ulaznim podacima \vec{x} , te čekanju dok jedan od njih ne stane. Efektivno, prvo pitamo za $n = 0$ vrijedi li desna strana u (7.6), odnosno je li ikoje od ta dva izračunavanja već na početku u završnoj konfiguraciji. Ako nije, pitamo istu stvar za $n = 1$: je li ikoje od tih izračunavanja nakon jednog koraka u završnoj konfiguraciji. Ako nije, pustimo ih još jedan korak ($n = 2$), i tako dalje. Jedini način da taj algoritam radi beskonačno dugo s \vec{x} , je da ni P -izračunavanje s \vec{x} ni Q -izračunavanje s \vec{x} ne stanu. Pritom je ključno da se radi o RAM-programima, gdje svaki korak mora završiti u konačnom vremenu.

7.1.1. Kontrakcija relacije

Još jedan način gledanja na (7.6) je da smo poredali sve testove $\text{Final}(\langle \vec{x} \rangle, \ulcorner P \urcorner, 0)$, $\text{Final}(\langle \vec{x} \rangle, \ulcorner Q \urcorner, 0)$, $\text{Final}(\langle \vec{x} \rangle, \ulcorner P \urcorner, 1)$, $\text{Final}(\langle \vec{x} \rangle, \ulcorner Q \urcorner, 1)$, $\text{Final}(\langle \vec{x} \rangle, \ulcorner P \urcorner, 2)$, \dots u niz tako da svaki dođe na red (svi početni komadi su konačni). Drugim riječima, imamo izračunljivu bijekciju između $\mathbb{N} + \mathbb{N} = \mathbb{N} \times \{0, 1\} \cong \mathbb{N} \times \text{bool}$ i \mathbb{N} .

Što bi se dogodilo da umjesto $\mathbb{N} \times \text{bool}$ uzmemo $\mathbb{N} \times \mathbb{N} \cong \mathbb{N}^2$? Umjesto jedne ograničene i jedne neograničene kvantifikacije, kao što smo imali u (7.5), sada bismo imali dvije neograničene — i komutiranje nam više nije dovoljno. Možemo li te dvije neograničene kvantifikacije zamijeniti jednom?

Na prvi pogled, treba nam izračunljiva bijekcija između \mathbb{N}^2 i \mathbb{N} , s izračunljivim inverzom (tzv. *pairing function*), ali pokazat će se da je injekcija dovoljna. Drugim riječima, samo nam treba neko kodiranje \mathbb{N}^2 . Napravili smo dva: Code^2 i bin^2 . Zaista je svejedno koje od njih (ili neko treće) koristimo, pa možemo to uobličiti kao sučelje (*interface*) za neki apstraktni tip podataka ($\text{std::pair}<\text{unsigned}, \text{unsigned}>$).

Lema 7.9. *Postoje primitivno rekurzivne funkcije pair^2 , fst^1 i snd^1 , takve da je $\text{fst} \circ \text{pair} = l_1^2$ i $\text{snd} \circ \text{pair} = l_2^2$.*

Točkovno, za sve $x, y \in \mathbb{N}$ vrijedi $\text{fst}(\text{pair}(x, y)) = x \wedge \text{snd}(\text{pair}(x, y)) = y$.

Prva implementacija. $\text{pair} := \text{Code}^2$, $\text{fst}(p) := p[0]$, $\text{snd}(p) := p[1]$. Tada je pair primitivno rekurzivna po propoziciji 3.4, a fst i snd po korolaru 6.2, jer su dobivene specijalizacijom primitivno rekurzivne funkcije part : $\text{fst} = \text{spec}(0, \text{part})$, $\text{snd} = \text{spec}(1, \text{part})$. Također za sve $x, y \in \mathbb{N}$ vrijedi

$$(\text{fst} \circ \text{pair})(x, y) = \text{fst}(\text{pair}(x, y)) = \text{part}(\text{Code}^2(x, y), 0) = \langle x, y \rangle[0] = x = l_1^2(x, y), \quad (7.7)$$

i analogno $\text{snd} \circ \text{pair} = l_2^2$, po propoziciji 3.14(2). \square

Druga implementacija. $\text{pair} := \text{bin}^2$, $\text{fst} := \text{arg}_1$, $\text{snd} := \text{arg}_2$. Tada je prva funkcija primitivno rekurzivna po propoziciji 4.62, a preostale dvije po propoziciji 4.70, iz koje slijedi i prikaz koordinatnih projekcija kao kompozicija $\text{fst} \circ \text{pair}$ i $\text{snd} \circ \text{pair}$. \square

Treća implementacija. $\text{pair}(i, j) := i + \sum_{t \leq i+j} t$. Ovo je standardna enumeracija „po sporednim dijagonalama”, primitivno rekurzivna po napomenama 2.51 i 2.54. Njenih je prvih devet vrijednosti prikazano u tablici

pair	0	1	2	3	...
0	0	1	3	6	...
1	2	4	7		
2	5	8			
\vdots	\vdots	\vdots			

(7.8)

Ona je prikazana samo da se vidi kako nije teško naći bijektivno kodiranje — ali ponovimo, za naše potrebe bijektivnost nije nužna. U ovoj implementaciji, funkcije fst i snd možemo implementirati grubom silom, kao u (5.39) i (5.40):

$$\text{fst}(p) := (\mu i \leq p)(\exists j \leq p)(p = \text{pair}(i, j)), \quad (7.9)$$

$$\text{snd}(p) := (\mu j \leq p)(\exists i \leq p)(p = \text{pair}(i, j)), \quad (7.10)$$

što je primitivno rekurzivno jer smo ograničili i i j s p , a što smo mogli jer je uvijek $\text{pair}(i, j) \geq \sum_{t \leq i+j} t \geq i + j \geq i, j$. \square

Ubuduće smatramo da smo fiksirali neku implementaciju, ali nećemo od nje ništa „privatno” koristiti osim sučelja opisanog u lemi. Za početak možemo dokazati ostatak tvrdnji potrebnih da bismo imali kodiranje.

Propozicija 7.10. *Funkcija pair je injekcija, i slika joj je primitivno rekurzivna.*

Dokaz. Pretpostavimo da su $a, b, c, d \in \mathbb{N}$ takvi da vrijedi $\text{pair}(a, b) = \text{pair}(c, d)$. Tada

$$a = l_1^2(a, b) = \text{fst}(\text{pair}(a, b)) = \text{fst}(\text{pair}(c, d)) = l_1^2(c, d) = c, \quad (7.11)$$

i analogno $b = d$, pa je $(a, b) = (c, d)$, odnosno pair je injekcija.

Za primitivnu rekurzivnost slike, koristimo istu tehniku kao u dokazu korolara 3.16: $p \in \mathcal{I}_{\text{pair}} \iff (\exists (x, y) \in \mathbb{N}^2)(p = \text{pair}(x, y))$, i jedini kandidat za x odnosno y je $\text{fst}(p)$ odnosno $\text{snd}(p)$. Dakle, vrijedi $p \in \mathcal{I}_{\text{pair}} \iff p = \text{pair}(\text{fst}(p), \text{snd}(p))$, što je primitivno rekurzivno jer su pair , fst , snd i $\chi_{=}$ takve. \square

Sad kada imamo specifikaciju sparivanja, pogledajmo detaljnije što smo napravili s relacijom Final na kraju prethodne točke. Zapravo smo dva njena zadnja argumenta (e i n) „spojili” u jedan, po kojem smo onda enumerirali testove završetka. Tamo smo za e imali samo dvije mogućnosti, $\ulcorner P \urcorner$ i $\ulcorner Q \urcorner$, ali lako je zamisliti da ih možemo imati i više — pa čak i sve prirodne brojeve na njihovom mjestu.

Definicija 7.11. Neka je $k \in \mathbb{N}_+$, te P^{k+1} relacija. Za k -mjesnu relaciju \hat{P} , zadanu s

$$\hat{P}(\vec{x}, y) :\iff P(\vec{x}, \text{fst}(y), \text{snd}(y)), \quad (7.12)$$

kažemo da je dobivena *kontrakcijom* zadnja dva argumenta od P . \triangleleft

Prvo, uočimo da su P i \hat{P} , „jednako teške” kad ih promatramo kao probleme. Za relaciju svedivosti \preceq smo u propoziciji 5.13 vidjeli da je refleksivna i tranzitivna. Ipak, ona nije parcijalni uređaj jer nije antisimetrična: relacija i njena kontrakcija su dobar kontraprimjer.

Lema 7.12. *Za svaku relaciju P mjesnosti barem 2, vrijedi $P \preceq \hat{P} \preceq P$.*

Dokaz. Druga „nejednakost” slijedi direktno iz definicije, i činjenice da su fst i snd (primitivno) rekurzivne. Prva će slijediti iz rekurzivnosti od pair , čim dokažemo da za sve \vec{x}, y, z vrijedi

$$P(\vec{x}, y, z) \iff \hat{P}(\vec{x}, \text{pair}(y, z)). \quad (7.13)$$

A to pak vrijedi jer je po definiciji \hat{P} ,

$$\hat{P}(\vec{x}, \text{pair}(y, z)) \iff P(\vec{x}, \text{fst}(\text{pair}(y, z)), \text{snd}(\text{pair}(y, z))) \iff P(\vec{x}, y, z), \quad (7.14)$$

budući da je po lemi 7.9 $\text{fst}(\text{pair}(y, z)) = y$ i analogno $\text{snd}(\text{pair}(y, z)) = z$. \square

Napomena 7.13. Štoviše, jer su „vezne funkcije” primitivno rekurzivne, vrijedi da je P primitivno rekurzivna ako i samo ako je \hat{P} primitivno rekurzivna. Ali to nam neće bitno trebati. \triangleleft

7.1.2. Kontrakcija kvantifikatora

Sada možemo i formalno dokazati da je više projekcija jednako izračunljivo kao i jedna — jer dvije projekcije su kao jedna projekcija kontrahirane relacije. Intuitivno, recimo za tromjesne rekurzivne relacije, ispitujemo postoje li y i z takvi da vrijedi $R(x, y, z)$, tako da parove (y, z) poredamo prema funkciji **pair** (npr. po sporednim dijagonalama), kako bismo bili sigurni da će svaki doći na red.

Propozicija 7.14. *Za svaku relaciju mjesnosti barem 3, vrijedi $\exists_* \exists_* P = \exists_* \hat{P}$.*

Dokaz. Za (\supseteq) , pretpostavimo $\vec{x} \in \exists_* \hat{P}$. Po definiciji projekcije to znači da postoji $t \in \mathbb{N}$ takav da vrijedi $\hat{P}(\vec{x}, t)$, odnosno $P(\vec{x}, \text{fst}(t), \text{snd}(t))$. No to znači da je $(\vec{x}, \text{fst}(t)) \in \exists_* P$, pa onda i $\vec{x} \in \exists_* \exists_* P$.

Za (\subseteq) , iz $\vec{x} \in \exists_* \exists_* P$ slijedi da postoji $y \in \mathbb{N}$ takav da je $(\vec{x}, y) \in \exists_* P$, što pak znači da postoji i $z \in \mathbb{N}$ takav da je $(\vec{x}, y, z) \in P$. Sada po (7.13) slijedi $\hat{P}(\vec{x}, t)$ za $t := \text{pair}(y, z)$, odnosno $\vec{x} \in \exists_* \hat{P}$. \square

Napomena 7.15. Ista tvrdnja bi se mogla dokazati za dva univerzalna kvantifikatora uzastopce. Zaključujemo da nizanje kvantifikatora iste vrste ne otežava bitno problem (u smislu postojanja algoritma, ne u smislu performansi). Ipak, ispreplitanje kvantifikatora $(\forall \exists \forall \dots)$ općenito otežava probleme koje promatramo. Uzmimo totalnost: $\{e\}^2$ je totalna ako i samo ako vrijedi $\forall x_1 \forall x_2 \exists n \text{Final}(\langle x_1, x_2 \rangle, e, n)$, i može se pokazati da je to bitno teže od bilo kakvog problema koji sadrži samo egzistencijalne ili samo univerzalne kvantifikatore, i nakon toga rekurzivnu relaciju. Ideja da problemi postaju sve teži kako dodajemo sve više kvantifikatora, uvijek suprotne vrste od one koju smo upravo dodali, formalizirana je u pojmu *aritmetičke hijerarhije*. Definiciju i neke osnovne teoreme možete naći u [16], a puno više detalja u [11]. \triangleleft

Jednostavna posljedica upravo dokazanog rezultata i projekcijske karakterizacije je da projekcija ne može otežati već poluodlučiv problem.

Propozicija 7.16. *Projekcija rekurzivno prebrojive relacije (ako je ova mjesnosti barem 2) je ponovo rekurzivno prebrojiva.*

Dokaz. Neka je $k \geq 2$ i R^k rekurzivno prebrojiva relacija. Po teoremu 7.7, postoji rekurzivna relacija P takva da je $R = \exists_* P$. No tada je projekcija $\exists_* R = \exists_* \exists_* P = \exists_* \hat{P}$ po propoziciji 7.14, što je rekurzivno prebrojivo opet po teoremu 7.7. Naime, iz leme 7.12 imamo $\hat{P} \preceq P$, pa je \hat{P} također rekurzivna po propoziciji 5.14. \square

Korolar 7.17. *Neka su $k, l \in \mathbb{N}_+$, te R^{k+l} rekurzivno prebrojiva relacija. Tada je i relacija Q^k , zadana s $Q(\vec{x}) :\iff (\exists \vec{y} \in \mathbb{N}^l) R(\vec{x}, \vec{y})$, također rekurzivno prebrojiva.*

Dokaz. Definiciju od Q možemo zapisati kao $Q = \exists_* \exists_* \dots \exists_* R = (\exists_*)^l R$, te tvrdnju možemo dokazati indukcijom po l . Baza ($l = 1$) je jednostavno propozicija 7.16, a u koraku iz pretpostavke da je $(\exists_*)^m R$ rekurzivno prebrojiva dokažemo da je $(\exists_*)^{m+1} R = \exists_* (\exists_*)^m R$ rekurzivno prebrojiva po istoj toj propoziciji. \square

7.2. Teorem o grafu za parcijalne funkcije

Davno smo bili dokazali teorem o grafu za totalne funkcije (teorem 3.44), i vidjeli smo da pruža opravdanje za skupovnoteorijsko gledanje funkcija kao njihovih grafova — totalna funkcija je jednako izračunljiva kao i njen graf. Kad su funkcije parcijalne, stvar je kompliciranija, jer (3.84) kaže da moramo uzeti u obzir i domen — koja ne mora biti izračunljiva, čak ni za izračunljive funkcije.

Sada kad smo napokon vidjeli da parcijalnost za izračunljive funkcije odgovara rekurzivnoj prebrojivosti za relacije, prirodno je pitati se vrijedi li i taj oblik teorema o grafu: parcijalna rekurzivnost f kao rekurzivna prebrojivost \mathcal{G}_f . Smjer slijeva nadesno slijedi iz projekcijske karakterizacije (teorem 7.7), i već ga sada možemo dokazati.

Teorem 7.18. *Graf svake parcijalno rekurzivne funkcije je rekurzivno prebrojiv.*

Ideja je slična kao kod projekcijske karakterizacije; jedina razlika između domene i grafa je što kod grafa moramo voditi računa i o funkcijskim vrijednostima. Zato nam više nije dovoljna relacija $\exists_* \text{Final}$ koja kaže „postoji neki broj koraka”, već „postoji neko izračunavanje koje stane” (iz kojeg možemo izvući funkcijsku vrijednost). Srećom, upravo to nam daje Kleenejev teorem o normalnoj formi.

Dokaz. Neka je $k \in \mathbb{N}_+$, i F^k parcijalno rekurzivna. Po korolaru 3.52 F ima indeks, označimo ga (napomena 3.56) s e . Tada po teoremu 3.50 imamo:

$$\mathcal{G}_F(\vec{x}, y) \iff \vec{x} \in \mathcal{D}_{\{e\}^k} \wedge \{e\}^k(\vec{x}) = y \iff \exists z (T_k(\vec{x}, e, z) \wedge U(z) = y). \quad (7.15)$$

Doista, ako je $(\vec{x}, y) \in \mathcal{G}_F$, tada je $\mathcal{G}_F \neq \emptyset$, odnosno $F \neq \emptyset$, iz čega slijedi $\text{Prog}(e)$ (kontrapozicijom propozicije 3.57(2)), pa postoji (jedinstven) RAM-program P takav da je $e = \ulcorner P \urcorner$. Po korolaru 3.58, P^k računa $\{e\}^k = F$.

Također, iz $(\vec{x}, y) \in \mathcal{G}_F$ zaključujemo $\vec{x} \in \mathcal{D}_F$, pa po definiciji 1.10, P -izračunavanje s \vec{x} stane. Označimo sa z kod tog izračunavanja. Tada po propoziciji 3.46 vrijedi $T_k(\vec{x}, e, z)$, te je $y = U(z)$ jer je to izlazni podatak tog izračunavanja.

U drugom smjeru, pretpostavimo da postoji $z \in \mathbb{N}$ takav da vrijedi $T_k(\vec{x}, e, z)$ i $y = U(z)$. Opet po propoziciji 3.46, slijedi da postoji RAM-program P takav da je $e = \ulcorner P \urcorner$, i z je upravo kod P -izračunavanja s \vec{x} . Kako taj kod postoji, zaključujemo da izračunavanje stane, pa je $\vec{x} \in \mathcal{D}_F$ (kao i prije, P^k računa F^k). Tada je $y = U(z)$ izlazni podatak tog izračunavanja, pa je po definiciji 1.10 $y = F(\vec{x})$, odnosno $(\vec{x}, y) \in \mathcal{G}_F$.

Sada zaključujemo ovako: za fiksni e , relacija zadana s

$$R(\vec{x}, y, z) :\iff T_k(\vec{x}, e, z) \wedge U(z) = y \quad (7.16)$$

je primitivno rekurzivna po propoziciji 2.39, kao konjunkcija dvije primitivno rekurzivne relacije. Tada je po (7.15) $\mathcal{G}_F = \exists_* R$, rekurzivno prebrojiva po teoremu 7.7. \square

Sljedeći veliki zalogaj je dokazati obrat teorema 7.18. Kod teorema za totalne funkcije, jednostavno smo koristili minimizaciju: po (3.86), funkcija je jednaka minimizaciji svog grafa. To vrijedi za sve funkcije (čak i za neizračunljive), ali neće nam pomoći ako je graf samo poluodlučiv: skup rekurzivno prebrojivih relacija nije zatvoren na minimizaciju (što bi to uopće značilo?), a i intuitivno, minimizacija poluodlučive relacije ne mora biti izračunljiva.

Razmislimo malo detaljnije o tome. Zamislimo da imamo poluodlučivu relaciju R^2 , i želimo računati vrijednosti funkcije $f := \mu R$. Za zadani x , recimo $x = 5$, dakle, tražimo najmanji y takav da vrijedi $R(5, y)$ — no problem je u tome što za zadani par $(5, y)$ možemo jedino sa sigurnošću ustanoviti da *jest* u R , ne i da nije (ako nije, postupak jednostavno neće nikada stati). Ako takvog y nema, to da algoritam neće nikad stati je sasvim u redu, ali što ako ga ima? Pomoću paralelizacije možemo pokrenuti sva testiranja $R(5, 0), R(5, 1), R(5, 2), \dots$ dok neko od njih ne stane — i ako to upravo bude $R(5, 0)$, onda imamo sreće: $f(5) = 0$. No pretpostavimo da smo umjesto toga dobili $R(5, 7)$. Iz toga sigurno možemo zaključiti $5 \in \exists_* R = \mathcal{D}_{\mu R} = \mathcal{D}_f$, dakle $f(5)$ je neki broj. Štoviše, to je broj koji sigurno nije veći od 7, ali koji? Imamo 8 mogućnosti, i dok god testovi $R(5, t), t < 7$ ne stanu, ne znamo koja od njih je prava vrijednost $f(5)$. Paradoksalno, ako to *jest* 7, tada nijedan od tih „manjih testova” neće stati, pa više nećemo dobiti nikakvu novu informaciju. Ako u međuvremenu saznamo da vrijedi i $R(5, 11)$, to neće nimalo utjecati na naš problem. Ali zamislimo da smo (nakon još puno koraka) dobili da vrijedi $R(5, 4)$. Imamo li se razloga radovati? S jedne strane, da — eliminirali smo brojeve 5, 6 i 7 kao potencijalne vrijednosti $f(5)$. Ali s druge strane, i dalje ne znamo koliko je $f(5)$. Ukratko, na ovaj način možemo dobiti $f(x)$ jedino ako je $f(x) = 0$ — što se ne čini pretjerano korisnim.

Ipak, funkcija g , koju dobijemo ako za svaki x uzmemo prvi y na koji naiđemo paralelnim izvršavanjem svih testova $R(x, y)$ (recimo, u gornjem scenariju je $g(5) = 7$), je izračunljiva, i nije baš sasvim bez veze s funkcijom f . Vidimo da ona *dominira* funkciju f : kad god je $f(x)$ definirano, tada je i $g(x)$ definirano, te vrijedi $g(x) \geq f(x)$.

Jedini problem je što nama nije zadana funkcija f , nego relacija R . Možemo li g karakterizirati pomoću R ? Da: prvo svojstvo kaže $\mathcal{D}_g \supseteq \mathcal{D}_f = \mathcal{D}_{\mu R} = \exists_* R$, a drugo kaže (za $x \in \exists_* R$) $f(x) = \mu y R(x, y) \leq g(x)$, što jednostavno možemo osigurati tako da tražimo $R(x, g(x))$. Reći da to vrijedi za sve x zapravo znači zahtijevati inkluziju $\mathcal{G}_g \subseteq R$, a tada po (3.85) slijedi (lako se vidi da je projekcija monotona) $\mathcal{D}_g = \exists_* \mathcal{G}_g \subseteq \exists_* R$, što s prvim svojstvom daje $\mathcal{D}_g = \exists_* R$. Formalizirajmo to svojsvo.

Definicija 7.19. Neka je $k \in \mathbb{N}_+$, te R^{k+1} relacija.

Za funkciju g^k kažemo da je *selektor* za R ako vrijedi $\mathcal{D}_g = \exists_* R$ i $\mathcal{G}_g \subseteq R$. ◁

Točkovno, uvjete na selektor možemo zapisati kao

$$\exists y (y = g(\vec{x})) \iff \exists y R(\vec{x}, y) \iff R(\vec{x}, g(\vec{x})). \quad (7.17)$$

Selektor je sasvim općenit pojam, i u skupovnoteorijskom smislu odgovara *izbornoj funkciji*: Ako za svaki $\vec{x} \in \mathbb{N}^k$ definiramo „prerez” (*section*) $R_{\vec{x}}^1(y) : \iff R(\vec{x}, y)$, tada je $(R_{\vec{x}})_{\vec{x} \in \exists_* R}$ indeksirana familija nepraznih skupova (jednomjesnih relacija), i selektor za R je upravo izborna funkcija za tu familiju (pogledajte [18, str. 92] za detalje).

U teoriji skupova postoji aksiom izbora, koji kaže da svaka takva familija ima izbornu funkciju, odnosno svaka relacija ima selektor. Zapravo, ovdje aksiom izbora nije ni potreban, jer je skup \mathbb{N} dobro uređen, pa uvijek možemo odabrati najmanji y za svaki \vec{x} za koji takav y postoji. Drugim riječima, trivijalno je μR selektor za R . Doista, vrijedi $\mathcal{D}_{\mu R} = \exists_* R$ po definiciji, te za $\mathcal{G}_{\mu R}(\vec{x}, y)$ vrijedi $y = \mu z R(\vec{x}, z) = \min R_{\vec{x}} \in R_{\vec{x}}$, dakle vrijedi $R_{\vec{x}}(y)$, odnosno $R(\vec{x}, y)$.

Nas, međutim, ne zanimaju proizvoljni selektori, već samo oni izračunljivi — a postupak koji smo opisali (paralelno testiranje svih $R(\vec{x}, y)$, $y \in \mathbb{N}$) pokazuje da svaka poluodlučiva relacija ima izračunljiv selektor.

Lema 7.20 (Teorem o selektoru). *Svaka rekurzivno prebrojiva relacija (mjesnosti barem 2) ima parcijalno rekurzivan selektor.*

Dokaz. Neka je $k' \geq 2$, i $R^{k'}$ rekurzivno prebrojiva. Označimo $k := k' - 1 \in \mathbb{N}_+$. Trebamo parcijalno rekurzivnu funkciju F^k takvu da vrijedi $\mathcal{D}_F = \exists_* R$ i $\mathcal{G}_F \subseteq R$.

Prvo, projekcijskom karakterizacijom se dočepamo *izračunljive* relacije, po cijenu još jednog egzistencijalnog kvantifikatora: po teoremu 7.7 postoji rekurzivna relacija P^{k+2} takva da je $R = \exists_* P$.

Sad za domenu tražene funkcije znamo da mora biti $\mathcal{D}_F = \exists_* R = \exists_* \exists_* P$, što je po propoziciji 7.14 jednako $\exists_* \hat{P} = \mathcal{D}_{\mu \hat{P}}$. Bi li moglo biti $F = \mu \hat{P}$? Ne, jer „tipovi” ne pašu: $\mu \hat{P}$ će nam dati zadnji argument od \hat{P} , dakle „par” brojeva y i z , gdje y predstavlja broj koji tražimo, a z je samo broj koraka nakon kojeg smo saznali da vrijedi $R(\vec{x}, y)$.

Dakle, tvrdimo da $F := \text{fst} \circ \mu \hat{P}$ zadovoljava sve uvjete. Očito je parcijalno rekurzivna jer je dobivena kompozicijom primitivno rekurzivne funkcije, i funkcije dobivene minimizacijom rekurzivne (lema 7.12 i propozicija 5.14) relacije.

Štoviše, kako je fst primitivno rekurzivna, i stoga totalna, vrijedi

$$\mathcal{D}_F = \mathcal{D}_{\text{fst} \circ \mu \hat{P}} = \mathcal{D}_{\mu \hat{P}} = \exists_* \hat{P} = \exists_* \exists_* P = \exists_* R, \quad (7.18)$$

što je upravo prvi uvjet za selektor. Još samo treba pokazati drugi uvjet, pa neka je $(\vec{x}, y) \in \mathcal{G}_F$. To po definiciji znači $\vec{x} \in \mathcal{D}_F = \exists_* \hat{P}$, i $y = \text{fst}(\mu \hat{P}(\vec{x}, t))$. Tada $\vec{x} \in \exists_* \hat{P}$ znači da postoji $t \in \mathbb{N}$ takav da vrijedi $\hat{P}(\vec{x}, t)$, pa ako najmanji takav označimo s v , vrijedi $\hat{P}(\vec{x}, v)$ i $y = \text{fst}(v)$. No ovo prvo po definiciji znači da vrijedi $P(\vec{x}, \text{fst}(v), \text{snd}(v))$, što je zbog drugog ekvivalentno s $P(\vec{x}, y, \text{snd}(v))$. To pak znači da postoji $z \in \mathbb{N}$ (konkretno, $z := \text{snd}(v)$) takav da je $(\vec{x}, y, z) \in P$, odnosno $(\vec{x}, y) \in \exists_* P = R$. \square

Dakle, za svaku rekurzivno prebrojivu relaciju možemo iz svakog nepraznog prereza $R_{\vec{x}}$ izračunljivo odabrati po jedan element y — to neće nužno biti najmanji, nego će

biti onaj čiju je pripadnost prerezu „najlakše” utvrditi — odnosno onaj za koji je kod para (y, n) najmanji, gdje je n broj koraka potrebnih da se utvrdi $R(\vec{x}, y)$.

Formalno, tražimo najmanji t takav da je $(fst(t), snd(t)) = (y, n)$ — to nije nužno baš $pair(y, n)$, iako jest u svakoj od tri implementacije leme 7.9 koje smo vidjeli. Možete se zabaviti dokazivanjem toga, ili smišljanjem implementacije $(pair, fst, snd)$ za koju to ne vrijedi — ali to zapravo nije toliko bitno, jer će nas zanimati primjena leme 7.20 samo na vrlo specijalne relacije.

Mi želimo dokazati obrat teorema 7.18, dakle rekurzivno prebrojiva relacija s kojom radimo nije bilo kakva — ona je graf funkcije, pa ima funkcijsko svojstvo. A funkcijsko svojstvo je upravo ono koje nam treba da se riješimo „latentnog nedeterminizma” u lemi 7.20, jer ono znači da svaki prerez ima najviše jedan element — pa „najmanji”, „bilo koji” i „jedini” element prereza znače jedno te isto, i nisu definirani ako i samo ako je prerez prazan.

Lema 7.21. *Za svaku brojevnju funkciju F , jedini selektor grafa \mathcal{G}_F je upravo F .*

Dokaz. Funkcija F očito jest selektor za \mathcal{G}_F : prvi uvjet $\mathcal{D}_F = \exists_* \mathcal{G}_F$ smo dokazali odavno (lema 3.43), a drugi $\mathcal{G}_F \subseteq \mathcal{G}_F$ je trivijalan.

Za jedinstvenost, neka je G proizvoljni selektor za \mathcal{G}_F . Tada po prvom uvjetu za selektor vrijedi $\mathcal{D}_G = \exists_* \mathcal{G}_F = \mathcal{D}_F$, dakle F i G imaju istu domenu. Za svaki \vec{x} iz te domene, po drugom uvjetu vrijedi $(\vec{x}, G(\vec{x})) \in \mathcal{G}_G \subseteq \mathcal{G}_F$, pa po definiciji grafa vrijedi $F(\vec{x}) = G(\vec{x})$.

Drugim riječima, G i F se podudaraju na zajedničkoj domeni, pa su jednake. \square

Sada još samo treba sve što smo napravili objediniti u jedan teorem.

Teorem 7.22 (Teorem o grafu za parcijalne funkcije). *Neka je F brojevnja funkcija.*

Tada je F parcijalno rekurzivna ako i samo ako je \mathcal{G}_F rekurzivno prebrojiv.

Dokaz. Smjer (\Rightarrow) smo već dokazali, pomoću projekcijske karakterizacije i Kleenejevog teorema o normalnoj formi (teorem 7.18).

Za smjer (\Leftarrow) , neka je \mathcal{G}_F rekurzivno prebrojiv. Prema lemi 7.20, postoji parcijalno rekurzivan selektor G za \mathcal{G}_F . No kako je prema lemi 7.21 jedini selektor za \mathcal{G}_F upravo F , zaključujemo $F = G$. Kako je G parcijalno rekurzivna, i F je parcijalno rekurzivna (jer je to ista funkcija). \square

7.2.1. Primjene teorema o grafu

Jedna jednostavna posljedica teorema o grafu je generalizacija propozicije 2.47 za parcijalne funkcije. Tamo smo vidjeli da možemo uzeti izračunljivu totalnu funkciju, i promijeniti joj konačno mnogo vrijednosti, ne kvareći njenu izračunljivost. Bilo je ključno da tako dobivena funkcija također bude totalna.

Što ako bismo smjeli *uklanjati* točke iz domene po volji? *Dodavanje* točaka je neizvedivo za totalne funkcije, ali ako je polazna funkcija parcijalna, možemo i to. Hoćemo li time pokvariti izračunljivost? Pokazuje se da nećemo, ako napravimo konačno mnogo takvih promjena.

Prvo dokažimo nekoliko jednostavnih ali korisnih lema, koje govore o skupovnim operacijama koje čuvaju parcijalnu rekurzivnost. Za unije i presjeke smo to već vidjeli (propozicije 7.8 i 7.5), vrijeme je da se pozabavimo razlikama. Napomenimo samo da općenito skupovna razlika dvije rekurzivno prebrojive relacije nije rekurzivno prebrojiva, jer *komplement* kao specijalni slučaj razlike $((R^k)^c = \mathbb{N}^k \setminus R)$ ne čuva rekurzivnu prebrojivost. Detaljnije ćemo vidjeti što se tu zbiva kad dokažemo Postov teorem.

Lema 7.23. *Neka je $k \in \mathbb{N}_+$, P^k rekurzivno prebrojiva relacija, i R rekurzivna relacija. Tada je $P \setminus R$ također rekurzivno prebrojiva.*

Dokaz. Znamo iz teorije skupova (uz \mathbb{N}^k kao univerzalni skup) da je $P \setminus R = P \cap R^c$. Sada je po propoziciji 2.37, R^c rekurzivna, pa je po propoziciji 7.2 rekurzivno prebrojiva — a onda tvrdnja slijedi po propoziciji 7.5. \square

Za *simetričnu* razliku, više nije dovoljno zahtijevati da R bude rekurzivna, pa čak ni primitivno rekurzivna — isti kontraprimjer, zapisan u obliku $(R^k)^c = R \triangle \mathbb{N}^k$, funkcionira. Ali ako je R *konačna*, dokaz prolazi.

Lema 7.24. *Neka je $k \in \mathbb{N}_+$, P^k rekurzivno prebrojiva relacija, i R^k konačna relacija. Tada je $P \triangle R$ također rekurzivno prebrojiva.*

Dokaz. Po definiciji je $P \triangle R := (P \setminus R) \cup (R \setminus P)$. R je rekurzivna po korolarima 2.46 i 2.33, dakle prva razlika je rekurzivno prebrojiva po lemi 7.23. Iz teorije skupova znamo da je podskup konačnog skupa konačan, pa je druga razlika $R \setminus P \subseteq R$ konačna, te je primitivno rekurzivna po korolaru 2.46, rekurzivna po korolaru 2.33, i rekurzivno prebrojiva po propoziciji 7.2. Sada tvrdnja slijedi iz propozicije 7.8 (unija dvije rekurzivno prebrojive relacije iste mjesnosti je ponovo rekurzivno prebrojiva). \square

Propozicija 7.25. *Neka je $k \in \mathbb{N}_+$, te F^k parcijalno rekurzivna, i G^k funkcija iste mjesnosti takva da je skup $\mathcal{G}_F \triangle \mathcal{G}_G$ konačan. Tada je i G parcijalno rekurzivna.*

Dokaz. Po teoremu o grafu, \mathcal{G}_F je rekurzivno prebrojiv skup (mjesnosti k). Ako označimo zadani konačni „diff” grafova s $E := \mathcal{G}_F \triangle \mathcal{G}_G$, iz teorije skupova znamo — $(\mathcal{P}(\mathbb{N}^k), \triangle)$ je grupa u kojoj je svaki element sam sebi inverz — da iz toga slijedi $\mathcal{G}_G = \mathcal{G}_F \triangle E$, što je rekurzivno prebrojiv skup po lemi 7.24. Opet po teoremu o grafu, iz toga slijedi da je G parcijalno rekurzivna funkcija. \square

Upravo napisani dokaz primjer je vrlo općenite tehnike kojom možemo dokazati da svakakve operacije na funkcijama čuvaju parcijalnu rekurzivnost: prebacimo se na grafove, dokažemo da odgovarajuća operacija na grafovima čuva rekurzivnu prebrojivost, i vratimo se natrag na funkcije. Evo još jednog primjera.

Dokaz teorema o grafu, i još prije projekcijska karakterizacija, daju nam intuiciju poluodlučivosti kao *čekanja*: čekamo da se nešto dogodi (konkretno, da izračunavanje stane), i ako se dogodi, znamo da se dogodilo, ali ako se ne dogodi, ne znamo hoće li se nikad ne dogoditi ili jednostavno nismo još dovoljno dugo čekali.

U tom kontekstu, paralelizacija kaže jednostavno da možemo čekati na više (čak beskonačno mnogo) stvari odjednom, dok se barem jedna od njih ne dogodi. Ako ih se dogodi i više, to također ne smeta dok nas zanima samo da/ne odgovor (kao u propoziciji 7.8 ili 7.16), ali ako imamo funkcije s kompliciranijim vrijednostima, to može biti problem („latentni nedeterminizam” selektora).

Da bismo sačuvali determinizam, moramo nekako „izvana” osigurati da se ne dogodi više stvari, odnosno trebamo imati nekoliko u parovima disjunktne poluodlučivih relacija. Ako sada na svakoj od njih definiramo neku izračunljivu funkciju, možemo čekati dok se ne dogodi neki od zadanih događaja, i onda izračunati odgovarajuću funkciju.

Time smo opisali već poznatu konstrukciju grananja, ali ovaj put s *poluodlučivim* uvjetima. Drugim riječima, da bi funkcija dobivena grananjem bila izračunljiva, ne moramo nužno moći za svaki uvjet ustanoviti u konačno mnogo koraka vrijedi li ili ne (odlučivost koju smo dosad uvijek pretpostavljali, u teoremima 2.44 i 3.61) — možemo i *čekati* dok neki uvjet ne postane istinit, i tada izračunati odgovarajuću granu. Pri tome treba paziti da po definiciji poluodlučivosti **ne možemo imati granu „inače”** — jer ni u kojem trenutku ne možemo biti sigurni da smo čekali dovoljno dugo da znamo da se nijedan uvjet nikada neće ispuniti.

Na operacijskom sustavu UNIX (i mnogim srodnim sustavima) postoji sistemski poziv `select` koji, do na implementacijske detalje, radi upravo to: prima listu uvjeta oblika „postoji li vrsta aktivnosti x (čitanje, pisanje, greška, ...) na deskriptoru y ”, i vraća se čim neki od tih uvjeta bude ispunjen. Iz praktičnih razloga, taj poziv ima i granu „inače”, u obliku isteka vremena (*timeout*) određenog pri pozivu. Kako nemamo sistemski sat (iako ga možemo emulirati brojenjem koraka izračunavanja — pokušajte!), nama će taj dio jednostavno biti beskonačna petlja, što je u skladu s konvencijom da ako u grananju ne navedemo granu G_0 , podrazumijeva se prazna funkcija.

Teorem 7.26 (Teorem o grananju, rekurzivno prebrojiva verzija). *Neka su $k, l \in \mathbb{N}_+$, neka su $G_1^k, G_2^k, \dots, G_l^k$ parcijalno rekurzivne funkcije, te $R_1^k, R_2^k, \dots, R_l^k$ u parovima disjunktne rekurzivno prebrojive relacije, sve iste mjesnosti. Tada je i funkcija $F := \{R_1 : G_1, R_2 : G_2, \dots, R_l : G_l\}$ također parcijalno rekurzivna.*

Dokaz. Prvo, osigurajmo da nijedna grana nije definirana izvan svog uvjeta. Dakle,

za svaki $i \in [1..l]$ definiramo $H_i := G_i|_{R_i}$. Svaka H_i je parcijalno rekurzivna po korolaru 7.6, i vrijedi $F = \{\mathcal{D}_{H_1} : H_1, \mathcal{D}_{H_2} : H_2, \dots, \mathcal{D}_{H_l} : H_l\}$. Doista, domene su im iste: $\bigcup_{i=1}^l \mathcal{D}_{H_i} = \bigcup_{i=1}^l (\mathcal{D}_{G_i} \cap R_i)$; i za svaki \vec{x} iz te domene postoji jedinstveni $j \in [1..l]$ takav da je $\vec{x} \in \mathcal{D}_{H_j}$, pa je $F(\vec{x}) = G_j(\vec{x}) = H_j(\vec{x})$.

Drugo, dokažimo

$$\mathcal{G}_F = \bigcup_{i=1}^l \mathcal{G}_{H_i}. \quad (7.19)$$

Za inkluziju (\subseteq), neka je $(\vec{x}, y) \in \mathcal{G}_F$. To znači da je $\vec{x} \in \mathcal{D}_F$ i $y = F(\vec{x})$. Prvo znači da postoji (jedinstveni) $j \in [1..l]$ takav da vrijedi $\vec{x} \in \mathcal{D}_{H_j}$, te ovo drugo onda znači $y = H_j(\vec{x})$. No tada je $(\vec{x}, y) \in \mathcal{G}_{H_j} \subseteq \bigcup_{i=1}^l \mathcal{G}_{H_i}$.

Za inkluziju (\supseteq), neka je $(\vec{x}, y) \in \bigcup_{i=1}^l \mathcal{G}_{H_i}$. Tada postoji $j \in [1..l]$ takav da je $(\vec{x}, y) \in \mathcal{G}_{H_j}$. To znači da je $\vec{x} \in \mathcal{D}_{H_j} = \mathcal{D}_{G_j|_{R_j}} = \mathcal{D}_{G_j} \cap R_j \subseteq R_j$, pa je $F(\vec{x}) \simeq G_j(\vec{x})$. No $\vec{x} \in \mathcal{D}_{G_j}$ znači da postoji $y' \in \mathbb{N}$ takav da je $G_j(\vec{x}) = y'$, pa je i $F(\vec{x}) = y'$. S druge strane $(\vec{x}, y) \in \mathcal{G}_{H_j} \subseteq \mathcal{G}_{G_j}$ znači da su (\vec{x}, y) i (\vec{x}, y') oba u \mathcal{G}_{G_j} , te je $y = y'$ jer \mathcal{G}_{G_j} ima funkcijsko svojstvo. Iz toga slijedi $F(\vec{x}) = y$, odnosno $(\vec{x}, y) \in \mathcal{G}_F$.

Sada je lako: po teoremu 7.18, svaki \mathcal{G}_{H_i} je rekurzivno prebrojiv. Po propoziciji 7.8, \mathcal{G}_F je rekurzivno prebrojiv. I za kraj, onda je F parcijalno rekurzivna po teoremu 7.22. \square

7.3. Postov teorem

Primijetimo samo da je korolar 3.62 specijalni slučaj teorema 7.26 (baš kao i teorema 3.61), zbog propozicije 7.2. Ipak, teorem 3.61 *nije* specijalni slučaj teorema 7.26, jer ima „inače” granu G_0 , za koju smo objasnili — barem intuitivno — zašto je ne možemo imati u rekurzivno prebrojivoj verziji.

Pokušajmo malo formalizirati to objašnjenje. U teoremima 3.61 i 2.44, „inače” se realizira kroz relaciju R_0 , koja je komplement unije svih ostalih R_i . Ako su sve R_i rekurzivno prebrojive, njihova unija je rekurzivno prebrojiva po propoziciji 7.8, pa zaključujemo da je jedini mogući problem u operaciji komplementa. Doista, pokazuje se da skup rekurzivno prebrojivih relacija nije zatvoren na komplement. Među rekurzivno prebrojivim relacijama svakako ima onih čiji su komplementi također rekurzivno prebrojivi — recimo, *rekurzivne* su takve; ali zapravo su to jedine takve relacije.

Intuicija je jasna: znamo da možemo čekati na više stvari istovremeno. Ako čekamo na dvije međusobno suprotne stvari, znamo da će se sigurno točno jedna dogoditi.

Teorem 7.27 (Post). *Neka je R brojevena relacija. Tada je R rekurzivna ako i samo ako su R i R^c obje rekurzivno prebrojive.*

Dokaz. Za smjer (\Rightarrow), neka je R rekurzivna. Tada je po propoziciji 2.37 R^c također rekurzivna. Sada su po propoziciji 7.2 obje rekurzivne relacije, R i R^c , rekurzivno prebrojive.

Smjer (\Leftarrow) je zanimljiviji. Pretpostavimo da su R i R^c rekurzivno prebrojive. Tada za funkciju χ_R vrijedi

$$\chi_R(\vec{x}) \simeq \begin{cases} 1, & \vec{x} \in R \\ 0, & \vec{x} \in R^c \end{cases}, \quad (7.20)$$

dakle (s k smo označili mjesnost od R) $\chi_R^k = \{R: C_1^k, R^c: C_0^k\}$ je parcijalno rekurzivna po teoremu 7.26 i propoziciji 2.19 — očito su R i R^c disjunktne.

No karakteristična funkcija χ_R je uvijek totalna, pa iz njene parcijalne rekurzivnosti zapravo slijedi da je rekurzivna, odnosno R je rekurzivna relacija. \square

Postov teorem pruža dobru intuiciju o tome zašto pojam koji rekurzivna prebrojivost formalizira zovemo *poluodlučivost*: figurativno, ako je problem „napola odlučiv” s jedne strane, i „napola” sa suprotne strane, onda je zapravo potpuno odlučiv.

Pomoću Postovog teorema možemo dokazati da razni skupovi nisu rekurzivno prebrojivi. Zapravo, kad god imamo skup koji nije rekurzivan (a imamo ih hrpu po Riceovom teoremu, na primjer), znamo da on ili njegov komplement (ili nijedan od njih) nije rekurzivno prebrojiv. Ipak, za samu egzistenciju skupova koji nisu rekurzivno prebrojivi ne treba nam Postov teorem — to možemo i kardinalnim argumentom. Zato najčešće Postov teorem koristimo kad *znamo* da je jedan od ta dva skupa rekurzivno prebrojiv, pa onda zaključujemo da drugi nije.

Korolar 7.28. *Neka je R rekurzivno prebrojiva relacija koja nije rekurzivna.*

Tada R^c nije rekurzivno prebrojiva.

Dokaz. Ovo je samo obrat po kontrapoziciji Postovog teorema. \square

Primjer 7.29. Relacija K^c nije rekurzivno prebrojiva. \triangleleft

Mogli bismo postaviti pitanje: možemo li naći relaciju takvu da *niti* ona *niti* njen komplement nisu rekurzivno prebrojive? Pokušajte dokazati da relacija B^2 , zadana s $B(x, y) : \iff K(x) \leftrightarrow K(y)$, ima to svojstvo.

Poluodlučive relacije smo uveli kao *domene* izračunljivih funkcija, a onda smo vidjeli da ih možemo gledati (one mjesnosti barem 2, i to s funkcijskim svojstvom) kao *grafove* izračunljivih funkcija. Što je s onima mjesnosti 1 (podskupovima $S \subseteq \mathbb{N}$)? Pokazuje se da na njih možemo gledati kao na *slike* izračunljivih funkcija.

Povijesno, upravo tako ih je uveo Emil Leon Post [10], i zato su nazvani *rekurzivno prebrojivima*: to su oni skupovi koji se mogu rekurzivno prebrojati (enumerirati), dakle zapisati u obliku $S = \{a_0, a_1, a_2, \dots\}$, gdje je preslikavanje $n \mapsto a_n$ rekurzivno. Ako to preslikavanje (s domenom \mathbb{N}) označimo s a , očito je $S = J_a$. Ipak, u međuvremenu se terminologija malo pomaknula, pa s tom intuicijom danas postoje dva problema.

Prvo, rekurzivnost danas podrazumijeva totalnost, a svaka totalna brojevena funkcija ima nepraznu domenu, pa mora imati i nepraznu sliku. Htjeli bismo da prazan skup

\emptyset^1 također bude rekursivno prebrojiv, pa ćemo ga morati odvojiti kao specijalni slučaj za totalne funkcije.

Drugo, enumeracija često podrazumijeva injektivnost, bez ponavljanja elemenata — no na taj način, naravno, možemo dobiti samo *prebrojive* skupove. Konačne skupove bismo također htjeli zvati rekursivno prebrojivima (jer su rekursivni — korolar 2.46), pa ćemo morati dopustiti ponavljanja.

Štoviše, kao u teoremu 7.7, enumeracija će moći biti *primitivno* rekursivna — do na izuzetak praznog skupa, kao što smo već rekli.

Teorem 7.30 (Teorem enumeracije). *Neka je R jednomjesna brojevena relacija ($R \subseteq \mathbb{N}$). Tada su sljedeće tvrdnje ekvivalentne:*

- (1) R je rekursivno prebrojiva;
- (2) R je slika neke parcijalno rekursivne funkcije;
- (3) R je slika neke primitivno rekursivne funkcije, ili $R = \emptyset$.

Dokaz. Kao i u teoremu 7.7, da (3) povlači (2) je trivijalno: svaka primitivno rekursivna funkcija je parcijalno rekursivna, a \emptyset^1 je slika parcijalno rekursivne funkcije \emptyset^1 (primjer 2.29).

Da (2) povlači (1) je malo kompliciranije, ali i dalje sasvim jasno koristeći ono što znamo o projekcijama: naime, $y \in \mathcal{I}_F$ znači $(\exists \vec{x} \in \mathcal{D}_F)(y = F(\vec{x}))$, odnosno $\exists \vec{x} \mathcal{G}_F(\vec{x}, y)$. Htjeli bismo tu egzistencijalnu kvantifikaciju prikazati kao višestruku projekciju, ali za to nam y treba biti na prvom mjestu. Definiramo $P(y, \vec{x}) := \mathcal{G}_F(\vec{x}, y)$ — tada je očito $P \preceq \mathcal{G}_F$, a po teoremu 7.18 je \mathcal{G}_F rekursivno prebrojiv, pa je po lemi 7.4 i P rekursivno prebrojiva. Sada je $\mathcal{I}_F(y) \iff \exists \vec{x} P(y, \vec{x})$, što je rekursivno prebrojivo po korolaru 7.17.

Opet je najzanimljiviji dio da (1) povlači (3). Ovdje možemo učiniti nešto bolje od oponašanja dokaza teorema 7.7 — možemo *iskoristiti* taj teorem da se dočepamo primitivne rekursivnosti: postoji primitivno rekursivna relacija P^2 takva da je $R = \exists_* P$. To znači da vrijedi $R(x) \iff \exists y P(x, y)$, i već smo vidjeli u prethodnom odlomku da projekcije možemo zamijeniti dodatnim argumentima. Dakle, htjeli bismo definirati $G(x, y) := x$ u slučaju da vrijedi $P(x, y)$ — no što ako ne vrijedi? Pa, ako *nikada* ne vrijedi, to zapravo znači da je $R = \emptyset$, te nemamo što dokazivati. Ako pak $R \neq \emptyset$, tada postoji neki fiksni (recimo najmanji) element $r \in R$, pa ga možemo iskoristiti kao *joker* u slučajevima kada ne vrijedi $P(x, y)$. Sve u svemu, funkcija G zadana s

$$G(x, y) := \begin{cases} x, & P(x, y) \\ r, & \text{inače} \end{cases} \quad (7.21)$$

(simbolički $G = \{P: I_1^2, C_r^2\}$) je primitivno rekursivna po teoremu 2.44, i tvrdimo $R = \mathcal{I}_G$.

Za (\subseteq), neka vrijedi $x_0 \in R = \exists_* P$. To po definiciji znači da postoji $y_0 \in \mathbb{N}$ takav da vrijedi $P(x_0, y_0)$. No tada je po (7.21) $G(x_0, y_0) = x_0 \in \mathcal{I}_G$.

Za (\supseteq) , neka je $z \in \mathcal{I}_G$ proizvoljan. To znači da postoji $(x_0, y_0) \in \mathbb{N}^2$ takav da vrijedi $G(x_0, y_0) = z$. Sada imamo dva slučaja: ako vrijedi $P(x_0, y_0)$, tada zaključujemo da postoji y (konkretno, y_0) takav da vrijedi $(x_0, y) \in P$, odnosno $x_0 \in \exists_* P = R$. No ako vrijedi $\neg P(x_0, y_0)$, tada je $z = G(x_0, y_0) = x_0 \in R$.

Ako pak $\neg P(x_0, y_0)$, tada je trivijalno $z = G(x_0, y_0) = r \in R$. \square

Napomena 7.31. Enumeracija G koju smo konstruirali je dvomjesna: ako baš želimo *niz*, možemo ga definirati kontrakcijom: $F(n) := G(\text{fst}(n), \text{snd}(n))$. Tehnikama kao u dokazu leme 7.12 (samo za funkcije umjesto relacija), lako se vidi $\mathcal{I}_F = \mathcal{I}_G$ — inkluzija (\subseteq) slijedi direktno iz definicije od F , a (\supseteq) iz $G(x, y) = F(\text{pair}(x, y))$.

Simbolički, $F = G \circ (\text{fst}, \text{snd})$, a $G = F \circ \text{pair}$. Općenito za svaku kompoziciju vrijedi $\mathcal{I}_{H \circ (G_1, \dots, G_l)} \subseteq \mathcal{I}_H$ — raspišite detalje! \triangleleft

7.4. Rekurzivno prebrojivi jezici

Rekurzivno prebrojive brojevnje relacije definirali smo u svrhu formaliziranja ideje poluodlučivosti, u brojevnom modelu. Prirodno je zapitati se kako bi ta formalizacija izgledala u jezičnom modelu. Znamo da tamo relacijama odgovaraju jezici, pa se zapravo pitamo: što znači da je jezik L rekurzivno prebrojiv?

Prvo, na početku točke 4.4.1 definirali smo kodiranje jezika $\langle L \rangle$, što je jednomjesna relacija, i rekli smo da svojstva izračunljivosti jezika L možemo gledati kroz analogna svojstva brojevnje relacije $\langle L \rangle$. Dakle, možemo reći da je L rekurzivno prebrojiv ako je $\langle L \rangle \subseteq \mathbb{N}$ rekurzivno prebrojiv.

Drugo, možemo prevesti samu definiciju. Kako su rekurzivno prebrojive relacije domene izračunljivih brojevnih funkcija, rekurzivno prebrojivi jezici bili bi domene Turing-izračunljivih jezičnih funkcija.

Treće, sama ta definicija znači da postoji RAM-stroj čije izračunavanje s \vec{x} stane ako i samo ako je \vec{x} element relacije za koju tvrdimo da je rekurzivno prebrojiva. Analogni iskaz u jezičnom modelu — postoji Turingov stroj \mathcal{T} takav da \mathcal{T} -izračunavanje s w stane ako i samo ako je $w \in L$ (kažemo da \mathcal{T} *prepoznaje* L) — obično se u teoriji formalnih jezika uzima za definiciju rekurzivno prebrojivih jezika.

I četvrto, analogon jezikâ u brojevnom modelu su zapravo *jednomjesne* relacije, pa ih možemo shvatiti kao slike, odnosno enumeracije nekih izračunljivih funkcija. Formalizacija toga u jezičnom modelu vodi na pojam *Turingovih enumeratora*, koji imaju dvije trake: radnu i izlaznu, te umjesto završnog stanja imaju *izlazno* stanje q_p . Enumerator nema ulaza (obje trake su na početku prazne) i očito radi beskonačno dugo (nema završnog stanja), te kažemo da *nabraja* jezik svih riječi koje se nalaze na izlaznoj traci u trenucima u kojima se enumerator nađe u stanju q_p .

Nakon svega što ste naučili, ne biste trebali biti začuđeni time da sva ta četiri pristupa karakteriziraju istu klasu jezika.

Teorem 7.32. *Neka je Σ abeceda, i $L \subseteq \Sigma^*$ jezik nad njom.*

Tada su sljedeće tvrdnje ekvivalentne:

- (\mathcal{R}) skup $\langle L \rangle$ je rekurzivno prebrojiv;
- (\mathcal{D}) L je domena neke Turing-izračunljive jezične funkcije φ ;
- (\mathcal{T}) postoji Turingov stroj koji prepoznaje L ;
- (\mathcal{E}) postoji Turingov enumerator koji nabraja L .

Skica dokaza. Pokazujemo samo osnovne ideje.

- (\mathcal{R}) \Rightarrow (\mathcal{D}) Po definiciji, postoji parcijalno rekurzivna funkcija F^1 takva da je $\mathcal{D}_F = \langle L \rangle$. Po korolaru 4.47, jezična funkcija $\varphi := \mathbb{N}^{-1}F$ je Turing-izračunljiva, i domena joj je $\mathcal{D}_\varphi = \{w \in \Sigma^* \mid \langle w \rangle \in \langle L \rangle\} = L$.
- (\mathcal{D}) \Rightarrow (\mathcal{T}) Neka je $L = \mathcal{D}_\varphi$, i \mathcal{T} Turingov stroj koji računa φ . Po definiciji 4.5, imamo da \mathcal{T} -izračunavanje s w stane ako i samo ako je $w \in \mathcal{D}_\varphi = L$ — što po definiciji znači da \mathcal{T} prepoznaje L .
- (\mathcal{T}) \Rightarrow (\mathcal{E}) Ovo je relativno standardni dokaz koji se napravi u teoriji formalnih jezika, iako je teško raspisati sve detalje. Osnovna ideja je emulirati paralelizaciju na Turingovom stroju. Skicu dokaza možete vidjeti u [12, theorem 3.21].
- (\mathcal{E}) \Rightarrow (\mathcal{R}) Efektivno, treba provesti čitav postupak iz točke 4.2.2 za enumerator $\mathcal{E} = (Q, \Sigma, \Gamma, \sqcup, \delta_2, q_0, q_p)$. Umjesto q_z , sada q_p kodiramo s 1 (vrijedi analogon leme 4.16). Funkcija $\delta_2: Q \times \Gamma^2 \rightarrow Q \times \Gamma^2 \times \{-1, 1\}^2$ (jer enumerator ima dvije trake) je malo kompliciranija za kodirati, ali iste ideje kao u dokazu leme 4.20 (proširenje konačne funkcije nulom, korolar 2.48) prolaze.

Dobijemo pet jednomjesnih (primaju samo broj koraka, jer enumerator nema ulaz) funkcija `State`, `WorkPosition`, `WorkTape`, `OutPosition` i `OutTape`, definiranih degeneriranom simultanom primitivnom rekurzijom (vrijedi analogon propozicije 3.19 za $k = 0$, i dokaže se isto kao propozicija 2.20 i korolar 2.34 — uvođenjem *dummy* argumenta) iz primitivno rekurzivnih funkcija, pa su primitivno rekurzivne. Sada definicija nabiranja enumeratorom kaže da je $w \in L$ (odnosno $\langle w \rangle \in \langle L \rangle$) ako i samo ako postoji korak n u kojem je `State`(n) = $\mathbb{N}Q(q_p) = 1$ i `OutTape`(n) = $\|w_{\sqcup} \dots\| = \text{Recode}(\langle w \rangle, b', b)$ (s b' i b smo označili broj znakova u Σ i Γ redom). Drugim riječima, vrijedi

$$t \in \langle L \rangle \iff \exists n (\text{State}(n) = 1 \wedge \text{OutTape}(n) = \text{Recode}(t, b', b)), \quad (7.22)$$

pa je $\langle L \rangle$ rekurzivno prebrojiv po teoremu 7.7. \square

Bibliografija

- [1] Joshua Bloch. *Extra, Extra – Read All About It: Nearly All Binary Searches and Mergesorts are Broken*. 2006. URL: <https://research.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html>.
- [2] Udi Boker i Nachum Dershowitz. *A Proof of the Church-Turing Thesis*. Siječanj 2005. URL: <http://www.cs.tau.ac.il/~nachumd/papers/Church-Turing-Theorem.pdf>.
- [3] Raymond Chen. *Why does the Windows calculator generate tiny errors when calculating the square root of a perfect square?* URL: <https://blogs.msdn.microsoft.com/oldnewthing/20160628-00/?p=93765>.
- [4] Edsger Wybe Dijkstra. „Why numbering should start at zero”. Kolovoz 1982. URL: <http://www.cs.utexas.edu/users/EWD/ewd08xx/EWD831.PDF>.
- [5] Maarten van Emden. *How recursion got into programming: a tale of intrigue, betrayal, and advanced programming-language semantics*. 2014. URL: <https://vanemden.wordpress.com/2014/06/18/how-recursion-got-into-programming-a-comedy-of-errors-3/>.
- [6] James Huggins i Charles Wallace. *An Abstract State Machine Primer*. Teh. izv. Travanj 2003.
- [7] Sandro Lovnički. „Interpreter za λ -račun”. Diplomski rad. Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet, rujan 2018.
- [8] Eric Mazur. *The make-believe world of real-world physics*. Srpanj 2008. URL: https://youtu.be/8dU_WNf8Wg0.
- [9] Ivan Posavčević. „Izračunljivost na skupovima \mathbb{Z} , \mathbb{Q} , \mathbb{R} i \mathbb{C} ”. Diplomski rad. Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet, rujan 2016. URL: <http://digre.pmf.unizg.hr/5277/1/diplomski.pdf>.
- [10] Emil Leon Post. „Recursively enumerable sets of positive integers and their decision problems”. *Bulletin of the American Mathematical Society* 50.5 (1944), str. 284–316. URL: <https://goo.gl/nJSynP>.
- [11] Joseph Robert Shoenfield. *Recursion theory*. Lecture notes in logic. Springer-Verlag, 1993. ISBN: 9783540570936.

- [12] Michael Sipser. *Introduction to the Theory of Computation*. 3. izdanje. Boston, MA: Course Technology, 2013. ISBN: 113318779X.
- [13] Raymond Merrill Smullyan. *Godel's Incompleteness Theorems*. Oxford Logic Guides. Oxford University Press, 1992. ISBN: 9780195364378.
- [14] Robert Irving Soare. „Computability and Recursion”. *Bulletin of Symbolic Logic* 2.3 (1996), str. 284–321. DOI: [10.2307/420992](https://doi.org/10.2307/420992). URL: <http://www.people.cs.uchicago.edu/~soare/History/handbook.pdf>.
- [15] Alan Mathison Turing. „On computable numbers, with an application to the Entscheidungsproblem”. *Proceedings of the London mathematical society* 2.1 (1937), str. 230–265.
- [16] Mladen Vuković. *Izračunljivost — predavanja*. 2015.
- [17] Mladen Vuković. *Matematička logika*. Element, 2009. ISBN: 978-953-197-519-3.
- [18] Mladen Vuković. *Teorija skupova — predavanja*. Siječanj 2015.
- [19] *Why is the copying instruction usually named MOV?* 2013. URL: <https://softwareengineering.stackexchange.com/questions/222254/why-is-the-copying-instruction-usually-named-mov>.