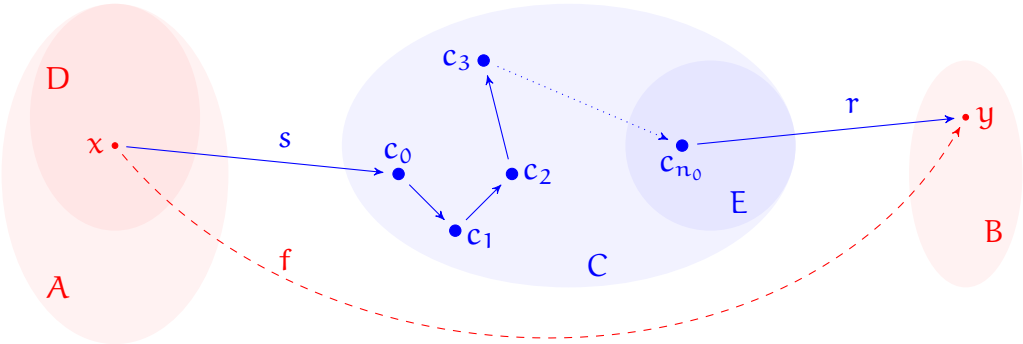
K O M P U T O N O M I K O N



Izračunljivost za računarce

Vedran Čačić

25. rujna 2022.

Sadržaj

Predgovor

Već nekoliko godina na Matematičkom odsjeku Prirodoslovno-matematičkog fakulteta u Zagrebu držim kolegij Izračunljivost — izvorno uveden kao prirodni nastavak kolegija Matematička logika (čijim je dijelom dugo bio), s ciljem dokaza Churchova teorema o neodlučivosti logike prvog reda. Tako je nastala knjiga [VukIzr15], izdvajanjem iz knjige [VukML09], prvenstveno namijenjene studentima teorijske matematike koji žele produbiti svoje znanje o matematičkoj logici.

U međuvremenu, zbog raznih okolnosti, kolegij su počeli upisivati mahom studenti računarstva, kojima je pojam algoritma daleko općenitiji i intuitivno bliži od onog potrebnog za dokaz Churchova teorema. 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 (implementirane u nekom programskom jeziku) počeo je već u umovima studenata računarstva istiskivati skupovnoteorijsku ideju uređene trojke $(\text{domena}, \text{kodomena}, \text{graf})$ kao asocijaciju na pojam „funkcija". Rekurzija više nije egzotična matematička konstrukcija, već sasvim uobičajen alat u repertoaru gotovo svakog programera. Jezici više nisu reprezentirani črčkarijama na papiru ili otiscima na traci (kao 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: 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 [VukIzr15]. Zahvaljujući njenom nastojanju da izgradi matematičku intuiciju — zanemarujući 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 [Maz08] 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 uočio da studenti pri programiranju koriste jedan mentalni model, a pri rješavanju zadataka sasvim drugačiji. I pritom naravno čine bitno više početničkih pogrešaka — jer taj drugi model izgrađuju tek nekoliko mjeseci, dok prvi izgrađuju desetak godina.

Knjiga koju čitate 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 [VukIzr15] je fantastična za izgradnju matematičke intuicije. Praktički jedini njen nedostatak je invalidacija računarske intuicije — ako tu intuiciju nemate, nedostatka nema. Trudio sam se održati kompatibilnost, tako da čak možete neke pojmove naučiti otamo, a neke odavdje.

Iako je knjigu sasvim moguće isprintati na papir i šarati po njoj, prvenstveno je namijenjena digitalnom čitanju. Zato ima mnogo 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 [Alt](+[Shift])+[←]. Pravi čitač omogućuje vam i navigaciju kroz naslove, bolji *rendering*, označavanje omiljenih stranica i još neke stvari zbog kojih ga se isplati instalirati. Moja preporuka je SumatraPDF na Windowsima, Okular na Linuxu te Document Viewer na Androidu. 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 LaTeXu, 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 $\mathcal{AMS}$ Euler i MnSymbol. 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. 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.

Ime ove knjige — *Komputonomikon* — relativno je doslovni prijevod njene svrhe: prikaz (εικόνα) zakona (νόμος) računanja (COMPVTVS). Kombiniranje grčkih i latinskih korijena omiljena je razonoda računaraca: promotrite primjer pridjeva „heksadecimalan”.

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

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

a. 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 skupova ...", i beskorisne propozicije poput „Postoji algoritam za sortiranje liste cijelih brojeva"? Dva su odgovora: 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 odavno mrtvom jeziku. Neke od tih knjiga [SW11] 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 [Emd14]. Može li se svaki rekurzivni algoritam zapisati nerekurzivno? Možemo li to dokazati? Također, ako implicitno vjerujemo da su svi programski jezici ekvivalentni, zašto cijelo vrijeme stvaramo nove? Razuman odgovor je da se razlikuju u *nečem drugom*, ne u algoritmima koje prezentiraju. Možemo li to *drugo* eliminirati, svodeći algoritme na „čistu esenciju"?

Drugi problem sastoji se u tome da programski jezici nastaju s raznim svrhama, ali izuzetno rijetko s primarnom svrhom vjerne reprezentacije matematičkih objekata — a 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 hardveru razne ustupke, koji se teško mogu matematički opravdati. Izuzetno je česta pojava, na primjer, da cijele brojeve računala ne reprezentiraju kao elemente prstena $\mathbb{Z}$, već prstena $\mathbb{Z}/2^{64}\mathbb{Z}$. Tako nastaje raskorak između algoritma i implementacije, koji ima važne praktične posljedice [Blo06]. Također, često se instrukcije ne izvršavaju redom kojim se nalaze u izvornom kodu, u svrhu bržeg izvršavanja na višejezgrenim procesorima — ponekad je čak sam pojam redoslijeda izvršavanja besmislen [LV16, str. 10].

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 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 sumnjati da su možda nerješivi.

Dok nitko nije dovodio u pitanje Euklidove konstrukcije, daleko preciznija formulacija geometrijske konstruktibilnosti bila je potrebna da se dokaže da je trisekcija kuta ravnalom i šestarom nemoguća. Dok je za pronalazak Cardanove ili Ferrarijeve formule bilo dovoljno znati nekoliko 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 se takvi dokazi za neke tvrdnje (kao što je hipoteza kontinuuma), niti za njihove negacije, ne mogu naći jer 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 (na primjer, 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 nultočku, nismo imali. Na slavnom Hilbertovu popisu 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 jednadžbe bile poznate tisućama godina) je Jurij Vladimirovič Matijasevič riješio deseti Hilbertov problem — očekivano, 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.

b. Predmet proučavanja

Cilj 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: reprezentirati njihove konfiguracije (ili njihove kodove) unutar svojih, i izvršavati korake njihova 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 Turingovu stroju. Tu tezu nije moguće formalno dokazati sve dok se ne dogovorimo oko općih aksioma izračunljivosti [BD05], 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 barem iste s obzirom na prirodno kodiranje ulaza i izlaza. Na

primjer, algoritmi za zbrajanje dekadski zapisanih i binarno zapisanih prirodnih brojeva su različiti, ali oba su zapravo samo reprezentacije brojevne funkcije add^2, s obzirom na različite zapise (dekadski odnosno binarni) samih prirodnih brojeva.

Također, jedan od tih modela — pa onda i svi ostali, simulacijom — je *univerzalan*: ne samo da svaka izračunljiva funkcija ima algoritam unutar tog modela, nego možemo naći *jedan* algoritam koji (ovisno o ulazima) može računati *sve* izračunljive funkcije, odnosno simulirati sve algoritme. Granica „dovoljne kompleksnosti" na kojoj se postiže opća izračunljivost i univerzalnost, za neke modele je začuđujuće nisko. Promotrit ćemo tri vrste takvih sustava: Turingove i RAM-strojeve te parcijalno rekurzivne funkcije.

c. Pregled po poglavljima

Slijedi okvirni prikaz rezultata obrađenih u pojedinim poglavljima 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 [Sho93]. Opisujemo tehniku spljoštenja (*inlining*) kojom se svaki 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 [Sip13]. Opisujemo Turing-izračunavanje parcijalno rekurzivnim funkcijama i transpiliramo RAM-strojeve u Turingove strojeve, čime dokazujemo ekvivalentnost brojevne i jezične 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 (posebni slučaj spljoštenja), teorem rekurzije (rješavanje 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.

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. 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 *determinističan*: 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 „Kakvi su algoritmi mogući?" postaje nešto preciznije pitanje „Koje su funkcije izračunljive?".

Da bismo funkciju mogli matematički zapisati, moramo precizirati 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 nespretan; 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 sljedbenik zadanog elementa. Nezgoda je i u tome što uobičajenim leksikografskim uređajem nije dobro uređen: na primjer, skup $\{0^n1 \mid n \in \mathbb{N}\}$ nema najmanji element.

Za dokazivanje teorema bolje je uzeti jednostavniji skup — u najvećem dijelu knjige promatrat ćemo *brojevne* algoritme, kojima su ulazni i izlazni podaci **prirodni brojevi**. U nekom smislu, skup prirodnih brojeva je najjednostavniji mogući skup na kojem se može obrađivati teorija izračunljivosti — svakako je najjednostavniji među beskonačnim skupovima, a izračunljivost na konačnim skupovima je trivijalna: algoritam za svaku funkciju se može napisati kao tablica (*lookup table*).

Odabir skupa $\mathbb{N}$ kao osnovnog isplatit će se kroz jednostavnost mnogih dokaza (jer imamo matematičku indukciju, jasan početak i sljedbenik, 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 zapisivanja

raznih objekata nizanjem znakova (Σ = ASCII): nitko nam ne mora objasniti kodiranje da bismo znali koji element od $\mathbb{Q}$ predstavlja *string* „-22/3".

Ipak, neintuitivnost kodiranja nadomjestit će jednostavnost algoritama: dok je, uz odgovarajuće tehnike [Pos16], 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), koristit ćemo njihovu jezičnu reprezentaciju, ali to će biti nakon što objasnimo općenito kodiranje sa Σ^* na $\mathbb{N}$ (i obrnuto).

Smatramo li nulu prirodnim brojem? Treba li brojenje početi od 0 ili od 1, dilema je stara koliko i samo računarstvo [EWD82]. 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 (pogledajte na primjer definiciju od $\{0,1\}^*$), skup s nulom imat će kraću oznaku.

$$\mathbb{N} := \{0, 1, 2, 3, \dots\} \tag{1.1}$$

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

Napomena 1.1: Govorili smo o izlaznim podacima u množini, no s obzirom na to da nas samo zanima postojanje algoritma, 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 $\mathbb{Z}$, vraćajući količnik i ostatak. Nju uvijek možemo, ako je nemamo kao osnovnu, emulirati pomoću dvije operacije, $/\!/$ i mod, koje vraćaju količnik i ostatak istog dijeljenja zasebno. Razlog zašto neki jezici imaju `divmod` kao posebnu funkciju je u tome da algoritmi za te dvije operacije imaju mnogo zajedničkih koraka — ako smo odredili količnik, obično možemo iz postupka kojim smo to učinili pročitati i ostatak (sjetite se npr. algoritma za dijeljenje višeznamenkastih 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 *koordinatne funkcije* $/\!/$ i mod, pa nam je dovoljno baviti se pitanjem jesu li $/\!/$ i mod izračunljive — u ovom slučaju, dakako, jesu. ◁

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 bitni dio identiteta algoritma. Primjerice, „zbroji dva broja" i „zbroji pet brojeva" su različiti algoritmi; štoviše, prvi od njih se pojavljuje kao korak (četiri puta) u drugome.

Posljedica toga je da u našem modelu ne postoje algoritmi s „varijabilnim brojem" ulaznih podataka, što je u računarstvu poznato pod nazivom *varargs*. Na nekoliko mjesta gdje nam budu potrebni, modelirat ćemo ih pomoću familije algoritama svih mogućih mjesnosti — recimo, zbrajanje kao $\mathsf{add}^k, k \in \mathbb{N}_+$. Mjesnost algoritma ili funkcije ćemo obično pisati u superskriptu ako je želimo naglasiti — neće dolaziti do zabune s eksponentima jer ni algoritme ni 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 $x_1, x_2, \ldots, x_k$ kad god 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 $f^7(\vec{x}, y, z)$, vidimo da 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 „pravi" argumenti 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. ◁

Pažljiv čitatelj će primijetiti da zahtijevamo da mjesnost bude pozitivna, 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 pojednostavljuje izlaganje, a takvi algoritmi nam nisu zanimljivi iz perspektive izračunljivosti: iz napomene 1.2 slijedi da nul-mjesni algoritmi mogu računati jedino konstante, a ne treba nam jako napredni formalizam da bismo zaključili da konstante *jesu* izračunljive.

1.1.1. Parcijalnost

Gdje god je dosad bilo govora o općenitim izračunljivim funkcijama, namjerno je upotrijebljen 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 je $\frac{\pi}{2} \in \mathbb{R} \setminus \mathcal{D}_{\mathrm{tg}}$. Takve funkcije označavamo oznakom $f : A \rightharpoonup B$, za razliku od *totalnih* funkcija koje označavamo $f : A \to B$ 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.

Napomena 1.3: 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. Naivna definicija algoritma, baš kao i naivna definicija skupa, dovodi do paradoksa (Russell). *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 u poglavlju 5, kad precizno definiramo pojmove. Napomenimo samo da je argument specijalni slučaj općeg načina zaključivanja „ne postoji objekt koji sam sebe nadvladava", koji se često koristi u raznim granama matematike [Tao09].

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 razlikovati algoritme, a iz teorije skupova znamo da je $\{0,1\}^*$ prebrojiv, dakle **svih algoritama ima prebrojivo mnogo**. Specijalno, svih jednomjesnih algoritama ima prebrojivo mnogo (očito ih ima beskonačno mnogo). Poredajmo ih sve u niz, i pogledajmo 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. Vrati $y + 1$.

Ako nacrtamo tablicu kojoj su retci jednomjesni algoritmi, a stupci ulazi za njih (prirodni brojevi), upravo opisani algoritam je klasična „promijenjena dijagonala" te tablice, koja se razlikuje od svakog njenog retka. Drugim riječima, *taj algoritam ne može biti na popisu* ako algoritmi nužno računaju totalne funkcije — ako je r-ti po redu, s ulazom r morao bi davati i y i $y + 1$ — ali s parcijalnim funkcijama nema kontradikcije, jer r-ti algoritam s ulazom r ne mora stati. $\triangleleft$

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 oblika njihovih definicija bit će jasno da algoritmi koji ih računaju uvijek stanu. Takve funkcije zvat ćemo *primitivno rekurzivnima*.

Recimo nekoliko riječi i o klasičnim izuzecima poput dijeljenja nulom. Primitivno rekurzivne funkcije moraju biti totalne pa si ne možemo priuštiti reći npr. „$3 \mathbin{/\!/} 0$ nije definirano" (dokazat ćemo da je $\mathbin{/\!/}$ primitivno rekurzivna operacija). U mnogim slučajevima to ćemo rješavati 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 $\vec{x} \notin \mathcal{D}_g$. Kazat ćemo tada da smo *parcijalno specificirali* (totalnu) funkciju f, u smislu: f je definirana svuda, ali nas zanimaju samo njene vrijednosti na nekom užem skupu.

U ovoj knjizi precizno ćemo pisati algoritme — ne pomoću pseudokoda, već formalnim konstrukcijama. Zato ćemo moći izračunati vrijednosti primitivno rekurzivne funkcije i izvan područja specifikacije — na primjer, algoritam za $\mathbin{/\!/}$ reći će nam da je $3 \mathbin{/\!/} 0 = 3$. Ponekad ćemo čak koristiti te „nedokumentirane značajke", jer će takvu funkciju biti lakše uklopiti u kasnije definicije bez rastava na slučajeve.

Iako sva računanja možemo shvatiti kao računanja funkcija, izlaganje je jednostavnije ako uvedemo i *relacije*, koje ćemo računati kao posebni slučaj računanja funkcija. Iz standardne skupovnoteorijske perspektive to se čini čudnim: nisu li relacije općenit pojam, a funkcije samo posebni 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 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 $\mathsf{isPrime} : \mathbb{N} \to \mathtt{bool}$, 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 relacije prirodno katkad zamišljati kao formule s relacijskim simbolima ($R(\vec{x})$, ili $x \, R \, y$ za dvomjesne relacije), a katkad kao skupove ($\vec{x} \in R$).

U skladu s uobičajenom praksom mnogih programskih jezika, prešutno koristimo standardno ulaganje skupa $\mathtt{bool}$ u $\mathbb{N}$, tako da preslikamo *false* $\mapsto 0$ i *true* $\mapsto 1$. Drugim riječima, na izračunljivost relacije R gledamo kao na izračunljivost njene *karakteristične funkcije* χ_R, koja ima istu mjesnost kao R. U suprotnom smjeru koristimo (opet standardnu) interpretaciju nule kao *false*, a svih ostalih prirodnih brojeva 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 karakteristične funkcije: na primjer, reći ćemo da je R primitivno rekurzivna ako je χ_R primitivno rekurzivna. Kod relacija ne moramo razmišljati o parcijalnosti: **karakteristične funkcije su uvijek totalne**. Relacije imaju drugi način modeliranja djelomične izračunljivosti, *rekurzivnu prebrojivost* o kojoj će biti više riječi u poglavlju 7.

1.2. Osnovni pojmovi i oznake

Dakle, promatrat ćemo (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 brojevna funkcija je oblika $f : \mathbb{N}^k \rightharpoonup \mathbb{N}$ za neki $k \in \mathbb{N}_+$. Skraćeno pišemo f^k i broj k zovemo *mjesnost* funkcije f. Svaka brojevna funkcija ima jedinstvenu mjesnost — osim prazne funkcije $\otimes$ s domenom $\varnothing$. Smatramo da i prazne funkcije imaju fiksnu mjesnost: umjesto jedne funkcije $\otimes$ promatramo familiju $\otimes^k, k \in \mathbb{N}_+$, proglašavajući na primjer $\otimes^3$ i $\otimes^8$ različitim funkcijama. Formalno to možemo napraviti tako da nam „funkcija" znači uređen par, kojem je prva komponenta uobičajena reprezentacija funkcije (skup uređenih parova ...), a druga komponenta mjesnost — ali nećemo biti toliko formalni, jer prazne funkcije nisu zanimljive iz perspektive izračunljivosti: računaju ih beskonačne petlje.

(Brojevna) 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 je želimo naglasiti. Kao i za funkcije, iako postoji samo jedan prazan skup, promatrat ćemo familiju $\varnothing^k, k \in \mathbb{N}_+$, smatrajući sve njene elemente različitim relacijama. Na kraju krajeva, njihove karakteristične funkcije *jesu* različite, jer imaju različite domene: recimo, $\mathcal{D}_{\chi_{\varnothing^3}} = \mathbb{N}^3$. (Radi se o nulfunkciji $C_0^3 : \mathbb{N}^3 \to \mathbb{N}$; razlikujte nulfunkciju, koja je totalna, od prazne funkcije koja nije definirana nigdje!)

Jezične funkcije ćemo uvijek definirati nad nekom *abecedom* (konačnim nepraznim skupom) Σ, kao funkcije $\varphi : \Sigma^* \rightharpoonup \Sigma^*$. Elementi od $\Sigma^* := \bigcup_{k \in \mathbb{N}} \Sigma^k$ su *riječi*: konačni nizovi *znakova* iz Σ, koje pišemo konkatenacijom — $\mathtt{aab}$ umjesto $(\mathtt{a}, \mathtt{a}, \mathtt{b})$. *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, abecedi dodajemo *separator* — znak koji služi razdvajanju argumenata. Recimo, višemjesne funkcije nad $\{\mathtt{a}, \mathtt{b}\}$ možemo reprezentirati kao jednomjesne funkcije nad $\{\mathtt{a}, \mathtt{b}, \mathtt{,}\}$ — tako da primjerice $\varphi^4(\mathtt{a}, \mathtt{abb}, \varepsilon, \mathtt{ba})$ računamo kao $\varphi^1(\mathtt{a,abb,,ba})$. Kažemo da je φ^1 dobivena *kontrakcijom* iz φ^4. Ovo je malo općenitije od brojevnih višemjesnih funkcija

jer možemo imati *varargs* (mjesnost možemo odrediti brojenjem separatora u ulaznoj riječi), ali i dalje nemamo nulmjesne funkcije: $\varphi(\varepsilon)$ je $\varphi^1(\varepsilon)$, ne $\varphi^0()$.

Analogon pojmu relacije u jezičnom slučaju je *jezik*: podskup od Σ^*. Iako karakteristična funkcija jezika nije ni brojevna ni jezična funkcija (ide sa Σ^* u $\mathtt{bool}$), svejedno možemo pomoću kodiranja skupa Σ^* reprezentirati i izračunljivost jezikā.

Domenu, sliku i graf funkcije f označavamo redom s $\mathcal{D}_f$, $\mathcal{I}_f$ i $\mathcal{G}_f$. To su relacije: za funkciju mjesnosti k, domena je mjesnosti k, slika je mjesnosti 1, a graf je mjesnosti $k+1$. 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 brojevna funkcija, oznakom $\tilde{f}$ označavamo njeno proširenje nulom: totalnu funkciju $\tilde{f} : \mathbb{N}^k \to \mathbb{N}$, 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 $[a{\cdot\cdot}b]$ ili $[a{\cdot\cdot}b\rangle$, gdje su $a, b \in \mathbb{N}$. Svaki takav interval skup je prirodnih brojeva iz odgovarajućeg realnog intervala: na primjer, $[1{\cdot\cdot}5\rangle = [1{\cdot\cdot}4] = \{1, 2, 3, 4\}$. Oznaka $A \uplus B$ označava uniju disjunktnih skupova (takvih da je $A \cap B = \varnothing$).

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 onog uređenog para iz skupa g čija je prva komponenta x), dok $\mathsf{g}(x)$ 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, $\mathsf{Divides}^2$, odnosno $\mathsf{isPrime}^1$.

Napomena 1.4: Algoritmiziranu jednakost (izračunljivu dvomjesnu brojevnu relaciju, u modernim programskim jezicima često označenu '==') označavamo uobičajenim simbolom '=' koji i inače koristimo za jednakost matematičkih objekata (funkcija, relacija, skupova, ...). Kod definicija skupova, i funkcija s prethodno specificiranom domenom (posebno kod totalnih funkcija), koristimo simbol ':='. Relacije definiramo formulama koristeći ':$\Longleftrightarrow$'. Često imamo potrebu vrijednosti funkcije specificirati izrazom, uz prešutnu pretpostavku „prirodne domene" (svi ulazni podaci za koje izraz ima vrijednost). Tada pišemo $f(\vec{x}) :\simeq \mathit{izraz}$. Ovisno o obliku izraza, definirat ćemo precizno značenje fraze „imati vrijednost". $\triangleleft$

Znak $\simeq$ između dva izraza znači da oba imaju vrijednost za iste vrijednosti varijabli koje se u njima pojavljuju, i da su im tada vrijednosti jednake. Drugačije rečeno, $\mathit{izraz1} \simeq \mathit{izraz2}$ znači da su definicije $f(\vec{x}) :\simeq \mathit{izraz1}$ i $f(\vec{x}) :\simeq \mathit{izraz2}$ ekvivalentne (definiraju istu funkciju), gdje su u $\vec{x}$ sve varijable koje se pojavljuju u bilo kojem od ta dva izraza. Razlog za izbjegavanje korištenja znaka $=$ u takvom slučaju sastoji se u tome što relacija $\simeq$, kao i svojstvo „imati vrijednost" na izrazima, **nisu izračunljive**. Još jedan razlog za korištenje neuobičajenog znaka je što mnoga „instinktivna pojednostavljenja" više nisu ispravna. 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 desni izraz ima vrijednost za sve $\vec{x} \in \mathbb{N}^3$, a lijevi 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. Radi se o RISC-arhitekturi sa samo tri tipa instrukcija (jedan od kojih je eliminabilan, ali zadržat ćemo ga radi jednostavnosti izlaganja).

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, a moguće ih je pozvati s proizvoljno velikim ulaznim podacima.

Pretpostavljamo da je program *fiksan*: ne može se mijenjati (tzv. *harvardska* arhitektura). Iako kōd 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 sposoban izvršavati razne programe (*von Neumannova* arhitektura). Da bismo počeli koristiti drugi operacijski sustav, dovoljno je instalirati ga i ponovo pokrenuti računalo; 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 i učiniti, ali tek u poglavlju 3. Krenimo s osnovnim definicijama.

Definicija 1.5: *RAM-stroj* je matematički (idealizirani) stroj, koji sadrži:

- *RAM-program*: fiksni konačni niz *instrukcija* $P := (I_0, I_1, I_2, \ldots, I_{n-1})$;
- *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 $[0 \cdot\cdot n]$. ◁

RAM-program najčešće pišemo kao $P := \begin{bmatrix} 0.\ I_0 \\ 1.\ I_1 \\ \vdots \\ (n-1).\ I_{n-1} \end{bmatrix}$, ili skraćeno $P := \begin{bmatrix} t.\ I_t \end{bmatrix}_{t<n}$.

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

Sadržaj registara se može mijenjati za vrijeme izvršavanja programa, u skladu s 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 1.6: Svaka *RAM-instrukcija* ima:

- (ako je dio RAM-programa P) *redni broj*, element skupa $[0 \cdot\cdot n_P\rangle$;
- *tip*, koji može biti: INC (*inkrement*), DEC (*dekrement*) ili GO TO (*skok*);
- (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 $[0 \cdot\cdot n_P]$. ◁

Dakle, RAM-instrukcija može biti jednog od tri oblika (s navedenim učincima):

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 1.7: Skup $\mathcal{I}ns$ svih RAM-instrukcija je prebrojiv.

Dokaz. Skup $\mathcal{I}ns_{\text{INC}}$ svih instrukcija tipa INC je prebrojiv: preslikavanje $f_1 : \mathbb{N} \to \mathcal{I}ns_{\text{INC}}$ 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 $\mathcal{I}ns_{\text{GO TO}}$ je prebrojiv, a skup $\mathcal{I}ns_{\text{DEC}}$ je ekvipotentan s $\mathbb{N} \times \mathbb{N}$, pa je i on prebrojiv. Sada je $\mathcal{I}ns$ prebrojiv kao (disjunktna) unija tih triju prebrojivih skupova. □

Korolar 1.8: Skup $\mathcal{P}rog$ svih RAM-programa je prebrojiv.

Dokaz. To slijedi iz činjenice da je $\mathcal{I}ns_{\text{INC}}{}^* \subseteq \mathcal{P}rog \subseteq \mathcal{I}ns^*$, i leme 1.7. Iz teorije skupova znamo da je skup A^* prebrojiv ako je A prebrojiv. □

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

Definicija 1.9: 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

$$c : \{\mathcal{R}_j \mid j \in \mathbb{N}\} \cup \{\text{PC}\} \to \mathbb{N} \tag{1.3}$$

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

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

1. c je završna ($pc = n_P$) i $c = d$ — to skraćeno pišemo $c \circlearrowleft$;
2. $I_{pc} = \text{INC}\ \mathcal{R}_j$ (za neki j), $r'_j = r_j + 1$, $pc' = pc + 1$ te $r'_i = r_i$ za sve $i \neq j$;

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

4. $I_{pc} = \text{DEC}\ \mathcal{R}_j, l$ (za neke j i l), $r_j = 0$, $pc' = l$ te $r'_i = r_i$ za sve i;

5. $I_{pc} = \text{GO TO}\ l$ (za neki l), $pc' = l$ te $r'_i = r_i$ za sve i. ◁

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 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 $pc' = l$.

Sada možemo formalizirati determinističnost RAM-stroja.

Lema 1.10: Svaka konfiguracija prelazi u neku, jedinstvenu, konfiguraciju.

Dokaz. Neka je $\mathcal{S}$ RAM-stroj s programom $(I_0, I_1, \ldots, I_{n-1})$ te $c = (r_0, r_1, \ldots, pc)$ njegova proizvoljna konfiguracija. Po definiciji je $pc \leq n$ — ako vrijedi jednakost, c je završna pa po pravilu 1 prelazi u samu sebe (nijedno drugo pravilo nije primjenjivo jer I_{pc} ne postoji). Ako je pak $pc < n$, pogledajmo tip od I_{pc}. Ako je to INC ili GO TO, pravilo 2 odnosno 5 točno propisuje novu konfiguraciju u koju c prelazi.

Inače, I_{pc} je tipa DEC, recimo $\text{DEC}\ \mathcal{R}_j, l$, i tada je opet nova konfiguracija jedinstveno određena, s obzirom na r_j. Ako je $r_j > 0$ („istina"), tada je primjenjivo samo pravilo 3, a ako je $r_j = 0$ („laž"), tada je primjenjivo samo pravilo 4; pravilo 3 nije primjenjivo jer po definiciji konfiguracije mora biti $r'_j \in \mathbb{N}$, a primjenom pravila 3 bismo dobili $r'_j = -1$. Svako od tih pravila također jednoznačno određuje novu konfiguraciju. □

Definicija 1.11: *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 $\vec{x} \in \mathbb{N}^k$. 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 brojevna 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 $(f(\vec{x}), \ldots, n_P)$;
- u suprotnom (ako $\vec{x} \notin \mathcal{D}_f$), P-izračunavanje s $\vec{x}$ ne stane. ◁

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

1.3.1. Skup $\mathcal{C}omp$

Navodimo tri lagane posljedice determinizma.

Propozicija 1.12: Za svaki RAM-algoritam P^k, za svaki ulaz $\vec{x} \in \mathbb{N}^k$, postoji jedinstveno P-izračunavanje s $\vec{x}$.

Dokaz. Za postojanje, induktivno definiramo

$$c_0 := \text{početna konfiguracija s ulazom } \vec{x}, \tag{1.4}$$

$$c_{n+1} := \text{jedinstvena konfiguracija u koju } c_n \text{ prelazi (prema lemi 1.10).} \tag{1.5}$$

Po Dedekindovu 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 1.10. □

Propozicija 1.13: 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 s i_1 i i_2 indekse na kojima se one prvi put pojavljuju. 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

$$c_{i_1} = c_{i_1+1} = c_{i_1+2} = \cdots = c_{i_2}, \tag{1.6}$$

kontradikcija. □

Korolar 1.14: Svaki RAM-algoritam računa jedinstvenu brojevnu funkciju.

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

$$S := \{\vec{x} \in \mathbb{N}^k \mid \text{P-izračunavanje s } \vec{x} \text{ stane}\} \tag{1.7}$$

i na tom skupu funkciju

$$f(\vec{x}) := c(\mathcal{R}_0), \text{ gdje je } c \text{ završna konfiguracija P-izračunavanja s } \vec{x}. \tag{1.8}$$

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

Za jedinstvenost, mjesnost funkcije je određena mjesnošću algoritma (uz prethodni dogovor da se prazne funkcije različitih mjesnosti razlikuju), njena domena je određena stajanjem izračunavanja (jedinstvenog zbog propozicije 1.12), a vrijednost funkcije u svakoj točki domene određena je završnom konfiguracijom (jedinstvenom zbog propozicije 1.13). □

Važna posljedica prethodnog rezultata je ograničenje broja izračunljivih funkcija.

Definicija 1.15: Neka je $k \in \mathbb{N}_+$ te f^k brojevna funkcija. Kažemo da je f^k *RAM-izračunljiva* ako postoji RAM-algoritam P^k koji je računa. Za svaki $k \in \mathbb{N}_+$, oznakom $\mathcal{C}\mathrm{omp}_k$ označavamo skup svih RAM-izračunljivih funkcija mjesnosti k. ◁

Oznaka $\mathcal{C}omp_k$ namjerno ne spominje RAM-model izračunavanja — pokazat ćemo da se *isti* skup brojevnih funkcija dobije i u drugim modelima koje ćemo razmatrati.

Teorem 1.16: Za svaki $k \in \mathbb{N}_+$, skup $\mathcal{C}omp_k$ je prebrojiv. Skup $\mathcal{C}omp$ svih RAM-izračunljivih brojevnih funkcija (svih mjesnosti) je također prebrojiv.

Dokaz. Neka je k fiksna mjesnost. Preslikavanje sa skupa $\mathcal{P}rog$ na skup $\mathcal{C}omp_k$, koje svakom RAM-programu P pridružuje funkciju koju algoritam P^k računa, je dobro definirano prema korolaru 1.14, i surjekcija je po definiciji 1.15. Iz toga je $\operatorname{card}(\mathcal{C}omp_k) \leq \operatorname{card}(\mathcal{P}rog)$, što je $\aleph_0$ po korolaru 1.8.

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

$$P_n^k := \left[\ t.\ \textsc{inc}\ \mathcal{R}_0\ \right]_{t<n}^k = \begin{bmatrix} 0.\ \textsc{inc}\ \mathcal{R}_0 \\ 1.\ \textsc{inc}\ \mathcal{R}_0 \\ \vdots \\ (n-1).\ \textsc{inc}\ \mathcal{R}_0 \end{bmatrix}^k \tag{1.9}$$

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

Sada je $\mathcal{C}omp = \bigcup_{k \in \mathbb{N}_+} \mathcal{C}omp_k$ prebrojiv kao unija prebrojivo mnogo prebrojivih skupova. □

Korolar 1.17: Za svaki $k \in \mathbb{N}_+$ postoji brojevna funkcija mjesnosti k koja nije RAM-izračunljiva.

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

$$\operatorname{card}\left(\mathbb{N}^{\mathbb{N}^k}\right) = \aleph_0^{\aleph_0^k} = \aleph_0^{\aleph_0} = \mathfrak{c} > \aleph_0. \tag{1.10}$$

Iz toga i teorema 1.16 slijedi $\mathcal{F}unc_k \nsubseteq \mathcal{C}omp_k$, pa je $\mathcal{F}unc_k \setminus \mathcal{C}omp_k \neq \varnothing$. □

1.3.2. Primjeri RAM-programa

RAM-programe za konstantne funkcije vidjeli smo već u dokazu teorema 1.16. 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* $\varnothing^k$, odnosno njenu karakterističnu funkciju. Također, za $n = 1$, program $\left[\ 0.\ \textsc{inc}\ \mathcal{R}_0\ \right]$ računa univerzalnu relaciju $\mathbb{N}^k$.

Napišimo RAM-program koji računa praznu funkciju $\otimes^k$. Po definiciji, to je program čije izračunavanje ne stane ni s kojim ulazom. 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). Takav program je $\left[\ 0.\ \textsc{go to}\ 0\ \right]$.

Dosadašnji programi nisu uopće koristili svoje ulaze. Najjednostavnija funkcija koja koristi svoj argument je identiteta (označena s I_1^1). Da bismo je izračunali, moramo prebaciti

vrijednost iz ulaznog registra $\mathcal{R}_1$ u izlazni registar $\mathcal{R}_0$. Koristeći naše razumijevanje izvršavanja imperativnih programa, vidimo da to čini RAM-algoritam

$$P^1_{I_1} := \begin{bmatrix} 0.\ \textsc{dec}\ \mathcal{R}_1, 3 \\ 1.\ \textsc{inc}\ \mathcal{R}_0 \\ 2.\ \textsc{go to}\ 0 \end{bmatrix}^1 . \tag{1.11}$$

Formalno, mogli bismo dokazati da svaki prolaz kroz petlju (čitanje 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 $r_1' = r_1 - r_1 = 0 \wedge r_0' = r_0 + r_1$, pa iz početne konfiguracije s ulazom x, $(0, x, 0, \ldots, 0)$, dolazimo u završnu konfiguraciju $(x, 0, 0, \ldots, 3)$ s izlaznim podatkom x. Na primjer, za $x = 2$ imamo sljedeću „šetnju" kroz konfiguracije:

$$\begin{aligned}(0,2,0,\ldots,0) &\rightsquigarrow (0,1,0,\ldots,1) \rightsquigarrow (1,1,0,\ldots,2) \rightsquigarrow (1,1,0,\ldots,0) \rightsquigarrow \\ &\rightsquigarrow (1,0,0,\ldots,1) \rightsquigarrow (2,0,0,\ldots,2) \rightsquigarrow (2,0,0,\ldots,0) \rightsquigarrow (2,0,0,\ldots,3) \circlearrowleft .\end{aligned} \tag{1.12}$$

Ubuduće nećemo biti tako precizni (upravo jer imamo razvijen osjećaj za programiranje 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" (konkatenniramo) više takvih blokova za različite ulazne registre, možemo dobiti RAM-programe za zbrajanje. Konkretno, funkcija add^3 zadana s $\mathsf{add}(x, y, z) := x + y + z$ je RAM-izračunljiva, jer je računa RAM-algoritam

$$P^3_{\mathsf{add}^3} := \begin{bmatrix} 0.\ \textsc{dec}\ \mathcal{R}_1, 3 \\ 1.\ \textsc{inc}\ \mathcal{R}_0 \\ 2.\ \textsc{go to}\ 0 \\ 3.\ \textsc{dec}\ \mathcal{R}_2, 6 \\ 4.\ \textsc{inc}\ \mathcal{R}_0 \\ 5.\ \textsc{go to}\ 3 \\ 6.\ \textsc{dec}\ \mathcal{R}_3, 9 \\ 7.\ \textsc{inc}\ \mathcal{R}_0 \\ 8.\ \textsc{go to}\ 6 \end{bmatrix}^3 . \tag{1.13}$$

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 (1.11) za I^1_1, kao i kraj *tog dijela* programa za add^3. Primijetite sličnost sa standardnom konvencijom o `end`-iteratoru u biblioteci STL jezika C++.

Vidjeli smo da su mnoge funkcije (prazna, konstante, identiteta, zbrajanje, ...) RAM-izračunljive. Ipak, pisati RAM-programe može biti dosta zamorno (na primjer za add^8 — mnogi dijelovi se ponavljaju uz neznatne izmjene u odredištima ili adresama registara) ili komplicirano (na primjer za množenje, ili potenciranje — povremeno bismo htjeli iskoristiti registar kao brojač za neku petlju, ali istodobno i sačuvati njegovu vrijednost). Prvi problem riješit ćemo makroima, a drugi funkcijskim programiranjem u poglavlju 2.

1.4. Makro-izračunljivost

Izvršavanje RAM-programa na RAM-stroju, pored prevođenja početne konfiguracije (s ulazom $\vec{x}$) u završnu (s izlazom $f(\vec{x})$), proizvede mnoge „popratne učinke" (*side-effects*) na njegovim registrima. Te učinke možemo objediniti (*enkapsulirati*) tako da čitav RAM-program P shvatimo kao jednu instrukciju P^* nekog kompliciranijeg stroja.

Ta oznaka sugerira dualnu upotrebu RAM-programa P: ako ga koristimo radi računanja k-mjesne funkcije (k ulaznih registara), promatramo ga kao algoritam P^k, a ako ga koristimo radi djelovanja na registre (svi registri „ulazni"), promatramo ga kao *makro* P^*.

Primjerice, za svaki $j \in \mathbb{N}$, RAM-program $P_j := \begin{bmatrix} 0.\ \textsc{dec}\ \mathcal{R}_j, 2 \\ 1.\ \textsc{go to}\ 0 \end{bmatrix}$ 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* za resetiranje jednog registra bez promjene ostalih. Taj makro zovemo $\textsc{zero}\ \mathcal{R}_j$.

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. 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 1.18: *Makro-stroj* je matematički stroj koji sadrži:

- *makro-program*: fiksni konačni niz *makro-instrukcija* $Q := (I_0, I_1, \ldots, I_{n-1})$, 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 $ac \in [0 \cdot\cdot n_P]$;
  - inače ($pc = n_Q$, ili $I_{pc} \in \mathcal{I}ns$), $ac = 0$. ◁

Sada možemo, slično kao lemu 1.7 i korolar 1.8, dokazati da su skupovi

$$\mathcal{MI}ns := \mathcal{I}ns \uplus \{P^* \mid P \in \mathcal{P}rog\}, \text{ i} \tag{1.14}$$

$$\mathcal{MP}rog := \{Q \in \mathcal{MI}ns^* \mid \text{sva odredišta u } Q \text{ su manja ili jednaka } n_Q\}, \tag{1.15}$$

svih makro-instrukcija, i svih makro-programa, prebrojivi. Ti rezultati nisu toliko bitni zbog tehnika koje ćemo uskoro razviti, ali predstavljaju dobru vježbu.

Definicija 1.19: *Konfiguracija makro-stroja* s programom $Q = (I_0, I_1, \ldots, I_{n_Q-1})$, registrima $\mathcal{R}_j$, $j \in \mathbb{N}$ te programskim brojačima PC i AC je bilo koje preslikavanje $c : \{\mathcal{R}_j \mid j \in \mathbb{N}\} \cup \{\text{PC}, \text{AC}\} \to \mathbb{N}$, takvo da je $c^{-1}[\mathbb{N}_+]$ konačan skup, $c(\text{PC}) \leq n_Q$, i još vrijedi $c(\text{AC}) = 0$ — osim u slučaju $I_{c(\text{PC})} = P^*$, kada je $c(\text{AC}) \leq n_P$. Skraćeno pišemo $c = \big(c(\mathcal{R}_0), c(\mathcal{R}_1), \ldots, c(\text{PC}), c(\text{AC})\big)$.

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 $c(\text{PC}) = n_Q$ (tada mora biti $c(\text{AC}) = 0$, jer $I_{c(\text{PC})}$ uopće ne postoji). ◁

Definicija 1.20: Za konfiguracije $c = (r_0, r_1, \ldots, pc, ac)$ i $d = (r'_0, r'_1, \ldots, pc', ac')$ istog makro-stroja s makro-programom $Q = (I_0, I_1, \ldots, I_{n_Q-1})$, 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 ($pc = n_Q$) — još pišemo $c \circlearrowleft$;
2. $ac = ac' = 0$, I_{pc} je RAM-instrukcija, a RAM-konfiguracija $(r_0, r_1, \ldots, pc)$ nije završna ($pc < n_Q$) i prelazi u RAM-konfiguraciju $(r'_0, r'_1, \ldots, pc')$ po programu Q (odnosno njegovoj instrukciji s rednim brojem pc);
3. $pc' = pc$, I_{pc} je makro P^*, a RAM-konfiguracija $(r_0, r_1, \ldots, ac)$ nije završna ($ac < n_P$) i prelazi u RAM-konfiguraciju $(r'_0, r'_1, \ldots, ac')$ po programu P (odnosno njegovoj instrukciji s rednim brojem ac);
4. $pc' = pc+1$, $I_{pc} = P^*$, RAM-konfiguracija $(r_0, r_1, \ldots, ac)$ je završna ($ac = n_P$) i $ac' = 0$. ◁

Drugim riječima, makro-stroj funkcionira 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 $(P, (\mathcal{R}_j)_{j\in\mathbb{N}}, \text{AC})$ 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 postigne 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 $(r_0, r_1, \ldots, n_Q, 0)$, u kojoj će r_0 predstavljati izlazni podatak.

Napomena 1.21: 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 1.19 kaže da u tom slučaju svaka konfiguracija mora preslikavati AC u 0, a definicija 1.20, 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.) ◁

Primjer 1.22: Uzmimo RAM-program P_{add^3} iz algoritma (1.13), i promotrimo makro-stroj s programom

$$Q := \begin{bmatrix} 0.\ \textsc{zero}\ \mathcal{R}_1 \\ 1.\ [\ ]^* \\ 2.\ \textsc{dec}\ \mathcal{R}_2, 1 \\ 3.\ P^*_{\mathsf{add}^3} \end{bmatrix}. \tag{1.16}$$

Neki prijelazi između konfiguracija tog stroja su:

$$\begin{aligned} &(0,2,4,0,\ldots,0,0) \rightsquigarrow (0,1,4,0,\ldots,0,1) \rightsquigarrow (0,1,4,0,\ldots,0,0) \rightsquigarrow (0,0,4,0,\ldots,0,1) \rightsquigarrow \\ &\quad \rightsquigarrow (0,0,4,0,\ldots,0,0) \rightsquigarrow (0,0,4,0,\ldots,0,2) \rightsquigarrow (0,0,4,0,\ldots,1,0) \rightsquigarrow (0,0,4,0,\ldots,2,0) \rightsquigarrow \\ &\quad \rightsquigarrow (0,0,3,0,\ldots,3,0) \rightsquigarrow (0,0,3,0,\ldots,3,3) \rightsquigarrow (0,0,2,0,\ldots,3,4) \rightsquigarrow (1,0,2,0,\ldots,3,5) \rightsquigarrow \\ &\quad \rightsquigarrow (1,0,2,0,\ldots,3,3) \rightsquigarrow (1,0,1,0,\ldots,3,4) \rightsquigarrow (2,0,1,0,\ldots,3,5) \rightsquigarrow (2,0,1,0,\ldots,3,3) \rightsquigarrow \\ &\quad \rightsquigarrow (2,0,0,0,\ldots,3,4) \rightsquigarrow (3,0,0,0,\ldots,3,5) \rightsquigarrow (3,0,0,0,\ldots,3,3) \rightsquigarrow (3,0,0,0,\ldots,3,6) \rightsquigarrow \\ &\qquad \rightsquigarrow (3,0,0,0,\ldots,3,9) \rightsquigarrow (3,0,0,0,\ldots,4,0) \circlearrowleft . \end{aligned} \tag{1.17}$$

Također, polazeći od „čistog” makro-stroja sa svim registrima resetiranim, imamo niz

$$(0,\ldots,0,0) \rightsquigarrow (0,\ldots,0,2) \rightsquigarrow (0,\ldots,1,0) \rightsquigarrow (0,\ldots,2,0) \rightsquigarrow (0,\ldots,1,0) \rightsquigarrow \cdots \tag{1.18}$$

u kojem nijedna konfiguracija nije završna. ◁

Sada se, jednako kao za RAM-model, može definirati *makro-algoritam*, *makro-izračunavanje*, izreka „makro-algoritam *računa* funkciju” te pojam *makro-izračunljive* funkcije. Na primjer, niz (1.18) pokazuje da Q-izračunavanje s (0) ne stane, dok niz (1.17) pokazuje da Q-izračunavanje s $(2,4)$ stane s izlaznim podatkom 3. Također, makro-algoritam Q^4 računa funkciju $f : \mathbb{N} \times \mathbb{N}_+ \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, zadanu s $f(x,y,z,t) := y + z - 1$.

Kao i u RAM-slučaju, uz malo više tehnikalija mogu se 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, pa je skup makro-izračunljivih funkcija *jednak* skupu $\mathcal{C}\mathrm{omp}$ RAM-izračunljivih funkcija.

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 suštinski jedina razlika. Sadržaj registara makro-stroja i RAM-stroja bit će isti kako se krećemo kroz izračunavanje, i izvršavat će se iste instrukcije (istog tipa nad istim registrima) istim redom, samo će one biti u različitim programima, s različitim rednim brojevima, pa će njihova odredišta trebati biti drugačija kako bi se odnosila na odgovarajuće instrukcije u drugom programu.

Ideja konstrukcije: „spljoštimo” gornju razinu (na kojoj su makroi P^*) i donju razinu (na kojoj su pojedinačne instrukcije programa P) u jednu razinu. U računarstvu se ta 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. 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 ih treba popraviti, a jednako tako i odredišta koja se na njih odnose. Precizirajmo taj postupak.

Definicija 1.23: Neka je Q makro-program.
Spljoštenje od Q je RAM-program $Q^\flat$, 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 (ako ga ima), kojoj su redni broj i odredište (ako ga ima) povećani za i. ◁

Propozicija 1.24: Preslikavanje $^\flat$ je totalna surjekcija sa skupa $\mathcal{MP}rog$ na skup $\mathcal{P}rog$.

Dokaz. Za početak trebamo vidjeti da za proizvoljni makro-program Q, postupak iz definicije 1.23 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 $\mathcal{I}ns \subset \mathcal{MI}ns$ (dakle $\mathcal{P}rog \subset \mathcal{MP}rog$) te je $^\flat$ na RAM-programima identiteta: uvjet petlje već na početku nije ispunjen, pa se program uopće ne mijenja. Dakle za svaki RAM-program P vrijedi $P^\flat = P$. □

Primjer 1.25: Spljoštimo program Q iz primjera 1.22. Prvi makro u Q nalazi se odmah na početku ($i = 0$) pa ne moramo renumerirati instrukcije koje implementiraju ZERO $\mathcal{R}_1$, već samo one ispod njih: trebamo im povećati odredišta i redne brojeve za $2 - 1 = 1$. Nakon prvog prolaza kroz petlju tako dobijemo makro-program Q'.

Sljedeći makro je onaj koji odgovara praznom RAM-programu, s rednim brojem $i = 2$. 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 $(3.\ \text{DEC}\ \mathcal{R}_2, 2)$, odredište ostaje 2, dok se redni broj smanjuje za 1 i postaje također 2. Dobivamo Q''.

$$Q' := \begin{bmatrix} 0.\ \textsc{dec}\ \mathcal{R}_1, 2 \\ 1.\ \textsc{go to}\ 0 \\ 2.\ [\ ]^* \\ 3.\ \textsc{dec}\ \mathcal{R}_2, 2 \\ 4.\ P^*_{\mathsf{add}^3} \end{bmatrix} \qquad Q'' := \begin{bmatrix} 0.\ \textsc{dec}\ \mathcal{R}_1, 2 \\ 1.\ \textsc{go to}\ 0 \\ 2.\ \textsc{dec}\ \mathcal{R}_2, 2 \\ 3.\ P^*_{\mathsf{add}^3} \end{bmatrix} \tag{1.19}$$

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

$$Q''' := \begin{bmatrix} 0.\ \textsc{dec}\ \mathcal{R}_1, 2 \\ 1.\ \textsc{go to}\ 0 \\ 2.\ \textsc{dec}\ \mathcal{R}_2, 2 \\ 3.\ \textsc{dec}\ \mathcal{R}_1, 6 \\ 4.\ \textsc{inc}\ \mathcal{R}_0 \\ 5.\ \textsc{go to}\ 3 \\ 6.\ \textsc{dec}\ \mathcal{R}_2, 9 \\ 7.\ \textsc{inc}\ \mathcal{R}_0 \\ 8.\ \textsc{go to}\ 6 \\ 9.\ \textsc{dec}\ \mathcal{R}_3, 12 \\ 10.\ \textsc{inc}\ \mathcal{R}_0 \\ 11.\ \textsc{go to}\ 9 \\ \\ \end{bmatrix} \qquad \begin{array}{l} \nu(0,0) := 0 \\ \nu(0,1) := 1 \\ \nu(0,2) := \nu(1,0) := \nu(2,0) := 2 \\ \nu(3,0) := 3 \\ \nu(3,1) := 4 \\ \nu(3,2) := 5 \\ \nu(3,3) := 6 \\ \nu(3,4) := 7 \\ \nu(3,5) := 8 \\ \nu(3,6) := 9 \\ \nu(3,7) := 10 \\ \nu(3,8) := 11 \\ \nu(3,9) := \nu(4,0) := 12 \end{array} \tag{1.20}$$

i gotovi smo: $Q''' = Q^\flat$, jer više nema makroa u programu. (Za objašnjenje funkcije ν čije vrijednosti su napisane pored $Q^\flat$, pogledajte skicu dokaza teorema 1.27.) ◁

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 $\mathcal{MP}\mathrm{rog}$ i $\mathcal{P}\mathrm{rog}$ — 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 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. Na ovaj postupak ćemo se vratiti u formalnom okruženju (s kodiranim RAM-programima), pri dokazu teorema o parametru (propozicija 6.4). Tamo ćemo kodirati jedan specijalni slučaj spljoštenja, ali pokazat će se da je taj slučaj dovoljan za sve situacije u kojima nam spljoštenje formalno treba. 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.

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 1.26: 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. ◁

Teorem 1.27: Za svaki makro-program Q, RAM-program $Q^\flat$ je ekvivalentan s Q.

Skica dokaza. Potrebno je vidjeti da funkcija iz $\mathbb{N}^2$ u $\mathbb{N}$, zadana s

$$\nu(pc, ac) := \left(\sum_{i<pc} \begin{cases} n_P, & I_i = P^* \\ 1, & I_i \in \mathcal{J}ns \end{cases}\right) + ac, \qquad \text{za } pc \le n_Q, \quad ac \le \begin{cases} 0, & I_{pc} \in \mathcal{J}ns \vee pc = n_Q \\ n_P, & I_{pc} = P^* \end{cases}, \tag{1.21}$$

ima sljedeće ključno svojstvo: svaki prijelaz između RAM-konfiguracija $\left(r_0, r_1, \ldots, \nu(pc, ac)\right)$ i $\left(r'_0, r'_1, \ldots, \nu(pc', ac')\right)$ po programu $Q^\flat$ odgovara jednom ili više prijelaza između makro-konfiguracija $(r_0, r_1, \ldots, pc, ac)$ i $(r'_0, r'_1, \ldots, pc', ac')$ po programu Q. Intuitivno, funkcija ν opisuje kako se točno transformiraju redni brojevi instrukcija pri spljoštenju, a usto i preslikava „završnu konfiguraciju programskih brojača" $(n_Q, 0)$ u $n_{Q^\flat}$. Recimo, ako je $(i.\, P^*)$ prvi makro u Q, iz formule (1.21) slijedi da je $\nu(j, 0) := j$ za sve $j < i$. Na kraju primjera 1.25, u (1.20), navedena je funkcija ν za konkretni makro-program Q iz primjera 1.22.

Iz toga onda slijedi da se pri izvršavanju programa i njegova spljoštenja zapravo izvršavaju iste instrukcije, samo su im odredišta i redni brojevi transformirani po funkciji ν. Dakle, semantike tih instrukcija — promjene sadržaja registara — iste su 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 $Q^\flat$, pa će posebno i sadržaj registra $\mathcal{R}_0$ biti isti. Štoviše, jer je $\nu(n_Q, 0) = n_{Q^\flat}$, $Q^\flat$-izračunavanje s $\vec{x}$ će stati ako i samo ako Q-izračunavanje s $\vec{x}$ stane, i tada će u $\mathcal{R}_0$ biti isti broj. Kako je $\vec{x}$ bio proizvoljan, zaključujemo da su Q i $Q^\flat$ ekvivalentni. □

1.4.2. Primjeri makroa

Teorem 1.27 ima dvije važne posljedice. Prvu možemo uobličiti kao korolar.

Korolar 1.28: Neka je $k \in \mathbb{N}_+$ te 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 1.21 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 1.27, $Q^\flat$ je ekvivalentan s Q, dakle za svaki k pa posebno i za mjesnost funkcije f, Q^k i $(Q^\flat)^k$ (pišemo skraćeno $Q^{\flat\, k}$) računaju istu funkciju. Drugim riječima, RAM-algoritam $Q^{\flat\, k}$ računa funkciju f, pa je ona RAM-izračunljiva. □

Napomena 1.29: Druga posljedica teorema 1.27 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 1.27 smijemo se ponašati kao da P može biti i *makro*-program, koji koristi već napisane makroe. Formalno, pri tome mislimo na $P^{\flat *}$, koji ima istu semantiku što se učinka na registre tiče. ◁

Evo primjera korištenja te tehnike. Prisjetimo se: za svaki $j \in \mathbb{N}$,

$$(\textsc{zero}\ \mathcal{R}_j) := \begin{bmatrix} 0.\ \textsc{dec}\ \mathcal{R}_j, 2 \\ 1.\ \textsc{go to}\ 0 \end{bmatrix}^* \text{ ima semantiku } r'_j = 0. \tag{1.22}$$

Sličnom tehnikom kao za identitetu (1.11) vidimo da za sve *različite* $i, j \in \mathbb{N}$ makro

$$(\textsc{remove}\ \mathcal{R}_i\ \textsc{to}\ \mathcal{R}_j) := \begin{bmatrix} 0.\ \textsc{zero}\ \mathcal{R}_j \\ 1.\ \textsc{dec}\ \mathcal{R}_i, 4 \\ 2.\ \textsc{inc}\ \mathcal{R}_j \\ 3.\ \textsc{go to}\ 1 \end{bmatrix}^{\flat *} \tag{1.23}$$

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 — tada moramo 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

$$(\textsc{mmove}\ n\ \textsc{from}\ \mathcal{R}_i..\ \textsc{to}\ \mathcal{R}_j..) := \left[\ t.\ \textsc{remove}\ \mathcal{R}_{i+t}\ \textsc{to}\ \mathcal{R}_{j+t}\ \right]^{\flat *}_{t<n} \tag{1.24}$$

sa semantikom $(\forall t < n)(r'_{j+t} = r_{i+t} \wedge r'_{i+t} = 0)$ — koristimo $(\forall t < n)$ kao pokratu za $(\forall t \in [0 \cdot\cdot n\rangle)$. 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 funkcijā koje se trenutno izvršavaju. Pozivom funkcije, vrh stoga se pomiče, otvarajući novi okvir u kojem će se funkcija izvršavati. Povratkom iz funkcije, vrh stoga se vraća na staro mjesto, eliminirajući taj okvir tako da od njega ostane jedino povratna vrijednost.

Na RAM-arhitekturi nemamo stog poziva, ali ga možemo emulirati pomoću registara. Otvaranje okvira duljine b realizirat ćemo kao pomak prvih b registara za b mjesta udesno (od $\mathcal{R}_b$ do uključivo $\mathcal{R}_{2b-1}$), a njegovo zatvaranje kao pomak u suprotnom smjeru. Osigurat ćemo da je b uvijek dovoljno velik da time sačuvamo sve relevantne registre pozivatelja te da osiguramo dovoljno nulā na početku za sve relevantne registre pozvane funkcije — „uvjerivši" njen program da se izvršava na zasebnom RAM-stroju.

Stvarna računala ne implementiraju stog na taj način jer je takav pristup nevjerojatno rastrošan, u prostoru (broju registara) i u vremenu (broju koraka). No kako nas zanima samo postojanje algoritama, ne i njihova složenost, taj pristup će nam biti dovoljno dobar.

Upravo konstruirani makroi resetiraju registre koje prenose — što ako ih želimo sačuvati? Jedina usporedba koju imamo je ona s nulom u instrukciji tipa DEC, dakle jedini način da RAM-program sazna sadržaj registra je da ga dekrementira do nule. No 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

$$(\textsc{move}\ \mathcal{R}_i\ \textsc{to}\ \mathcal{R}_j\ \textsc{using}\ \mathcal{R}_k) := \begin{bmatrix} 0.\ \textsc{zero}\ \mathcal{R}_j \\ 1.\ \textsc{remove}\ \mathcal{R}_i\ \textsc{to}\ \mathcal{R}_k \\ 2.\ \textsc{dec}\ \mathcal{R}_k, 6 \\ 3.\ \textsc{inc}\ \mathcal{R}_i \\ 4.\ \textsc{inc}\ \mathcal{R}_j \\ 5.\ \textsc{go to}\ 2 \end{bmatrix}^{\flat *} \tag{1.25}$$

ima semantiku $r'_j = r_i \wedge r'_k = 0$ (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. REMOVE $\mathcal{R}_i$ TO $\mathcal{R}_k$	0	0	r_i
2.–5. (petlja)	r_i	r_i	0

(1.26)

Naglasimo da smo za tu operaciju morali „žrtvovati" (resetirati) jedan registar sa strane.

Zbog $i \neq k$, uvjet na parametre makroa REMOVE je zadovoljen.

Napomena 1.30: Vjerojatno smatrate čudnim naziv MOVE za ono što biste intuitivno zvali COPY (dok se ono što biste zvali MOVE ovdje zove REMOVE, a ono što biste zvali REMOVE ovdje se zove ZERO). Niste jedini: pogledajte recimo [Net13]. Razlozi su izgubljeni u dubinama povijesti, ali moderne računalne arhitekture uglavnom terminološki prate arhitekturu x86, koja instrukciju za kopiranje podataka između registara (i drugih lokacija) zove `mov`. Tu terminologiju i mi slijedimo ovdje. ◁

Svrha instrukcije MOVE je prijenos argumenata: kad pri funkcijskom pozivu otvorimo novi okvir za računanje pozvane funkcije, želimo u ulazne registre staviti argumente s kojima je pozvana. Također želimo da zatvaranjem tog okvira i vraćanjem kontrole pozivatelju registri s argumentima 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.

Neka je $k \in \mathbb{N}_+$ te $j_1, j_2, \ldots, j_k > k$ prirodni brojevi. Definiramo makro

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

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

1.4.3. Funkcijski makro

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

Prvo definiramo jedan korisni pojam. Kako svaka RAM-instrukcija djeluje na najviše jedan registar, 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 $i \geq m_P$. Može biti i $m_P = 0$, ako program uopće ne koristi registre (ako je prazan, ili se sastoji samo od instrukcija tipa GO TO).

Za makro-program Q, možemo prirodno definirati $m_Q := m_{Q^\flat}$ — 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 izračunavanja te funkcije na RAM-stroju 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 $m_{P^k} := \max\{m_P, k+1\}$. Zbog $k \in \mathbb{N}_+$, uvijek je $m_{P^k} \geq 2$.

Definicija 1.31: Neka je $k \in \mathbb{N}_+$, $f^k \in \mathcal{C}omp_k$ te P_f^k RAM-algoritam koji računa f^k. Neka su $m, j_0, j_1, \ldots, j_k \in \mathbb{N}$. Definiramo $b := 1 + \max\{m_{P_f} - 1, m - 1, k, j_0, j_1, \ldots, j_k\}$ te pomoću tog broja definiramo *funkcijski makro*

$$\left(P_f(\mathcal{R}_{j_1}, \ldots, \mathcal{R}_{j_k}) \to \mathcal{R}_{j_0} \text{ USING } \mathcal{R}_m..\right) := \begin{bmatrix} 0.\ \text{MMOVE } b \text{ FROM } \mathcal{R}_0.. \text{ TO } \mathcal{R}_b.. \\ 1.\ \text{ARGS } \mathcal{R}_{b+j_1}, \mathcal{R}_{b+j_2}, \ldots, \mathcal{R}_{b+j_k} \\ 2.\ P_f^* \\ 3.\ \text{REMOVE } \mathcal{R}_0 \text{ TO } \mathcal{R}_{b+j_0} \\ 4.\ \text{MMOVE } b \text{ FROM } \mathcal{R}_b.. \text{ TO } \mathcal{R}_0.. \end{bmatrix}^{\flat *}. \triangleleft \tag{1.28}$$

Propozicija 1.32: Semantika funkcijskog makroa, uz oznake iz definicije 1.31 te pokratu $\vec{r} := (r_{j_1}, r_{j_2}, \ldots, r_{j_k})$, opisana je sljedećim tvrdnjama:

1. Ako je $\vec{r} \in \mathcal{D}_f$, tada je $r'_{j_0} = f(\vec{r}) \wedge (\forall t \in [b \cdot\cdot 2b\rangle)(r'_t = 0)$.
   (Specijalno, zbog $b \geq m$, za sve $i \in [0 \cdot\cdot m\rangle \setminus \{j_0\}$ vrijedi $r'_i = r_i$.)

2. Ako $\vec{r} \notin \mathcal{D}_f$, izvršavanje funkcijskog makroa ne stane.

Dokaz. Prvo dokažimo da su zadovoljeni svi uvjeti na parametre korištenih makroa: 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 učinke pojedinih makro-instrukcija iz definicije 1.31.

Nakon prve instrukcije tipa MMOVE, prvih b registara bit će resetirano, a njihove stare vrijednosti $r_0, r_1, \ldots, r_{b-1}$ bit će u bloku $\mathcal{B}$ koji se sastoji od idućih b registara. Posebno, u $\mathcal{R}_{b+j_t}$ nalazit će se r_{j_t}, za sve $t \in [1 \cdot\cdot k]$.

Dakle, instrukcija ARGS će u ulazne registre $\mathcal{R}_1, \ldots, \mathcal{R}_k$ zapisati upravo vrijednosti $\vec{r}$. Ostale registre neće mijenjati, pa će $\mathcal{R}_0$ i dalje biti resetiran, kao i svi registri $\mathcal{R}_i, i \in \langle k \cdot\cdot b\rangle$, a u $\mathcal{B}$ će 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 1.21 slijedi da će imati isti učinak 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 (i neće promijeniti sadržaj bloka $\mathcal{B}$). Kako P_f^k računa funkciju f, a u „početnoj" konfiguraciji mu se u ulaznim registrima nalazi $\vec{r} \in \mathcal{D}_f$, 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 tipa REMOVE, koja će tu vrijednost $f(\vec{r})$ zapisati u registar $\mathcal{R}_{b+j_0}$, koji se nalazi u $\mathcal{B}$ zbog $j_0 < b$. Svi ostali registri iz $\mathcal{B}$ i dalje će sadržavati *backup* početnih vrijednosti prvih b registara. Ne znamo što će biti u prvih b registara (osim što će $\mathcal{R}_0$ biti resetiran) jer to ovisi o konkretnom programu P_f, ali zapravo to nije ni bitno.

Naime, zadnja instrukcija tipa MMOVE će čitav taj blok prepisati *backup*-blokom $\mathcal{B}$, vrativši prvih b registara na originalne vrijednosti (konkretno, zanimat će nas da se sačuva prvih $m \leq b$ registara), osim što će u $\mathcal{R}_{j_0}$ pisati vraćena vrijednost iz $\mathcal{R}_{b+j_0}$, dakle $f(\vec{r})$. $\mathcal{B}$ će time biti resetiran. To sve možemo prikazati tablicom:

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

. (1.29)

Tablica nije dovoljno precizna za sve mogućnosti: recimo, može biti $j_0 = 1$ ako želimo promijeniti $\mathcal{R}_1$ *in-place*. No zajedno s tekstnim opisom, 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, P_f-izračunavanje s $\vec{r}$ neće stati. To pak znači da rad makro-stroja, koji izvršava funkcijski makro, neće stati (zapet će u beskonačnoj petlji „na donjoj razini", izvršavajući instrukciju 2. P_f^*). Prema skici dokaza teorema 1.27, neće stati ni RAM-stroj koji izvršava spljoštenje tog makroa, što smo trebali dokazati. □

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

2. Rekurzivne funkcije

Iz uglavnom povijesnih razloga, prvi doticaj s programiranjem za većinu ljudi bude kroz *imperativno* programiranje: algoritmi kao *programi*, nizovi *naredaba* koje mijenjaju stanje neke zajedničke *memorije* nad kojom se izvršavaju. Mnogo *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 ostvaruje *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 se mogu shvatiti kao algoritmi 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, primjerice na prirodnim brojevima, ako ih znamo potencirati, množiti i zbrajati.

U toj paradigmi, umjesto praznog programa, imamo aksiom da je nulfunkcija izračunljiva. Umjesto instrukcije INC koja mijenja sadržaj registra na kojem djeluje, imamo funkciju *sljedbenika*, koja proizvodi novi prirodni broj: 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, konstanti C_2^2 i C_3^2 te koordinatnih projekcija I_1^2 i I_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 (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 određeni broj puta, i kao takve ne mogu biti beskonačne. To znači da algoritmi dobiveni primitivnom rekurzijom uvijek stanu, pa su *primitivno rekurzivne* funkcije koje oni računaju 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. Tehniku za to razvit ćemo tek u točki 6.2, a zasad ćemo samo uvesti jedan posebni oblik koji odgovara *minimizaciji* relacije — traženju najmanjeg prirodnog broja s nekim svojstvom. To neće nužno dati totalnu funkciju (na primjer, 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 *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 u [VukIzr15, dodatak]. Važniji rezultat, koji ćemo dokazati, je da se skup parcijalno rekurzivnih funkcija podudara sa skupom $\mathcal{C}\mathrm{omp}$ RAM-izračunljivih funkcija.

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

Definicija 2.1: *Inicijalne funkcije* su sljedeće:

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

Prvu stavku možemo interpretirati kao izračunljivost (karakteristične funkcije) prazne relacije $\varnothing^1$. Treća stavka zapravo govori da je identiteta $\mathrm{id}_{\mathbb{N}^*}$ izračunljiva, ako je razdvojimo po mjesnostima u familiju $\mathrm{id}^k, k \in \mathbb{N}$ pa u skladu s napomenom 1.1 svaku $\mathrm{id}^k = \mathrm{id}_{\mathbb{N}^k}$ prikažemo pomoću k koordinatnih funkcija $\mathsf{I}_n^k, n \in [1 \cdot\cdot k]$.

Napomena 2.2: Sve inicijalne funkcije su totalne: $\mathcal{D}_{\mathsf{Z}} = \mathcal{D}_{\mathsf{Sc}} = \mathbb{N}$, a $\mathcal{D}_{\mathsf{I}_n^k} = \mathbb{N}^k$. ◁

Vidimo da inicijalnih funkcija zapravo ima beskonačno (prebrojivo) mnogo, ali su samo triju mogućih tipova. Nulfunkcija i sljedbenik omogućavaju reprezentaciju brojeva kao konstantnih jednomjesnih funkcija, a koordinatne projekcije omogućavaju individualni rad sa svakim pojedinim argumentom funkcije prema njegovu 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 (algoritam $[\,]^1$) računa funkciju Z. Iz semantike instrukcije REMOVE vidimo da za sve $k \geq n \geq 1$ makro-algoritam $\left[\ 0.\ \textsc{remove}\ \mathcal{R}_n\ \textsc{to}\ \mathcal{R}_0\ \right]^k$ računa I_n^k. Također, makro-algoritam $\left[\begin{array}{l} 0.\ \textsc{remove}\ \mathcal{R}_1\ \textsc{to}\ \mathcal{R}_0 \\ 1.\ \textsc{inc}\ \mathcal{R}_0 \end{array}\right]^1$ računa Sc:

$$\begin{array}{l|cc} & \mathcal{R}_0 & \mathcal{R}_1 \\ & 0 & x \\ \hline 0.\ \textsc{remove}\ \mathcal{R}_1\ \textsc{to}\ \mathcal{R}_0 & x & 0 \\ 1.\ \textsc{inc}\ \mathcal{R}_0 & x+1 & 0 \end{array}. \quad \square \tag{2.1}$$

Korolar 2.4: Svaka inicijalna funkcija je RAM-izračunljiva.

Dokaz. Spljoštenja makro-programa iz dokaza propozicije 2.3 daju nam RAM-algoritme za inicijalne funkcije. □

2.1. Kompozicija

Kompozicija je intuitivno jednostavna operacija (kao i slijedno izvršavanje naredaba u imperativnim programima): izračunamo vrijednost jedne funkcije, i uvrstimo je u definiciju druge funkcije. No definicija je prilično tehnička.

Htjeli bismo reći: „kompozicija $H \circ G$ dviju izračunljivih funkcija, G^k i H^l, je izračunljiva funkcija". Da bi ta kompozicija bila dobro definirana, kodomena od G^k bi morala biti $\mathbb{N}^l$. U skladu s napomenom 1.1, to znači da imamo l izračunljivih koordinatnih funkcija, $G_1^k, \ldots, G_l^k$, svaka od kojih daje po jedan ulazni podatak za H^l. Zato komponiranje više nije nužno binarna operacija: pišemo $H \circ (G_1, \ldots, G_l)$.

Drugi tehnički detalj tiče se domene kompozicije. Promotrimo kompoziciju $F^1 := I_1^2 \circ (Z^1, \otimes^1)$. Ta funkcija je svakako izračunljiva, ali *što* je ona? Konkretno, koliko je $F(5)$? Na prvi pogled, $F(5) = I_1\big(Z(5), \otimes(5)\big) = Z(5) = 0$ — i tako za svaki ulaz, dakle $F = Z$. Na drugi pogled, da bismo izračunali $F(5)$, moramo izračunati $Z(5) =: y_1$, zatim „izračunati" $\otimes(5) =: y_2$, i na kraju izračunati $I_1(y_1, y_2) = y_1$. Tako algoritam za računanje F ne stane s ulazom 5 (niti s ikojim drugim ulazom), jer njegov drugi korak ne stane — dakle $F = \otimes$. 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 Haskella. Prednost je bogatija semantika (više izraza ima vrijednost), a i 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
```
(2.2)

Drugi pristup zove se *marljiva* evaluacija (*eager evaluation*) i podrazumijeva se 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
```
(2.3)

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, za dokaz ekvivalentnosti s RAM-izračunljivošću odnosno za konstrukciju kompajlera u RAM-programe), a poslije ćemo opisati kako simulirati lijenu evaluaciju kad

nam bude potrebna. Jedno od mjesta gdje će nam sigurno 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 $H \circ (G_1, \ldots, G_l)$ definirana u $\vec{x}$ ako je *svaka* G_i definirana u $\vec{x}$, a 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, …, G_l^k i H^l funkcije. Za funkciju F^k definiranu s

$$\mathcal{D}_F := \left\{ \vec{x} \in \bigcap_{i=1}^{l} \mathcal{D}_{G_i} \,\middle|\, \big(G_1(\vec{x}), G_2(\vec{x}), \ldots, G_l(\vec{x})\big) \in \mathcal{D}_H \right\}, \tag{2.4}$$

$$F(\vec{x}) := H\big(G_1(\vec{x}), G_2(\vec{x}), \ldots, G_l(\vec{x})\big), \text{ za sve } \vec{x} \in \mathcal{D}_F, \tag{2.5}$$

kažemo da je dobivena *kompozicijom* iz funkcija G_1, G_2, …, G_l i H.

Skraćeno pišemo $F := H \circ (G_1, G_2, \ldots, G_l)$.

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

U skladu s napomenom 1.4, izraze (2.4) i (2.5) zajedno skraćeno pišemo kao

$$F(\vec{x}) :\simeq H\big(G_1(\vec{x}), G_2(\vec{x}), \ldots, G_l(\vec{x})\big), \tag{2.6}$$

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

Dakle, kod komponiranja parcijalnih funkcija moramo voditi računa o domeni — ali dobro je znati da sāmo komponiranje ne može narušiti totalnost.

Lema 2.6: Neka su $k, l \in \mathbb{N}_+$ te neka su G_1^k, G_2^k, …, G_l^k i H^l funkcije.

Ako je H totalna, tada je $\mathcal{D}_{H \circ (G_1, \ldots, G_l)} = \bigcap_{i=1}^{l} \mathcal{D}_{G_i}$.

Dokaz. Tvrdnja slijedi iz (2.4), jer je uvjet $\big(G_1(\vec{x}), \ldots, G_l(\vec{x})\big) \in \mathcal{D}_H = \mathbb{N}^l$ uvijek ispunjen za $\vec{x} \in \bigcap_{i=1}^{l} \mathcal{D}_{G_i}$. □

Korolar 2.7: Ako je H^1 totalna funkcija, i G brojevna funkcija, tada je $\mathcal{D}_{H \circ G} = \mathcal{D}_G$.

Dokaz. Ovo je tvrdnja leme 2.6 za $l = 1$. □

Propozicija 2.8: Svaka kompozicija totalnih funkcija je totalna.

Dokaz. Ako je svaka G_i^k totalna, tada je $\mathcal{D}_{G_i} = \mathbb{N}^k$ za sve i, pa je po lemi 2.6 također i $\mathcal{D}_{H \circ (G_1, \ldots, G_l)} = \bigcap_{i=1}^{l} \mathbb{N}^k = \mathbb{N}^k$. □

Primjer 2.9: $C_2^3 = Sc \circ Sc \circ Z \circ I_1^3$. Doista, za sve $(x, y, z) \in \mathbb{N}^3$ je

$$\begin{aligned}(Sc \circ Sc \circ Z \circ I_1^3)(x, y, z) &= (Sc \circ Sc \circ Z)\big(I_1^3(x, y, z)\big) = (Sc \circ Sc \circ Z)(x) = \\ &= (Sc \circ Sc)\big(Z(x)\big) = (Sc \circ Sc)(0) = Sc\big(Sc(0)\big) = Sc(1) = 2 = C_2^3(x, y, z). \quad ◁\end{aligned} \tag{2.7}$$

Lema 2.10: Skup RAM-izračunljivih funkcija, $\mathcal{C}\mathrm{omp}$, 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, a zatim izračunamo H s tako dobivenim brojevima.

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

$$Q_F := \begin{bmatrix} 0.\ P_{G_1}(\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_k) \to \mathcal{R}_{k+1} \text{ USING } \mathcal{R}_{k+l+1}.. \\ 1.\ P_{G_2}(\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_k) \to \mathcal{R}_{k+2} \text{ USING } \mathcal{R}_{k+l+1}.. \\ \vdots \\ (l-1).\ P_{G_l}(\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_k) \to \mathcal{R}_{k+l} \text{ USING } \mathcal{R}_{k+l+1}.. \\ l.\ P_H(\mathcal{R}_{k+1}, \mathcal{R}_{k+2}, \ldots, \mathcal{R}_{k+l}) \to \mathcal{R}_0 \text{ USING } \mathcal{R}_{k+l+1}.. \end{bmatrix} \tag{2.8}$$

računa $F := H \circ (G_1, G_2, \ldots, G_l)$. Prema definiciji 1.11, neka je $\vec{x} \in \mathbb{N}^k$ proizvoljan. Ako je $\vec{x} \in \mathcal{D}_F$, prema definiciji (2.4) je za sve $i \in [1 \cdot\cdot\, l]$, $\vec{x} \in \mathcal{D}_{G_i}$, 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 1.32, svi ostali registri do $\mathcal{R}_{k+l}$ ostati sačuvani, odnosno računanje y_i neće uništiti one ranije izračunane, niti ikoji x_j potreban za računanje kasnijih. Uvjet $\vec{y}^l \in \mathcal{D}_H$, također iz (2.4), znači da će i izvršavanje zadnje 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
$0.\ P_{G_1} \ldots$	0	x_1	x_k	$G_1(\vec{x})$	0	0	0
$1.\ P_{G_2} \ldots$	0	x_1	x_k	$G_1(\vec{x})$	$G_2(\vec{x})$	0	0
$\vdots$							
$(l-1).\ P_{G_l} \ldots$	0	x_1	x_k	$G_1(\vec{x})$	$G_2(\vec{x})$	$G_l(\vec{x})$	0
$l.\ P_H \ldots$	$F(\vec{x})$	x_1	x_k	$G_1(\vec{x})$	$G_2(\vec{x})$	$G_l(\vec{x})$	0

. (2.9)

Ako $\vec{x} \notin \mathcal{D}_F$, prema (2.4) 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 1.32, 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 zadnjoj instrukciji, pa opet Q_F-izračunavanje s $\vec{x}$ neće stati. Sada prema teoremu 1.27 RAM-algoritam $Q_F^{\flat k}$ također računa F, pa je $F \in \mathcal{C}\mathrm{omp}$. □

2.2. Primitivna rekurzija

Iz dokaza leme 2.10 je jasno kako točno kompozicija odgovara slijednom izvršavanju naredbi. Ipak, već za funkcije poput zbrajanja nam trebaju *petlje*: 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 jednostavni oblik rekurzije, koji odgovara petljama čiji je broj ponavljanja zadan prije početka njihova izvršavanja. Tek kasnije ćemo vidjeti kako se u ovom modelu mogu rješavati opće rekurzije.

Za funkcionalni opis petlje moramo prvo opisati *inicijalizaciju* G, koja odgovara stanju prije početka izvršavanja petlje, a zatim *tijelo* (ponekad zvano *korak*) petlje kao funkciju H koja preslikava stanje na početku jednog prolaza kroz petlju, u stanje na kraju tog prolaza.

Programski, primjerice u Pythonu, ograničene petlje kakve promatramo imaju oblik:

```
z = ...  # inicijalizacija varijable z, u nekom kontekstu
for i in range(y): z = ...z...  # tijelo, ažurira z
```
(2.10)

Tijelo, pored konteksta $\vec{x}$ i prethodno izračunane vrijednosti z, obično može koristiti i kontrolnu varijablu (u kodu nazvanu `i`), koja broji koliko smo puta prošli kroz petlju.

Definicija 2.11: Neka je $k \in \mathbb{N}_+$ te neka su G^k i H^{k+2} totalne funkcije. Za funkciju F^{k+1} definiranu s

$$F(\vec{x}, 0) := G(\vec{x}), \tag{2.11}$$

$$F(\vec{x}, y+1) := H\big(\vec{x}, y, F(\vec{x}, y)\big), \text{ za sve } y \in \mathbb{N}, \tag{2.12}$$

kažemo da je dobivena *primitivnom rekurzijom* iz funkcija G i H.

Skraćeno pišemo $F := G \mathbin{\text{ꝑ}} H$. Smatramo da operator ꝑ 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 $G \mathbin{\text{ꝑ}} H \in \mathcal{F}$. ◁

Napomena 2.12: Iz Dedekindova teorema rekurzije [VukTS15] slijedi da je jednadžbama (2.11) i (2.12) zadana jedinstvena totalna funkcija. Također, primitivna rekurzija je *definirana* samo za totalne funkcije, pa se ne moramo baviti domenama od G, H i $G \mathbin{\text{ꝑ}} H$ (uvijek su jednake redom $\mathbb{N}^k$, $\mathbb{N}^{k+2}$ i $\mathbb{N}^{k+1}$ za neki $k \in \mathbb{N}_+$). ◁

Budući da ne promatramo brojevne funkcije s nula ulaznih podataka, funkcija G koja zadaje početni uvjet mora biti barem jednomjesna, a onda funkcija F dobivena primitivnom rekurzijom mora biti barem dvomjesna. Svakako bismo htjeli definirati i jednomjesne funkcije primitivnom rekurzijom: kanonski primjer je vjerojatno faktorijel,

$$0! := 1, \qquad \texttt{f = 1} \tag{2.13}$$

$$(n+1)! := (n+1) \cdot n!, \qquad \texttt{for i in range(n): f = (i+1)*f} \tag{2.14}$$

samo zasad ne možemo reći da je funkcija $\mathsf{factorial}^1$ $(n \mapsto n!)$ dobivena primitivnom rekurzijom, jer ne postoji odgovarajuća funkcija G. Recimo, za $G := C_1^1$, i odgovarajuću funkciju H^3, definicija 2.11 bi nam dala $\mathsf{factorial}^2$, što nije funkcija koju tražimo. (Ali nije ni daleko od nje, kao što ćemo vidjeti kasnije.)

Definicija višemjesnih funkcija primitivnom rekurzijom sasvim lijepo funkcionira.

Primjer 2.13: $\mathsf{add}^2 = I_1^1 \mathbin{\text{ꝑ}} \mathsf{Sc} \circ I_3^3$. Doista, zbrajanje dva broja možemo prikazati kao

$$x + 0 = x \qquad \mathsf{add}(x, 0) = I_1^1(x), \tag{2.15}$$

$$x + (y+1) = (x+y) + 1 \qquad \mathsf{add}(x, y+1) = (\mathsf{Sc} \circ I_3^3)\big(x, y, \mathsf{add}(x, y)\big). \tag{2.16}$$

Slično, $\mathsf{mul}^2 = Z \mathbin{\text{ꝑ}} \mathsf{add}^2 \circ (I_1^3, I_3^3)$. (Sami napišite pow!) ◁

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 2.14: Skup $\mathcal{C}\mathrm{omp}$ je zatvoren na primitivnu rekurziju.

Dokaz. Neka je $k \in \mathbb{N}_+$ te neka su $G^k, H^{k+2} \in \mathcal{C}\mathrm{omp}$ 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 $F^{k+1} := G \,\mathrm{PR}\, H$. 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 $\mathcal{R}_{k+2}$ — ne možemo koristiti $\mathcal{R}_{k+1}$ jer ide u suprotnom smjeru: svakim izvršavanjem tijela petlje $\mathcal{R}_{k+1}$ se dekrementira, dok se kontrolna varijabla mora inkrementirati (brojeći prolaze kroz petlju).

Kontrolnu varijablu ne treba inicijalizirati: kako $\mathcal{R}_{k+2}$ nije ulazni registar, na početku izračunavanja njegov sadržaj već jest 0. Također, inkrementirati kontrolnu varijablu moramo *nakon* izvođenja tijela petlje (funkcijski makro s P_H) jer želimo brojiti samo „završene" prolaze.

Sve u svemu, tvrdimo da makro-program

$$Q_F := \left[\begin{array}{l} 0.\ P_G(\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_k) \to \mathcal{R}_0 \ \text{USING}\ \mathcal{R}_{k+3}.. \\ 1.\ \text{DEC}\ \mathcal{R}_{k+1}, 5 \\ 2.\ P_H(\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_k, \mathcal{R}_{k+2}, \mathcal{R}_0) \to \mathcal{R}_0 \ \text{USING}\ \mathcal{R}_{k+3}.. \\ 3.\ \text{INC}\ \mathcal{R}_{k+2} \\ 4.\ \text{GO TO}\ 1 \end{array}\right] \tag{2.17}$$

računa F. U tu svrhu, po definiciji 1.11, neka je $(\vec{x}, y) \in \mathbb{N}^{k+1}$ proizvoljan. Na početku Q_F-izračunavanja s $(\vec{x}, y)$ imamo makro-konfiguraciju $(0, \vec{x}, y, 0, 0, \ldots, 0, 0)$, koja izvršavanjem funkcijskog makroa rednog broja 0 prelazi u $\big(G(\vec{x}), \vec{x}, y, 0, 0, \ldots, 1, 0\big)$ (jer je G totalna). Broj u $\mathcal{R}_0$ je prema (2.11) jednak $F(\vec{x}, 0)$. Sada ako za sve $i \in [0 \cdot\cdot\, y]$ označimo $d_i := \big(F(\vec{x}, i), \vec{x}, y - i, i, 0, \ldots, 1, 0\big)$, upravo smo došli do konfiguracije d_0.

Tvrdimo da za svaki $i < y$, $d_i \rightsquigarrow^* d_{i+1}$. Doista, iz $i < y$ slijedi $y - i > 0$, pa imamo

$$\begin{aligned} d_i = \big(F(\vec{x}, i), \vec{x}, y - i, i, 0, \ldots, 1, 0\big) &\rightsquigarrow \big(F(\vec{x}, i), \vec{x}, y - i - 1, i, 0, \ldots, 2, 0\big) \rightsquigarrow^* \\ \rightsquigarrow^* \big(H\big(\vec{x}, i, F(\vec{x}, i)\big), \vec{x}, y - i - 1, i, 0, \ldots, 3, 0\big) &\rightsquigarrow \big(F(\vec{x}, i+1), \vec{x}, y - i - 1, i+1, 0, \ldots, 4, 0\big) \rightsquigarrow \\ &\rightsquigarrow \big(F(\vec{x}, i+1), \vec{x}, y - (i+1), i+1, 0, \ldots, 1, 0\big) = d_{i+1}. \end{aligned} \tag{2.18}$$

Ovdje je bilo važno da je i H totalna, pa računanje vrijednosti $H\big(\vec{x}, i, F(\vec{x}, i)\big)$ (prema (2.12) jednake $F(\vec{x}, i+1)$) doista stane nakon konačno mnogo koraka, predstavljenih strelicom $\rightsquigarrow^*$ u (2.18). Sada odmah indukcijom po i slijedi da za svaki $i \leq y$, Q_F-izračunavanje s $(\vec{x}, y)$ sadrži konfiguraciju d_i. Specijalno, u njemu postoji konfiguracija d_y, koja prelazi u završnu konfiguraciju

$$d_y = \big(F(\vec{x}, y), \vec{x}, y - y, y, 0, \ldots, 1, 0\big) \rightsquigarrow \big(F(\vec{x}, y), \vec{x}, 0, y, 0, \ldots, 5, 0\big) \circlearrowleft \tag{2.19}$$

oblika $\big(F(\vec{x}, y), \ldots\big)$, što smo i trebali. Sada prema teoremu 1.27, RAM-algoritam $(Q_F^\flat)^{k+1}$ također računa F, pa je $F \in \mathcal{C}\mathrm{omp}$. $\square$

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 2.13). Takav RAM-program je ogroman 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 2.15: Skup *primitivno rekurzivnih* funkcija je najmanji skup funkcija koji sadrži sve inicijalne funkcije te je zatvoren na kompoziciju i na primitivnu rekurziju. $\triangleleft$

Tehnički, 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 $\mathcal{F}unc := \bigcup_{k\in\mathbb{N}_+}\mathcal{F}unc_k$ svakako zadovoljava uvjete.

Definicija 2.15 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 2.16: Neka je F brojevna funkcija. Tada je F primitivno rekurzivna ako i samo ako se F 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, tada ona svakako pripada skupu $\mathcal{S}$ jer ju je moguće dobiti iz inicijalnih pomoću 0 primjena kompozicije i primitivne rekurzije.

Ako su $k, l \in \mathbb{N}_+$ te $G_1^k, \ldots, G_l^k, H^l \in \mathcal{S}$, 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 $H \circ (G_1, \ldots, G_l)$ 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 — što je konačno mnogo, pa je $H \circ (G_1, \ldots, G_l) \in \mathcal{S}$.

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, uz r_G odnosno r_H primjena primitivne rekurzije. Tada se $G \mathbin{\mathrm{PR}} H$ može dobiti iz inicijalnih funkcija pomoću najviše $c_G + c_H$ primjena kompozicije, i najviše $r_G + r_H + 1$ primjena primitivne rekurzije — što je konačno mnogo, pa je $G \mathbin{\mathrm{PR}} H \in \mathcal{S}$.

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 $c = r = 0$, odnosno F mora biti inicijalna. Skup primitivno rekurzivnih funkcija sadrži sve inicijalne funkcije, pa tako i F.

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 $F = H \circ (G_1, \ldots, G_l)$ za neke $G_1, \ldots, G_l, H \in \mathcal{S}$, ili pak $F = G \mathbin{\text{PR}} H$, 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 $F = G \mathbin{\text{PR}} H$, tada je $r > 0$ te su G i H dobivene 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 $F = G \mathbin{\text{PR}} H$ primitivno rekurzivna. Analogno bi se dokazalo da je F oblika $H \circ (G_1, \ldots, G_l)$ primitivno rekurzivna. □

Napomena 2.17: 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 2.9 navedena je simbolička definicija od C_2^3, a u primjeru 2.13, simboličke definicije od add^2 i mul^2. ◁

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 (2.15)–(2.16) sadrži točkovnu definiciju zbrajanja dvaju brojeva, iz koje se vidi da je ono primitivno rekurzivno. U desnom stupcu je također točkovna definicija, ali manje čitljiva jer je namještena na oblik (2.11)–(2.12), iz kojeg se odmah može pročitati simbolička definicija. Programski, imali bismo:

```
z = x                          # G(x)=x
for i in range(y): z = z + 1   # H(x,i,z)=Sc(z)
```
(2.20)

Točkovne definicije su katkad 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 — evo primjera.

Primjer 2.18: Točkovna definicija

$$f(x, y, z, 0) := g\big(x, h(x, y, z)\big) \tag{2.21}$$

$$f(x, y, z, t + 1) := h\big(z, f(x, y, z, t), g(t, z)\big) \tag{2.22}$$

ekvivalentna je simboličkoj definiciji

$$f^4 := g^2 \circ (I_1^3, h^3) \mathbin{\text{PR}} h^3 \circ \big(I_3^5, I_5^5, g^2 \circ (I_4^5, I_3^5)\big). \tag{2.23}$$

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

Stroga formalizacija točkovnih definicija, i njihova pretvaranja u simboličke, zahtijevala bi alate 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 1.27 — jer nam sveukupno do univerzalnosti treba samo konačno mnogo oblika takvih definicija, pa možemo svaki od njih zasebno pretvoriti u simbolički oblik ako treba.

Definiciju 2.15 možemo iskoristiti za dokazivanje da sve primitivno rekurzivne funkcije imaju neko svojstvo $\wp$, baš kao što smo to učinili u dokazu propozicije 2.16. 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. Skup primitivno rekurzivnih funkcija je podskup bilo kojeg skupa koji ima ta svojstva, pa tako i od $\mathcal{S}$. (Zaključivanje matematičkom indukcijom također je takvog oblika: $\mathbb{N}$ je najmanji skup koji sadrži 0 i zatvoren je na Sc.) Evo dva primjera.

Propozicija 2.19: 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 2.8 slijedi da je taj skup zatvoren na kompoziciju, a iz napomene 2.12 da je zatvoren i na primitivnu rekurziju. Dakle tvrdnja slijedi po definiciji 2.15. □

Propozicija 2.20: Sve primitivno rekurzivne funkcije su RAM-izračunljive.

Dokaz. Po definiciji 2.15, koristeći korolar 2.4 te leme 2.10 i 2.14. □

2.2.2. Primjeri primitivno rekurzivnih funkcija i relacija

Već smo vidjeli simboličke definicije od add^2 i mul^2, što prema propoziciji 2.16 znači da su one primitivno rekurzivne. Sada ćemo vidjeti još brojne druge primjere. Prvo poopćimo primjer 2.9.

Propozicija 2.21: Za sve $n \in \mathbb{N}$ i za sve $k \in \mathbb{N}_+$, konstantna funkcija C_n^k, zadana s $\mathsf{C}_n^k(\vec{x}) := n$, primitivno je rekurzivna.

Dokaz. Fiksirajmo $k \in \mathbb{N}_+$, i dokažimo tvrdnju „sve C_n^k su primitivno rekurzivne", indukcijom po n. Baza: $\mathsf{C}_0^k = \mathsf{Z} \circ \mathsf{I}_1^k$ je simbolička definicija od C_0^k. Doista,

$$(\mathsf{Z} \circ \mathsf{I}_1^k)(\vec{x}) = \mathsf{Z}\big(\mathsf{I}_1^k(\vec{x})\big) = \mathsf{Z}(x_1) = 0 = \mathsf{C}_0^k(\vec{x}). \tag{2.24}$$

To znači da je C_0^k dobivena iz dvije inicijalne funkcije kompozicijom, pa je primitivno rekurzivna (po propoziciji 2.16, 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 $\mathsf{C}_{m+1}^k = \mathsf{Sc} \circ \mathsf{C}_m^k$ simbolička definicija od C_{m+1}^k. Doista,

$$(\mathsf{Sc} \circ \mathsf{C}_m^k)(\vec{x}) = \mathsf{Sc}\big(\mathsf{C}_m^k(\vec{x})\big) = \mathsf{Sc}(m) = m + 1 = \mathsf{C}_{m+1}^k(\vec{x}). \tag{2.25}$$

To znači da je C^k_{m+1} 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 (2.13)–(2.14). Takve funkcije neće biti „u korijenu" dobivene primitivnom rekurzijom (već kompozicijom), ali će biti primitivno rekurzivne.

Propozicija 2.22: Neka je $a \in \mathbb{N}$ i H^2 primitivno rekurzivna funkcija. Tada je primitivno rekurzivna i funkcija F^1, zadana s

$$F(0) := a, \tag{2.26}$$

$$F(x+1) := H\big(x, F(x)\big). \tag{2.27}$$

Dokaz. Dodat ćemo još jedan argument funkciji F, koji neće raditi ništa osim što će povećavati sve mjesnosti za 1. Precizno, definiramo funkciju F^2 (različitu od F^1, jer je mjesnost dio identiteta funkcije) s $F(x,y) := F(y)$ (za sve x i y). Tada jednakosti

$$F(x,0) = F(0) = a = C^1_a(x), \tag{2.28}$$

$$F(x,y+1) = F(y+1) = H\big(y, F(y)\big) = H\big(y, F(x,y)\big) =: H\big(x, y, F(x,y)\big), \tag{2.29}$$

kažu da je F^2 dobivena (pravom) primitivnom rekurzijom iz primitivno rekurzivnih funkcija $G^1 := C^1_a$ i $H^3 := H^2 \circ (I^3_2, I^3_3)$, pa je primitivno rekurzivna. Sada je (primjerice) $F(x) = F(0,x)$, odnosno $F^1 = F^2 \circ (Z, I^1_1)$, iz čega slijedi da je i F^1 primitivno rekurzivna. $\square$

Definicija 2.23: Za takve funkcije ubuduće pišemo „simboličku definiciju" $F^1 := a \,\mathrm{PR}\, H^2 := C^0_a \,\mathrm{PR}\, H^2$, imajući na umu da je to samo pokrata za $F^1 := \big(C^1_a \,\mathrm{PR}\, H^2 \circ (I^3_2, I^3_3)\big) \circ (Z, I^1_1)$.

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

Pomoću propozicije 2.22 možemo dokazati primitivnu rekurzivnost raznih funkcija.

Primjer 2.24: Jednadžbe (2.13) i (2.14) kažu da je $\mathrm{factorial}^1 = 1 \,\mathrm{PR}\, \mathrm{mul}^2 \circ (\mathrm{Sc} \circ I^2_1, I^2_2)$ simbolička definicija funkcije faktorijel, pa je ona primitivno rekurzivna. $\triangleleft$

Primjer 2.25: Funkcija *prethodnik* zadana je s $\mathrm{pd}(x) := \max\{x-1, 0\}$. Iz toga slijedi $\mathrm{pd}\big(\mathrm{Sc}(x)\big) = x$ 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 $\mathrm{pd}(0) = 0$ moramo posebno definirati. Te dvije jednakosti (napisane desno u obliku koda u Pythonu):

$$\mathrm{pd}(0) = 0, \qquad \texttt{p = 0} \tag{2.30}$$

$$\mathrm{pd}(y+1) = y, \qquad \texttt{for i in range(y): p = i} \tag{2.31}$$

kažu da je pd dobivena degeneriranom primitivnom rekurzijom $\mathrm{pd} = 0 \,\mathrm{PR}\, I^2_1$, pa je primitivno rekurzivna po propoziciji 2.22. $\triangleleft$

Napomena 2.26: Još jedan način definiranja funkcije pd, koji ćemo često koristiti kasnije, je sljedeći: htjeli bismo definirati $\mathrm{pd}(x)$ 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 točkicom $\cdot$ označimo tu transformaciju (formalno, $n^\cdot$ je n ako je pozitivan, a 1 ako je $n = 0$), tada možemo precizno definirati $\mathrm{pd}(x) := x^\cdot - 1 (= x \dot{-} 1$, pogledajte primjer 2.27). $\triangleleft$

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

Primjer 2.27: Za $x, y \in \mathbb{N}$, označimo $x \dot{-} y := \max\{x - y, 0\}$. Iz jednakosti/koda

$$x \dot{-} 0 = x, \qquad \texttt{s = x} \tag{2.32}$$

$$x \dot{-} (y + 1) = \mathsf{pd}(x \dot{-} y), \qquad \texttt{for i in range(y): s = pd(s)} \tag{2.33}$$

vidimo da je *ograničeno oduzimanje* dobiveno primitivnom rekurzijom iz funkcija I_1^1 i $\mathsf{pd} \circ \mathsf{I}_3^3$, pa je ono primitivno rekurzivna operacija. „Operacija" nam znači dvomjesnu totalnu brojevnu funkciju koja se piše infiksno između operanada. Kao funkciju, ograničeno oduzimanje zovemo sub. Dakle, $\mathsf{sub}^2 = \mathsf{I}_1^1 \mathbin{\text{PR}} \mathsf{pd} \circ \mathsf{I}_3^3$. Još jedan način dolaska do te simboličke definicije je da uzmemo simboličku definiciju za add^2 (primjer 2.13) i zamijenimo u njoj Sc s pd. ◁

Osim računskih operacija (dijeljenje s ostatkom definirat ćemo kasnije, kad uvedemo još neke tehnike), brojeve možemo i *uspoređivati*, raznim dvomjesnim relacijama poput $<$, $\geq$ ili $=$. Rekosmo, izračunljivost relacija zapravo je izračunljivost njihovih karakterističnih funkcija.

Primjer 2.28: Za početak pogledajmo pozitivnost — formalno, jednomjesnu relaciju $\mathbb{N}_+$. Njena karakteristična funkcija je 0 u nuli, a 1 u svim ostalim prirodnim brojevima, pa se u literaturi još zove funkcija predznaka ili *signum*. Definicija te funkcije vodi na degeneriranu primitivnu rekurziju $\chi_{\mathbb{N}_+} = 0 \mathbin{\text{PR}} \mathsf{C}_1^2$, iz čega zaključujemo da je $\mathbb{N}_+$ primitivno rekurzivan. ◁

Kada imamo pozitivnost i ograničeno oduzimanje, možemo odmah vidjeti primitivnu rekurzivnost strogog uređaja.

Primjer 2.29: Rastavom na slučajeve vidimo da je $x > y$ ako i samo ako je $x \dot{-} y$ pozitivan, iz čega je $\chi_> = \chi_{\mathbb{N}_+} \circ \mathsf{sub}$, primitivno rekurzivna funkcija. Očito je $x < y$ ako i samo ako je $y > x$, dakle $\chi_< = \chi_> \circ (\mathsf{I}_2^2, \mathsf{I}_1^2)$ je također primitivno rekurzivna. ◁

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 $\otimes^1$, što je bilo gotovo trivijalno u RAM-paradigmi. Da bismo dobili i parcijalne (netotalne) funkcije, moramo uvesti novi operator, pored $\circ$ i $\mathbin{\text{PR}}$. Taj operator — *minimizacija* — djelovat će na *relacijama* (zadanim karakterističnim funkcijama) i tražit će najmanji prirodni broj koji ima neko svojstvo. Intuitivno, odgovarat će *neograničenim* petljama (poput petlje `while` u Pythonu, samo s negiranim uvjetom) koje se ne izvršavaju određeni broj puta, nego dok se ne ispuni neki uvjet.

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

Definicija 2.30: Neka je $k \in \mathbb{N}_+$ i R^{k+1} relacija. Za funkciju F^k definiranu s

$$\mathcal{D}_F := \{\vec{x} \in \mathbb{N}^k \mid \exists y R(\vec{x}, y)\} =: \exists_* R, \tag{2.34}$$

$$F(\vec{x}) := \min\{y \in \mathbb{N} \mid R(\vec{x}, y)\}, \text{ za sve } \vec{x} \in \exists_* R, \tag{2.35}$$

kažemo da je dobivena *minimizacijom* relacije R. Pišemo $F^k := \mu R^{k+1}$, ili točkovno $F(\vec{x}) :\simeq \mu y R(\vec{x}, y)$. Za skup funkcija $\mathcal{F}$ kažemo da je *zatvoren na minimizaciju* ako za svaki $k \in \mathbb{N}_+$, za svaku $(k+1)$-mjesnu relaciju R, $\chi_R \in \mathcal{F}$ povlači $\mu R \in \mathcal{F}$. ◁

Domena funkcije μR^{k+1} je k-mjesna relacija koju označavamo s $\exists_* R$ i zovemo je *projekcija* relacije R. Motivaciju za taj naziv možemo vidjeti ako zamislimo tromjesnu relaciju R^3 grafički prikazanu kao skup točaka u 3D-prostoru. Tada je grafički prikaz od $\exists_* R^3$ upravo 2D-projekcija (niz os z) toga skupa na ravninu x-y. Točka (x, y) će biti u $\exists_* R$ ako i samo ako iznad nje postoji neka točka $(x, y, z) \in R$ (može ih biti i više).

U tom prikazu 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 R iznad (x, y).

Da bi izraz $\mu y R(\vec{x}, y)$ imao vrijednost, očito je nužno da bude $\vec{x} \in \exists_* R$. Za obrat — da za *svaki* $\vec{x} \in \exists_* R$, izraz $\mu y R(\vec{x}, y)$ ima vrijednost — zaslužna je dobra uređenost od $\mathbb{N}$: ako postoji neki $y \in \mathbb{N}$ za koji vrijedi $R(\vec{x}, y)$, tada postoji i najmanji takav.

Primjer 2.31: Praznu funkciju možemo dobiti minimizacijom prazne relacije: konkretno, za svaki $k \in \mathbb{N}_+$, $\otimes^k = \mu \varnothing^{k+1}$. Naime, projekcija prazne relacije je opet prazna, a jedina funkcija s praznom domenom je prazna funkcija. ◁

Dodavanjem minimizacije proširili smo skup izračunljivih funkcija u ovom modelu.

Definicija 2.32: 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. ◁

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

Lema 2.33: Skup rekurzivnih funkcija zatvoren je na kompoziciju.

Dokaz. Neka su $k, l \in \mathbb{N}_+$ te $G_1^k, \ldots, G_l^k$ i H^l rekurzivne funkcije. Tada su one po definiciji totalne i parcijalno rekurzivne. Skup parcijalno rekurzivnih funkcija je zatvoren na kompoziciju, pa je $H \circ (G_1, \ldots, G_l)$ parcijalno rekurzivna, no prema propoziciji 2.8, ona je i totalna — dakle rekurzivna funkcija. □

Lema 2.34: 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 $F^{k+1} := G \mathbin{\text{PR}} H$, i ona je totalna po napomeni 2.12. Također, kako su G i H parcijalno rekurzivne, a skup parcijalno rekurzivnih funkcija je po definiciji zatvoren na primitivnu rekurziju, F je parcijalno rekurzivna. Iz toga dvojeg slijedi: F je rekurzivna funkcija. □

Korolar 2.35: Svaka primitivno rekurzivna funkcija je rekurzivna.

Dokaz. Po definiciji 2.15: sve inicijalne funkcije su totalne po napomeni 2.2, a parcijalno su rekurzivne po definiciji, dakle rekurzivne su. Sada samo treba primijeniti leme 2.33 i 2.34. □

Korolar 2.36: Neka je $a \in \mathbb{N}$ i H^2 rekurzivna funkcija. Tada je i funkcija $a \mathbin{\text{PR}} H$ rekurzivna.

Dokaz. Po definiciji 2.23 je $a \mathbin{\text{PR}} H^2 = \left(C_a^1 \mathbin{\text{PR}} H^2 \circ (I_2^3, I_3^3)\right) \circ (Z, I_1^1)$ pa tvrdnja slijedi iz propozicije 2.21, korolara 2.35 te lema 2.33 i 2.34. □

Slično kao propoziciju 2.16, 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 bilo poopćenje simboličke definicije za parcijalno rekurzivne funkcije. No takvim funkcijama najčešće ćemo pisati točkovne definicije — a i Kleenejev teorem o normalnoj formi, koji ćemo dokazati kasnije, reći će da tolika općenitost nije potrebna: dovoljno je promatrati funkcije oblika $U \circ \mu T$, za primitivno rekurzivne U i T.

Pogledajmo sada kako možemo proširiti rezultat iz propozicije 2.20 na parcijalno rekurzivne funkcije. Nedostaje nam jedino opis izvršavanja neograničenih petlji na RAM-stroju.

Ideja nije ništa revolucionarno: kao „kontrolnu varijablu" koristimo upravo $\mathcal{R}_0$, iz istog razloga kao u dokazu leme 2.14 — da eventualnim završetkom petlje bude već spremna kao izlazni podatak. U tom smislu, već je inicijalizirana na nulu 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`, ne `while`. Iz sličnog razloga kao naši strojevi, Pascal `while`-uvjet provjerava na vrhu, a `until`-uvjet na dnu petlje. Semantika skoka je ista: ako je uvjet istinit nastavljamo (ulazimo u `while`-petlju, ili izlazimo iz `until`-petlje), a ako je lažan skačemo na drugi kraj (izlazimo iz `while`-petlje, ili ponovo izvršavamo `until`-petlju) — što je baš semantika instrukcije tipa DEC.

Ali `until`-petlje imaju jedan nedostatak: njihovo tijelo uvijek se izvrši barem jednom (kao kod petlje `do...while` u jeziku C). Kako se u petlji mora nalaziti instrukcija INC $\mathcal{R}_0$, čini se da teško možemo postići da $\mathcal{R}_0$ na kraju bude 0, što može biti 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! Na početak programa možemo staviti instrukciju tipa GO TO, kako bismo preskočili instrukciju INC $\mathcal{R}_0$ kod prvog prolaza.

Lema 2.37: Skup $\mathcal{C}omp$ je zatvoren na minimizaciju.

Dokaz. Neka je $k \in \mathbb{N}_+$ te 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

$$Q_F := \left[\begin{array}{l} 0.\ \text{GO TO } 2 \\ 1.\ \text{INC } \mathcal{R}_0 \\ 2.\ P_R(\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_k, \mathcal{R}_0) \to \mathcal{R}_{k+1} \text{ USING } \mathcal{R}_{k+2}.. \\ 3.\ \text{DEC } \mathcal{R}_{k+1}, 1 \end{array}\right] \tag{2.36}$$

računa funkciju $F^k := \mu R$.
U tu svrhu, po definiciji 1.11, neka je $\vec{x} \in \mathbb{N}^k$ proizvoljan. Moramo dokazati dvije tvrdnje:

1. Ako Q_F-izračunavanje s $\vec{x}$ stane, tada $\vec{x} \in \exists_* R = \mathcal{D}_{\mu R}$.

2. Ako je $\mu y R(\vec{x}, y) = n \in \mathbb{N}$, tada Q_F-izračunavanje s $\vec{x}$ stane s rezultatom n.

Primijetimo da se za vrijeme Q_F-izračunavanja s $\vec{x}$ sadržaj registara $\mathcal{R}_1, \ldots, \mathcal{R}_k$ ne mijenja (RAM-instrukcije u Q_F nemaju odredišta između 1 i k, a funkcijski makro čuva prvih $k+1$ registara), pa je čitavo vrijeme njihov sadržaj jednak $\vec{x}$.

Za prvu tvrdnju, jedini način da promatrano izračunavanje stane je da makro-stroj dođe u završnu konfiguraciju oblika $(y, \vec{x}, t, \ldots, 4, 0)$, a jedini pak način da se to dogodi je dolazak iz konfiguracije $(y, \vec{x}, t+1, \ldots, 3, 0)$, jer nema odredišta 4. Makro-instrukcija izvršena prije toga morala je biti funkcijski makro, što zbog činjenice da P_R računa χ_R znači da je $\chi_R(\vec{x}, y) = t+1 > 0$, odnosno vrijedi $R(\vec{x}, y)$, pa je $\vec{x} \in \exists_* R$.

Za drugu tvrdnju, dokažimo prvo (indukcijom po n) pomoćnu tvrdnju: ako ni za koji $y < n$ ne vrijedi $R(\vec{x}, y)$, tada makro-stroj u nekom trenutku dođe u konfiguraciju $(n, \vec{x}, 0, \ldots, 2, 0) =: q_n$. Za $n = 0$ antecedens trivijalno vrijedi, ali vrijedi i konzekvens: početak izračunavanja je

$$(0, \vec{x}, 0, \ldots, 0, 0) \rightsquigarrow (0, \vec{x}, 0, \ldots, 2, 0) = q_0. \tag{2.37}$$

Pretpostavimo sada da za sve $y < n+1$ vrijedi $\neg R(\vec{x}, y)$. Tada posebno to vrijedi i za sve $y < n$, pa po pretpostavci indukcije makro-stroj nekada dođe u konfiguraciju q_n. Dalje se izračunavanje odvija ovako (zbog $n < n+1$ je $\neg R(\vec{x}, n)$, što znači da funkcijski makro stavi $\chi_R(\vec{x}, n) = 0$ u $\mathcal{R}_{k+1}$):

$$q_n = (n, \vec{x}, 0, \ldots, 2, 0) \rightsquigarrow^* (n, \vec{x}, 0, \ldots, 3, 0) \rightsquigarrow (n, \vec{x}, 0, \ldots, 1, 0) \rightsquigarrow (n+1, \vec{x}, 0, \ldots, 2, 0) = q_{n+1}.$$

Sada, ako je $n = \mu y R(\vec{x}, y)$, tada očito ni za koji $y < n$ ne vrijedi $R(\vec{x}, y)$, pa prema pomoćnoj tvrdnji makro-stroj dođe u konfiguraciju q_n. Dalje se izračunavanje odvija ovako (zbog $R(\vec{x}, n)$ funkcijski makro stavi $\chi_R(\vec{x}, n) = 1$ u $\mathcal{R}_{k+1}$):

$$q_n = (n, \vec{x}, 0, \ldots, 2, 0) \rightsquigarrow^* (n, \vec{x}, 1, \ldots, 3, 0) \rightsquigarrow (n, \vec{x}, 0, \ldots, 4, 0) \circlearrowleft .$$

Sada prema teoremu 1.27 RAM-algoritam $Q_F^{\flat k}$ računa F, pa je $F \in \mathcal{C}omp$. □

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

Teorem 2.38: Svaka parcijalno rekurzivna funkcija je RAM-izračunljiva.

Dokaz. Skup $\mathcal{C}omp$ sadrži sve inicijalne funkcije prema korolaru 2.4, a zatvoren je na kompoziciju, primitivnu rekurziju i minimizaciju prema lemama 2.10, 2.14 i 2.37 — dakle nadskup je najmanjeg takvog skupa, skupa svih parcijalno rekurzivnih funkcija. □

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, ili pak 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`) $\mathtt{compile}$ 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 $\mathtt{compile}[L] := Q_L^\flat$. Ako se nalazimo na unutarnjem čvoru N, ovisno o tome kako je dobiven iz svoje djece koristimo dokaz leme 2.10, 2.14 ili 2.37 da utvrdimo koji oblik makro-programa — (2.8), (2.17) ili (2.36) — računa N, u taj predložak (*template*) uvrštavajući funkcijske makroe koji na odgovarajućem mjestu pozivaju $\mathtt{compile}[D_i]$, gdje su D_i djeca čvora N. (Zbog *postorder*-obilaska, D_i su već kompilirani.) Dobivši tako makro-program Q_N, definiramo $\mathtt{compile}[N] := Q_N^\flat$. Na kraju *postorder*-obilaska nalazi se korijen K, i $\mathtt{compile}[K]$ će biti RAM-program koji računa parcijalno rekurzivnu funkciju (ili relaciju) zadanu simboličkom definicijom.

Za obrat teorema 2.38 morat ćemo se više potruditi. To je tema poglavlja 3.

2.4. Tehnike za rad s (primitivno) rekurzivnim funkcijama

Korištenje marljive evaluacije olakšalo nam je dokaz RAM-izračunljivosti simbolički definiranih funkcija — jer marljiva kompozicija upravo odgovara slijednom izvršavanju instrukcija u programu — ali će otežati dokaz suprotnog smjera, jer neke, u RAM-modelu jednostavne, programske tehnike zasad ne znamo ostvariti u funkcijskom modelu s marljivom evaluacijom.

Jedan primjer 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. Možemo napisati makro

$$(\textsc{if}\ \mathcal{R}_j\ \textsc{then}\ P^*_{true}\ \textsc{else}\ P^*_{false}) := \begin{bmatrix} 0.\ \textsc{dec}\ \mathcal{R}_j, 4 \\ 1.\ \textsc{inc}\ \mathcal{R}_j \\ 2.\ P^*_{true} \\ 3.\ \textsc{go to}\ 5 \\ 4.\ P^*_{false} \end{bmatrix}^{\flat *}, \tag{2.38}$$

ali ne možemo napisati odgovarajuću funkciju If^3 takvu da $f(\vec{x}, y) \simeq \mathrm{If}\big(y, g(\vec{x}), h(\vec{x})\big)$ 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 je nezaustavljanje *jedini* problem — jer nas zanima samo postojanje algoritma a ne i performanse, s totalnim izračunljivim funkcijama nešto poput If možemo napraviti relativno lako. Za početak, za implementaciju $\textsc{else}$ trebamo dokazati da negiranje uvjeta ne kvari izračunljivost.

Propozicija 2.39: Neka je $k \in \mathbb{N}_+$ te R^k (primitivno) rekurzivna relacija.
Tada je i relacija $(R^c)^k$, zadana s $R^c(\vec{x}) :\Longleftrightarrow \neg R(\vec{x})$, također (primitivno) rekurzivna.

Uočimo 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 kao na skupove k-torki. Zato koristimo oznaku za skupovni komplement, dok u definiciji

formulom koristimo logičku negaciju. U sljedećoj propoziciji to ćemo koristiti za dvomjesne logičke veznike odnosno skupovne operacije.

Drugo, mjesnost R^c smatramo istom kao mjesnost od R. Čak i u slučaju prazne relacije, $(\varnothing^k)^c = \mathbb{N}^k$ („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 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. Želimo prikazati karakterističnu funkciju χ_{R^c} pomoću χ_R. Dakle, potrebna nam je funkcija koja 0 preslikava u 1, a 1 u 0. Jedna takva je $x \mapsto 1 - x$, ali nama treba *brojevna* funkcija s $\mathbb{N}$ u $\mathbb{N}$. Zato ćemo uzeti $f_0(x) := 1 \dot{-} x$, koja jednako djeluje na skupu $\{0, 1\}$, a sve vrijednosti su joj prirodni brojevi. Ta funkcija je primitivno rekurzivna po propoziciji 2.16: $f_0 := \mathrm{sub} \circ (C_1^1, I_1^1)$ njena je simbolička definicija, sub je primitivno rekurzivna po primjeru 2.27, C_1^1 po propoziciji 2.21, a I_1^1 je inicijalna.

Sada točkovna jednakost $\chi_{R^c}(\vec{x}) = 1 \dot{-} \chi_R(\vec{x})$ 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 2.35. Prema lemi 2.33, tada je i χ_{R^c} rekurzivna kao kompozicija dvije rekurzivne funkcije. □

Ubuduće ćemo samo napisati točkovnu 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, a ako su zadane funkcije rekurzivne, tada je i tražena funkcija rekurzivna. Efektivno, imamo poopćenje napomene 2.17, gdje polazimo od rekurzivnih funkcija umjesto od primitivno rekurzivnih.

Korolar 2.40: Brojevne relacije nestrogog uređaja $\leq$ i $\geq$ su primitivno rekurzivne.

Dokaz. Tvrdnje slijede iz primjera 2.29, propozicije 2.39 te očitih ekvivalencija

$$x \leq y \iff \neg(x > y) \qquad (\leq) = (>)^c, \tag{2.39}$$

$$x \geq y \iff \neg(x < y) \qquad (\geq) = (<)^c \tag{2.40}$$

(točkovno u lijevom stupcu, simbolički u desnom). □

Propozicija 2.41: 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

$$Q_1(\vec{x}) :\Longleftrightarrow R(\vec{x}) \wedge P(\vec{x}) \qquad Q_1 := R \cap P, \tag{2.41}$$
$$Q_2(\vec{x}) :\Longleftrightarrow R(\vec{x}) \vee P(\vec{x}) \qquad Q_2 := R \cup P, \tag{2.42}$$
$$Q_3(\vec{x}) :\Longleftrightarrow R(\vec{x}) \to P(\vec{x}) \qquad Q_3 := (R \setminus P)^c, \tag{2.43}$$
$$Q_4(\vec{x}) :\Longleftrightarrow R(\vec{x}) \leftrightarrow P(\vec{x}) \qquad Q_4 := (R \triangle P)^c. \tag{2.44}$$

Dokaz. Prvo pokažimo da su skupovne i logičke definicije ekvivalentne za upravo definirane relacije. Za Q_1 i Q_2 to je upravo definicija presjeka i unije. Za $Q_3(\vec{x})$ imamo

$$\vec{x} \in (R \setminus P)^c \Leftrightarrow \neg(\vec{x} \in R \wedge \vec{x} \notin P) \Leftrightarrow \vec{x} \notin R \vee \vec{x} \in P \Leftrightarrow \vec{x} \in R \to \vec{x} \in P. \tag{2.45}$$

Za $Q_4(\vec{x})$, 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 2.39, trebamo naći dvomjesne primitivno rekurzivne funkcije f_i koje će preslikavati $\chi_R(\vec{x})$ i $\chi_P(\vec{x})$ u $\chi_{Q_i}(\vec{x})$, odnosno one koje će na skupu $\{0, 1\}$ djelovati onako kako propisuju tablice istinitosti za pojedine logičke veznike.

Za konjunkciju odnosno presjek, to je upravo $f_1 = \mathrm{mul}^2$ — u starijoj literaturi za konjunkciju se još rabi izraz „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:

$$\varphi \vee \psi \Longleftrightarrow \neg(\neg\varphi \wedge \neg\psi) \qquad f_2(x, y) := f_0\big(f_0(x) \cdot f_0(y)\big), \tag{2.46}$$
$$\varphi \to \psi \Longleftrightarrow \neg\varphi \vee \psi \qquad f_3(x, y) := f_2\big(f_0(x), y\big), \tag{2.47}$$
$$\varphi \leftrightarrow \psi \Longleftrightarrow (\varphi \to \psi) \wedge (\psi \to \varphi) \qquad f_4(x, y) := f_3(x, y) \cdot f_3(y, x), \tag{2.48}$$

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

Sada primjenom propozicije 2.39 na Q_3 i Q_4 dobijemo da je skupovna razlika, kao i simetrična skupovna razlika, (primitivno) rekurzivnih relacija iste mjesnosti ponovo (primitivno) rekurzivna.

Korolar 2.42: Jednakost (dvomjesna brojevna relacija) je primitivno rekurzivna.

Dokaz. To slijedi iz korolara 2.40, propozicije 2.41 i očite ekvivalencije („Cantor–Bernstein za konačne skupove”) $x = y \Longleftrightarrow x \le y \wedge x \ge y$, simbolički $(=) = (\le) \cap (\ge)$. □

2.4.1. Teorem o grananju za totalne funkcije

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

Lema 2.43: Za svaki $k \in \mathbb{N}_+$, funkcije add^k i mul^k, zadane s

$$\mathsf{add}(x_1, x_2, \ldots, x_k) := x_1 + x_2 + \cdots + x_k, \tag{2.49}$$
$$\mathsf{mul}(x_1, x_2, \ldots, x_k) := x_1 \cdot x_2 \cdots x_k, \tag{2.50}$$

primitivno su rekurzivne.

Dokaz. Matematičkom indukcijom po k. Za $k = 1$, vidimo da je $\mathsf{add}^1 = \mathsf{mul}^1 = I_1^1$, inicijalna funkcija. Za $k = 2$, tvrdnja slijedi iz primjera 2.13. Pretpostavimo sad da su za neki $l \in \mathbb{N} \setminus \{0, 1\}$, funkcije add^l i mul^l primitivno rekurzivne. Tada definiciju

$$x_1 + x_2 + \cdots + x_l + x_{l+1} := (x_1 + x_2 + \cdots + x_l) + x_{l+1} \tag{2.51}$$

možemo zapisati kao

$$\mathsf{add}(x_1, x_2, \ldots, x_l, x_{l+1}) = \mathsf{add}\big(\mathsf{add}(x_1, x_2, \ldots, x_l), x_{l+1}\big) \tag{2.52}$$

ili simbolički

$$\mathsf{add}^{l+1} = \mathsf{add}^2 \circ \big(\mathsf{add}^l \circ (I_1^{l+1}, I_2^{l+1}, \ldots, I_l^{l+1}), I_{l+1}^{l+1}\big), \tag{2.53}$$

pa tvrdnja slijedi iz pretpostavke indukcije i primjera 2.13. Potpuno analogno, zamjenom add s mul, slijedi i druga tvrdnja za $l + 1$, odnosno po principu matematičke indukcije za svaki $k \in \mathbb{N}_+$. □

Propozicija 2.44: Neka su $k, l \in \mathbb{N}_+$ te R_1^k, R_2^k, …, 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 2.41 za Q_1, samo umjesto funkcije $f_1 = \mathsf{mul}^2$ koristimo mul^l. Konkretno, tvrdimo da je

$$\chi_{\bigcap_{i=1}^l R_i} = \chi_{R_1} \cdot \chi_{R_2} \cdots \chi_{R_l} = \mathsf{mul}^l \circ (\chi_{R_1}, \chi_{R_2}, \ldots, \chi_{R_l}) \tag{2.54}$$

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

Za uniju, možemo opet iskoristiti De Morganovo pravilo i propoziciju 2.39, ali možemo i upotrijebiti drugačiju tehniku, koja će nam biti korisna kasnije. U našem skupu nema negativnih brojeva, pa je zbroj 0 jedino ako su svi pribrojnici 0 — odnosno, zbroj je pozitivan ako i samo ako je neki pribrojnik pozitivan. To znači da l-struka unija odgovara funkciji $\chi_{\mathbb{N}_+} \circ \mathsf{add}^l$:

$$\chi_{\bigcup_{i=1}^l R_i} = \chi_{\mathbb{N}_+} \circ \mathsf{add}^l \circ (\chi_{R_1}, \chi_{R_2}, \ldots, \chi_{R_l}), \tag{2.55}$$

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

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

Definicija 2.45: Neka su $k, l \in \mathbb{N}_+$, neka su G_0^k, G_1^k, ..., G_l^k funkcije, a R_1^k, R_2^k, ..., R_l^k u parovima disjunktne relacije iste mjesnosti. Za funkciju F^k definiranu s

$$\mathcal{D}_F := \bigcup_{i=0}^{l} (\mathcal{D}_{G_i} \cap R_i), \quad \text{gdje je } R_0 := \left(\bigcup_{i=1}^{l} R_i\right)^c, \tag{2.56}$$

$$F(\vec{x}) := \begin{cases} G_1(\vec{x}), & R_1(\vec{x}) \\ G_2(\vec{x}), & R_2(\vec{x}) \\ & \vdots \\ G_l(\vec{x}), & R_l(\vec{x}) \\ G_0(\vec{x}), & \text{inače} \end{cases} \quad \text{za sve } \vec{x} \in \mathcal{D}_F, \tag{2.57}$$

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

Zahtjev da su svi uvjeti u parovima disjunktni znači da je za svaki $\vec{x} \in \mathbb{N}^k$ najviše jedan od njih ispunjen. Tako ne moramo brinuti o redoslijedu provjeravanja uvjeta — no ako već imamo fiksiran redoslijed ne nužno disjunktnih uvjeta R_1, R_2, ..., R_l, uvijek možemo napraviti nove disjunktne uvjete s istom unijom:

$$P_1 := R_1, \tag{2.58}$$

$$P_i := R_i \setminus \bigcup_{j=1}^{i-1} R_j, \text{ za sve } i \in [2 \cdot\cdot\, l], \tag{2.59}$$

koji će biti (primitivno) rekurzivni ako su R_i takvi, po propozicijama 2.41 i 2.44. 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, ali ako jesu, to možemo dokazati već sada.

Teorem 2.46 (Teorem o grananju, (primitivno) rekurzivna verzija): Neka su $k, l \in \mathbb{N}_+$, neka su G_0^k, G_1^k, G_2^k, ..., G_l^k (primitivno) rekurzivne funkcije, a R_1^k, R_2^k, ..., R_l^k u parovima disjunktne (primitivno) rekurzivne relacije, sve iste mjesnosti.

Tada je i funkcija $F := \mathrm{if}\{R_1 : G_1, R_2 : G_2, \ldots, R_l : G_l, G_0\}$ također (primitivno) rekurzivna.

Ovdje ne smijemo ispustiti G_0, jer $\otimes^k$ nije (primitivno) rekurzivna!

Dokaz. $R_0 := \left(\bigcup_{i=1}^{l} R_i\right)^c$ je (primitivno) rekurzivna po propozicijama 2.44 i 2.39.
Tvrdimo da je

$$F = \chi_{R_0} \cdot G_0 + \chi_{R_1} \cdot G_1 + \chi_{R_2} \cdot G_2 + \cdots + \chi_{R_l} \cdot G_l, \tag{2.60}$$

dakle dobivena je 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 (2.60) u $\vec{x}$. Ako je $\vec{x}$ u nekoj R_i, tada je $\chi_{R_i}(\vec{x}) = 1$, pa je i-ti (brojeći od nule) pribrojnik u (2.60) jednak $G_i(\vec{x})$. Štoviše, jer su uvjeti u parovima disjunktni, $\vec{x} \notin R_j$ za sve $j \in [1 \cdot\cdot\, l] \setminus \{i\}$, dok iz definicije R_0 slijedi također $\vec{x} \notin R_0$ — što znači da 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 Morganovu pravilu nije ni u njihovoj uniji, dakle $\vec{x} \in R_0$, pa je početni pribrojnik u (2.60) 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. □

Domena funkcije dobivene grananjem (2.56) je još kompliciranija od domene funkcije dobivene kompozicijom (2.4). I ovdje zato pišemo definiciju u stilu napomene 1.4,

$$F(\vec{x}) :\simeq \begin{cases} G_1(\vec{x}), & R_1(\vec{x}) \\ & \vdots \\ G_0(\vec{x}), & \text{inače}, \end{cases} \tag{2.61}$$

uz dogovor da izraz na desnoj strani ima vrijednost samo za one $\vec{x}$ za koje izraz u i-tom retku ima vrijednost, ako uvjet u tom retku vrijedi, a za one $\vec{x}$ za koje izraz u zadnjem retku ima vrijednost, ako nijedan od prethodnih uvjeta ne vrijedi. Dakle, ne zahtijevamo (kao prije) da *svaki* podizraz izraza na desnoj strani ima vrijednost, 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 2.46, dokazat ćemo da *konačnom promjenom* („editiranjem") vrijednosti ne možemo pokvariti izračunljivost funkcije.

Lema 2.47: Svaka jednočlana brojevna relacija je primitivno rekurzivna.

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

$$R(\vec{x}) \iff \vec{x} \in \{\vec{c}\} \iff \vec{x} = \vec{c} \iff x_1 = c_1 \wedge \cdots \wedge x_k = c_k, \tag{2.62}$$

a svaki pojedini konjunkt $(x_i = c_i)$ predstavlja primitivno rekurzivnu relaciju, karakteristične funkcije $\chi_= \circ (I_i^k, C_{c_i}^k)$, primitivno rekurzivne po korolaru 2.42, propoziciji 2.21 i definiciji 2.15. Dakle R je primitivno rekurzivna po propoziciji 2.44. □

Korolar 2.48: Svaka konačna brojevna relacija je primitivno rekurzivna.

Dokaz. Odmah po propoziciji 2.44 i lemi 2.47, jer je $\{\vec{c}_1, \vec{c}_2, \ldots, \vec{c}_l\} = \bigcup_{i=1}^{l} \{\vec{c}_i\}$. □

Propozicija 2.49: Neka je $k \in \mathbb{N}_+$, neka je G^k (primitivno) rekurzivna funkcija te 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 2.19.

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 $\vec{c}_1, \vec{c}_2, \ldots, \vec{c}_l$. Ako je taj skup prazan, tada je $F = G$ pa je (primitivno) rekurzivna po pretpostavci. Inače je

$$F(\vec{x}) = \begin{cases} F(\vec{c}_1), & \vec{x} = \vec{c}_1 \\ & \vdots \\ F(\vec{c}_l), & \vec{x} = \vec{c}_l \\ G(\vec{x}), & \text{inače} \end{cases} = \begin{cases} C^k_{F(\vec{c}_1)}(\vec{x}), & \vec{x} \in \{\vec{c}_1\} \\ & \vdots \\ C^k_{F(\vec{c}_l)}(\vec{x}), & \vec{x} \in \{\vec{c}_l\} \\ G(\vec{x}), & \text{inače} \end{cases}, \tag{2.63}$$

dakle F je dobivena grananjem iz funkcija $C^k_{F(\vec{c}_i)}$ (primitivno rekurzivnih po propoziciji 2.21, jer je $F(\vec{c}_i) \in \mathbb{N}$ zbog totalnosti od F), funkcije G (primitivno) rekurzivne po pretpostavci propozicije, i iz uvjeta $\{\vec{c}_i\}$ primitivno rekurzivnih po propoziciji 2.47. Po teoremu 2.46, F je također (primitivno) rekurzivna. □

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

Korolar 2.50: 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^k_0 razlikuju samo na konačnom skupu $\mathcal{D}_g$, pa tvrdnja slijedi iz propozicija 2.49 i 2.21. □

2.4.2. Programiranje s ograničenim petljama

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 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 $G_1, \ldots, G_l$, 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 smo dobili mogućnost da broj koraka izračunavanja prenesemo kao ulazni podatak. Slikovito, pretvorili smo $G_i(\vec{x})$ u $G(\vec{x}, i)$, pa možemo reći da smo „dignuli supskript" (ili „spustili superskript", ako se radi o mjesnosti) na razinu ulaznog podatka. U ovoj točki napravit 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 2.51: Jedan primjer smo već vidjeli: sa 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 dinamičke strane imamo 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 (dok je C_n^k definirana kompozicijom). I ovdje možemo „definirati"

$$I_{k+1}^{k+1}(\vec{x}, 0) = C_0^k(\vec{x}), \qquad I_{k+1}^{k+1}(\vec{x}, y+1) = Sc\big(I_{k+1}^{k+1}(\vec{x}, y)\big), \tag{2.64}$$

odnosno $I_{k+1}^{k+1} = C_0^k \mathbin{\text{PR}} Sc \circ I_{k+2}^{k+2}$, ali I_{k+2}^{k+2} nije ni po čemu osnovnija od I_{k+1}^{k+1} (jer su obje inicijalne), pa nam ta tehnika ovdje ne pomaže. ◁

Evo malo kompliciranijeg primjera: lema 2.43 kaže da je za svaki k funkcija add^k primitivno rekurzivna; ako joj damo k brojeva, ili je komponiramo s k izračunljivih funkcija, ona će ih zbrojiti (funkcije će zbrojiti na presjeku njihovih domena). Sada ćemo promotriti dinamičku varijantu: operator $\sum$ koji za unaprijed zadanu *totalnu* funkciju G prima broj koji kaže koliko prvih njenih vrijednosti (u nekom kontekstu) treba zbrojiti.

Napomena 2.52: U ostatku ove točke dokazat ćemo šest rezultata o ograničenom programiranju: neke neposredno primitivnom rekurzijom, a neke kompozicijom, s već dobivenim funkcijama. Svi će oni imati isto sučelje: (primitivno) rekurzivnu k-mjesnu funkciju G ili relaciju R, kontekst $\vec{x}$ i granicu y; te će biti definirani iteracijom po svim $i < y$.

U primjenama, najčešće će granica ovisiti o kontekstu: $i < B(\vec{x})$ za neku izračunljivu funkciju B. To samo znači da ćemo primijeniti kompoziciju s B na mjestu zadnjeg argumenta y, što ostaje (primitivno) rekurzivno ako je B takva.

Drugo, granica može biti uključena: $i \le B(\vec{x})$ je ekvivalentno s $i < Sc\big(B(\vec{x})\big)$, što samo znači da je granica $Sc \circ B$, koja je (primitivno) rekurzivna ako je B takva.

I treće, kontekst može biti prazan: u slučaju $k = 1$, ulaz za G odnosno R je samo i. Tada je primitivna rekurzija degenerirana, što također ne smeta jer ona čuva primitivnu rekurzivnost po propoziciji 2.22, a rekurzivnost po korolaru 2.36. ◁

Lema 2.53: 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

$$F_1(\vec{x}, y) := \sum_{i<y} G(\vec{x}, i), \qquad F_2(\vec{x}, y) := \prod_{i<y} G(\vec{x}, i). \tag{2.65}$$

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

$$F_1(\vec{x}, 0) := 0 \qquad F_2(\vec{x}, 0) := 1 \tag{2.66}$$

$$F_1(\vec{x}, y+1) := F_1(\vec{x}, y) + G(\vec{x}, y) \qquad F_2(\vec{x}, y+1) := F_2(\vec{x}, y) \cdot G(\vec{x}, y) \tag{2.67}$$

Tada je $F_2 = C_1^{k-1} \mathbin{\text{PR}} H$ (ili $1 \mathbin{\text{PR}} H$ za $k = 1$), gdje je $H(\vec{x}, y, z) = z \cdot G(\vec{x}, y)$ (primitivno) rekurzivna pa je i F_2 takva. Sasvim analogno za F_1. □

U uvodnom učenju programiranja, obično se prije sumiranja nauči *brojiti* elemente koji zadovoljavaju neko svojstvo: inicijaliziramo brojač na 0, prolazimo kroz sve elemente koji dolaze u obzir, i za svaki koji zadovoljava svojstvo, inkrementiramo brojač. Ali kako smatramo

bool podskupom od $\mathbb{N}$, uz *false* = 0 i *true* = 1, brojenje je upravo *sumiranje istinitosnih vrijednosti*. Naredbu `if uvjet: brojač += 1` možemo zapisati kao `brojač += uvjet`, čime algoritam za brojenje postaje obični algoritam za sumiranje. Formalizirajmo to.

Lema 2.54: Neka je $k \in \mathbb{N}_+$ te R^k (primitivno) rekurzivna relacija. Tada je funkcija F^k zadana s

$$F(\vec{x}, y) := \operatorname{card}\{i \in \mathbb{N} \mid i < y \wedge R(\vec{x}, i)\}, \tag{2.68}$$

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

Dokaz. Prema pretpostavci, karakteristična funkcija χ_R^k je (primitivno) rekurzivna. Sada tvrdnja slijedi iz leme 2.53, jer tvrdimo da vrijedi

$$(\# i < y)\, R(\vec{x}, i) = \sum_{i<y} \chi_R(\vec{x}, i). \tag{2.69}$$

Doista, za svaki $\vec{x} \in \mathbb{N}^k$, ako skup $S := [0 \cdot\cdot\, y\rangle$ rastavimo na dva dijela,

$$S_1 := \{i \in S \mid R(\vec{x}, i)\} \quad \text{i} \quad S_2 := \{i \in S \mid \neg R(\vec{x}, i)\}, \tag{2.70}$$

tada je $\sum_{i \in S} \chi_R(\vec{x}, i) = \sum_{i \in S_1} 1 + \sum_{i \in S_2} 0 = \operatorname{card} S_1$. □

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 ispitujući postoji li $d \in [2 \cdot\cdot\, n\rangle$ (ili $d \in \left[2 \cdot\cdot\, \lfloor\sqrt{n}\rfloor\right]$) 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 brže pretraživati konačne skupove, a i dobiti odgovor za prebrojive skupove *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 se ne bavimo performansama algoritama.

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

Propozicija 2.55: Neka je $k \in \mathbb{N}_+$ te R^k (primitivno) rekurzivna relacija.

Tada su takve i relacije P^k i Q^k, zadane s

$$P(\vec{x}, y) :\Longleftrightarrow (\exists i < y)\, R(\vec{x}, i), \qquad Q(\vec{x}, y) :\Longleftrightarrow (\forall i < y)\, R(\vec{x}, i). \tag{2.71}$$

Dokaz. Kao u dokazu leme 2.54, fiksirajmo $\vec{x}$ i uvedimo oznake $S := [0 \cdot\cdot\, y\rangle$ te $S_R := \{i \in S \mid R(\vec{x}, i)\} \subseteq S$. Sada vrijedi

$$P(\vec{x}, y) \Longleftrightarrow S_R \neq \varnothing \Longleftrightarrow \operatorname{card} S_R > 0 \Longleftrightarrow (\# i < y)\, R(\vec{x}, i) \in \mathbb{N}_+ \text{ te} \tag{2.72}$$

$$Q(\vec{x}, y) \Longleftrightarrow S_R = S \Longleftrightarrow \operatorname{card} S_R = \operatorname{card} S \Longleftrightarrow (\# i < y)\, R(\vec{x}, i) = y \tag{2.73}$$

(za smjer $\operatorname{card} S_R = \operatorname{card} S \Rightarrow S_R = S$ koristimo rezultat iz teorije skupova da konačan skup ne može biti ekvipotentan svom pravom podskupu), pa tvrdnja propozicije slijedi iz leme 2.54, primjera 2.28 (za P) i korolara 2.42 (za Q). Uočimo da je (2.72) dinamizirani (2.55), zbog veze (2.69) između brojenja i sumiranja. □

Još ne znamo ništa o izračunljivosti *neograničene* kvantifikacije (*projekcije*) relacije. 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, objasnit ćemo u poglavlju 7.

2.4.3. Ograničena minimizacija

Definicija 2.56: Neka je $k \in \mathbb{N}_+$ te R^k relacija. Za funkciju F^k definiranu s

$$F(\vec{x}, y) := \mu i\big(i < y \to R(\vec{x}, i)\big) =: (\mu i < y)\, R(\vec{x}, i), \tag{2.74}$$

kažemo da je dobivena *ograničenom minimizacijom* relacije R. ◁

Napomena 2.57: U (2.74) smo napisali := umjesto :≃, jer izraz u sredini uvijek ima vrijednost: kondicional će sigurno nekada postati istinit, najkasnije za $i = y$, jer će tada antecedens $i < y$ postati lažan. Zato će $F(\vec{x}, y)$ biti najmanji broj $i < y$ koji zadovoljava $R(\vec{x}, i)$ ako takav postoji, a inače će biti $F(\vec{x}, y) = y$.

Dakle, F je uvijek totalna funkcija. Štoviše, ako je R rekurzivna, F će biti parcijalno rekurzivna — jer je dobivena (neograničenom) minimizacijom relacije P zadane s $P(\vec{x}, y, i) :\Leftrightarrow i < y \to R(\vec{x}, i)$, rekurzivne zbog primjera 2.29 i propozicije 2.41. Ukratko, ograničenom minimizacijom rekurzivne relacije dobivamo rekurzivnu funkciju. ◁

Dokazali smo da ograničena minimizacija čuva rekurzivnost. Važno je da čuva i *primitivnu* rekurzivnost, što ćemo sada dokazati. U tom dokazu, naravno, ne možemo koristiti minimizaciju, već ćemo traženu funkciju definirati primitivnom rekurzijom.

Propozicija 2.58: Neka je $k \in \mathbb{N}_+$ te R^k primitivno rekurzivna relacija. Tada je i funkcija F^k, dobivena ograničenom minimizacijom relacije R, također primitivno rekurzivna.

Dokaz. Za $y = 0$, vrijednost funkcije F je očito 0, jer je antecedens kondicionala odmah na početku lažan ($0 \not< 0$) pa je kondicional istinit. Sada pretpostavimo da imamo već izračunanu vrijednost $z := F(\vec{x}, y)$ (zbog napomene 2.57 je $z \le y$), i pitamo se koliko je $F(\vec{x}, y + 1)$. Imamo tri slučaja.

Ako je $z < y$, to znači da postoji „dobar" $i < y < y + 1$ takav da vrijedi $R(\vec{x}, i)$, i z je najmanji takav i, pa je očito $F(\vec{x}, y + 1)$ također jednako z. Primijetimo da ovdje sigurno vrijedi $R(\vec{x}, z)$.

Ako je $z = y$, to znači da ne postoji dobar $i < y$, pa se pitamo vrijedi li $R(\vec{x}, y)$. Ako vrijedi, y je najmanji dobar $i < y + 1$, pa je $F(\vec{x}, y + 1) = y$, što je opet jednako z. Inače, nismo našli dobar $i < y + 1$, pa je $F(\vec{x}, y + 1) = y + 1$ u skladu s napomenom 2.57.

Sve u svemu, vidimo da je $F = C_0^{k-1} \mathbin{\text{PR}} H$ (ili $0 \mathbin{\text{PR}} H$ za $k = 1$), gdje je H, zadana s

$$H(\vec{x}, y, z) = \begin{cases} z, & R(\vec{x}, z) \\ Sc(y), & \text{inače} \end{cases}, \tag{2.75}$$

primitivno rekurzivna po teoremu 2.46 — pa je i F takva. □

3. Univerzalna izračunljivost

Sada već možemo i prilično komplicirane funkcije dokazati primitivno rekurzivnima. Sljedeći veliki zalogaj koji ćemo uzeti je kodiranje konačnih nizova prirodnih brojeva (skupa $\mathbb{N}^*$).

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, implementirat ćemo funkciju U koja prima izračunavanje koje je stalo i vraća njegov izlazni podatak. Ubuduće 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 pamtiti samo do završne konfiguracije kao konačni nizovi konfiguracija. Same pak konfiguracije također se mogu pamtiti kao konačni nizovi — vrijednost programskog brojača i sadržaj samo relevantnih registara. Vidjet ćemo da se mnogi objekti koje ćemo željeti kodirati mogu prikazati kao konačni nizovi drugih objekata, tako da ako već imamo kodiranje tih jednostavnijih objekata, kodiranje $\mathbb{N}^*$ dat će nam odmah mogućnost kodiranja složenih objekata.

Drugi razlog je preciznost 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 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 to su familije algoritama $\mathcal{A}^k, k \in \mathbb{N}_+$ (i eventualno konstanta $\mathcal{A}^0$). Naše kodiranje je upravo takva familija $\mathsf{Code}^k, k \in \mathbb{N}_+$, s konstantom 1 koja igra ulogu Code^0. Prvo preciznije definirajmo 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 $\mathbb{N}\mathcal{K} : \mathcal{K} \to \mathbb{N}$, kojoj je slika $\mathcal{I}_{\mathbb{N}\mathcal{K}}$ (skup svih kodova, gledan kao jednomjesna brojevna relacija) rekurzivna, a parcijalni inverz (lijevi inverz s obzirom na kompoziciju) $\mathbb{N}\mathcal{K}^{-1} : \mathbb{N} \rightharpoonup \mathcal{K}$, s domenom $\mathcal{D}_{\mathbb{N}\mathcal{K}^{-1}} = \mathcal{I}_{\mathbb{N}\mathcal{K}}$, je također izračunljiva funkcija. ◁

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, ne možemo reći „$\mathbb{N}\mathcal{K} \in \mathcal{C}omp$"). 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 $g : \mathcal{K} \rightharpoonup \mathcal{K}$, 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, a rezultat (ako je definiran, odnosno ako je $\kappa \in \mathcal{D}_g$) kodira natrag funkcijom $\mathbb{N}\mathcal{K}$. Prateća je funkcija uvijek brojevna; ako je izračunljiva u nekom smislu (recimo, parcijalno rekurzivna) i kodiranje relativno kanonsko, prirodno je smatrati funkciju g izračunljivom u tom istom smislu. Primijetimo da $\mathbb{N}\mathcal{K} \circ g \circ \mathbb{N}\mathcal{K}^{-1}$ ne možemo smatrati simboličkom definicijom prateće funkcije, jer to nije kompozicija brojevnih funkcija. Moramo nekako drugačije, koristeći samo prirodne brojeve, zapisati prateću funkciju. Ako to uspijemo izvesti za velik broj 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 posebni slučaj pratećih funkcija, za $\mathbb{N}\mathtt{bool}(\mathit{false}) := 0$, $\mathbb{N}\mathtt{bool}(\mathit{true}) := 1$), pa čak i 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 (ostavivši nepromijenjenima ulaze koji već jesu prirodni brojevi), 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 particiju od $\mathcal{K}$) koju onda zovemo *konstruktor*, a koordinatne funkcije njenog inverza (*getters* ili *komponente*) služe kao podloga za 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 $\operatorname{card}\mathcal{K} \leq \operatorname{card}\mathbb{N} = \aleph_0$. Ovo je izuzetno važno, jer opravdava intuiciju da samo konačni i prebrojivi skupovi imaju *totalnu reprezentaciju*: sve njihove elemente možemo reprezentirati u (po volji velikom) računalu.

Neegzaktnosti 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 [Chen16], ne možemo u računalu reprezentirati proizvoljni realni broj — i to nema veze s ograničenom veličinom naših računala. Ograničenost memorije samo predstavlja razliku između konačnih i prebrojivih skupova; razlika prebrojivih i neprebrojivih skupova mnogo je veća i njen prijelaz zahtijeva sasvim nove paradigme. ◁

3.1. Kodiranje konačnih nizova

Sada se možemo pozabaviti samom implementacijom kodiranja $\mathbb{N}^*$. Kako bismo najlakše kodirali $\vec{x} = (x_1, x_2, \ldots, x_k)$ 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, promatramo injekciju $p(x, y) := 2^x \cdot 3^y$ te koristimo osnovni teorem aritmetike (rastav na prim-faktore) da bismo iz $p(x, y)$ natrag dobili x i y. To možemo proširiti na proizvoljnu mjesnost jer prim-brojeva ima beskonačno mnogo: uzmemo ih dovoljno redom po veličini, potenciramo odgovarajućim eksponentima i pomnožimo — ali to nije kodiranje skupa $\mathbb{N}^*$. Naime, takvo preslikavanje nije injekcija, jer se primjerice $(1, 2, 0, 0)$ 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 tog postupka 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: Označimo prim-brojeve redom s $p_0 := 2$, $p_1 := 3$, $p_2 := 5$, ...

Definiramo $\langle\rangle := 1$ te za svaki $k \in \mathbb{N}_+$ definiramo

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

Time je definirana funkcija $\langle\cdots\rangle : \mathbb{N}^* \to \mathbb{N}$. To doduše nije brojevna 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 2.43.

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

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

$$\mathsf{Code}^k = \mathsf{mul}^k \circ \big(\mathsf{pow} \circ (C_2^k, \mathsf{Sc} \circ I_1^k), \mathsf{pow} \circ (C_3^k, \mathsf{Sc} \circ I_2^k), \ldots, \mathsf{pow} \circ (C_{p_{k-1}}^k, \mathsf{Sc} \circ I_k^k)\big), \tag{3.2}$$

iz koje po propozicijama 2.21 i 2.16, primjeru 2.13 te lemi 2.43 slijedi tvrdnja. □

Simbolička definicija je napisana kako bi bilo sasvim 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 znati da postoji barem k prim-brojeva te da su konstante s tim vrijednostima izračunljive.

Propozicija 3.5: Preslikavanje $\langle\cdots\rangle : \mathbb{N}^* \to \mathbb{N}$ 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 $k - 1 \in \mathbb{N}$, pa je $\langle\vec{x}\rangle$ djeljiv s p_{k-1}, što $\langle\vec{y}\rangle$ nije ($p_{k-1} \notin \{p_0, p_1, \ldots, p_{l-1}\}$, pa ne dijeli njihov umnožak), kontradikcija. Analogno ne može biti $k < l$ — dakle je $k = l$.

Za svaki $i \in [1 \cdot\cdot\, k]$, tvrdimo da ne može biti $x_i > y_i$. U suprotnom, kōd bi bio djeljiv s $p_{i-1}^{y_i+1}$, pa bi nakon dijeljenja lijeva strana bila djeljiva s p_{i-1}, a desna ne bi. Analogno se vidi da nije ni $x_i < y_i$, dakle mora biti $x_i = y_i$. Time smo dokazali $\vec{x} = \vec{y}$. □

Sljedeće bismo trebali pokazati da je $\mathcal{I}_{\langle\ldots\rangle} =: \mathsf{Seq}$ 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 precizirajmo u kakvom obliku tražimo naš inverz.

Naglasimo odmah da tražimo primitivno rekurzivne, dakle totalne funkcije. Njihovo djelovanje bit će parcijalno specificirano na skupu Seq, iako će se napisani algoritmi moći izvršiti na svakom prirodnom broju.

Da bismo odredili konačni niz iz kojeg je dobiven kōd c, moramo prvo odrediti njegovu duljinu. Dakle, tražimo totalnu funkciju lh takvu da za svaki $c \in \mathsf{Seq}$, $\mathsf{lh}(c)$ bude duljina konačnog niza čiji je c kōd. Takvih funkcija ima neprebrojivo mnogo jer je Seq^c beskonačan (recimo, sadrži sve neparne brojeve veće od 3), 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 $\mathsf{lh}(c) =: k$, na prvi pogled imamo tipičnu situaciju algoritma s više izlaza: od jednog broja c trebamo dobiti k njih, pa bismo u skladu s napomenom 1.1 to trebali reprezentirati pomoću koordinatnih funkcija $\mathsf{part}_1, \mathsf{part}_2, \ldots, \mathsf{part}_k$. Ipak, to nema baš 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 kōd izračunavanja koje stane preslikava u njegov rezultat, bit će potrebno odabrati *posljednju* konfiguraciju u konačnom nizu. A čak i da imamo izračunljive funkcije part_i za sve i, iz toga ne slijedi da je preslikavanje $c \mapsto \mathsf{part}_{\mathsf{lh}(c)}(c)$ 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 $\mathsf{part}(c, i)$ (skraćena oznaka $c[i]$) bude ono što smo bili nazvali $\mathsf{part}_{i+1}(c)$ — pomaknuli smo indekse za 1 jer želimo shvatiti konačne nizove kao polja (*arrays*), koja se u većini modernih programskih jezika indeksiraju od nule. Bitno nam je da to vrijedi samo za $c \in \mathsf{Seq}$ i $i \in [0 \cdot\cdot \mathsf{lh}(c)\rangle$ — za ostale uređene parove mora biti nekako definirana jer je primitivno rekurzivna, ali nije nam bitno kako točno. Pokazat će se da za $c \in \mathsf{Seq}$ i $i \geq \mathsf{lh}(c)$ 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 $\mathsf{part}_1, \ldots, \mathsf{part}_k$ u drugom, zapravo modeliramo ono što jezik C zove struktura (`struct`) s k članova. C ima strukture kao zasebne tipove podataka prvenstveno jer one mogu sadržavati podatke različitih tipova. Kako mi sve kodiramo prirodnim brojevima, to nam neće bitno trebati. ◁

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 duljina polja. 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 sa sadržajem spremnika (na primjer, Python ima ugrađenu funkciju `len`) pa je zato i mi kodiramo tako da ju je moguće odrediti iz koda.

Funkcije Code^k su prikladne ako imamo fiksnu mjesnost i možemo argumente zadati posebno. Što dobijemo ako primijenimo ideju dinamizacije na tu familiju funkcija? Argumenti su tada zadani nekom funkcijom G te posebni argument y kaže koliko ih ima (usporedite s točkom 2.4.2). Operator koji dinamički kodira zadani broj vrijednosti neke funkcije zovemo operatorom *povijesti*, jer često argument y predstavlja vrijeme, odnosno broji korake u nekom postupku (konkretno, trebat će nam za brojenje koraka u 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

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

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

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

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

Može biti i $k = 1$; jedan važni slučaj je $G := Z$. Recimo,

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

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

Sada bismo htjeli dokazati rezultat analogan onom u lemi 2.53, samo za operator povijesti. Kao što smo već rekli, to će zahtijevati ponešto teorije brojeva.

3.1.1. Potrebni rezultati iz teorije brojeva

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

Sjetimo se napomene 2.26: često brojevnoteorijske funkcije nisu definirane u nuli, koju zato zamjenjujemo jedinicom, pišući $n^{\cdot}$ za n ako je pozitivan, a 1 ako je $n = 0$.

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

Prije dokaza napomenimo da je nula problematična u operaciji $/\!\!/$, ali ne i u operaciji $\bmod$: definicija $x \bmod 0 := x$ uobičajena je u modernoj teoriji brojeva.

Dokaz. Cjelobrojno dijeljenje zapravo je ponovljeno oduzimanje, onoliko puta koliko se može. Dakle, za zadane x i y, tražimo najveći t takav da je $t\cdot y \le x$. Relacija $t\cdot y \le x$ jest 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 $t\cdot y \le x$ postane laž, ostat će laž za sve veće t. Kod takvih relacija, „prva laž" i „posljednja istina" su susjedne, dakle možemo naći prethodnik najmanjeg t za kojeg vrijedi suprotno.

Ipak, to je neograničena minimizacija, koja neće dati primitivno rekurzivnu funkciju — možemo li je nekako ograničiti? Svakako: $y^{\cdot} \ge 1$ znači $t\cdot y^{\cdot} \ge t$ za svaki t, posebno $(x+1)\cdot y^{\cdot} \ge x+1 > x$. 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

$$x /\!\!/ y = \mathsf{pd}\big((\mu t \le x)(t\cdot y > x)\big). \tag{3.6}$$

Za $y > 0$, samo trebamo formalizirati navedeni argument: označimo $z := x /\!\!/ y$. Tada je $zy \le x$ (i $ty \le zy \le x$ za sve $t \le z$), a $(z+1)y > x$, dakle $z+1$ je najmanji t koji pomnožen s y daje broj veći od x. S druge strane je $z \le x /\!\!/ 1 = x$, pa je $z+1 \le x+1$. Ako je $z+1 \le x$, 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 $x \mathbin{/\!/} 0 = \left\lfloor \frac{x}{1} \right\rfloor = x$, a na desnoj je prethodnik minimizacije po $t \leq x$, relacije $t \cdot 0 = 0 > x$. Ta relacija je uvijek lažna (nemamo negativnih brojeva), pa minimizacija daje $x + 1$, a njen prethodnik onda daje upravo x, kao što i treba.

Ostatak: tvrdimo da je

$$x \bmod y = x \dot{-} x \mathbin{/\!/} y \cdot y \tag{3.7}$$

(redoslijed izvođenja operacija je: $\mathbin{/\!/}$, pa $\cdot$, pa $\dot{-}$). Za $y > 0$ 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 $x \dot{-} x \mathbin{/\!/} 0 \cdot 0 = x \dot{-} x \cdot 0 = x \dot{-} 0 = x$. □

Korolar 3.10: Djeljivost je primitivno rekurzivna relacija.

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

$$y \mid x \iff x \bmod y = 0. \tag{3.8}$$

Zaista, za $y > 0$, postojanje broja z takvog da je $z \cdot y = x$ (definicija djeljivosti) zapravo znači da je $z = x \mathbin{/\!/} y$, pa je $x \bmod y = x \dot{-} z \cdot y = x \dot{-} x = 0$. U drugom smjeru, ako je $x \dot{-} x \mathbin{/\!/} y \cdot y = 0$, iz toga slijedi (kao i prije, vrijedi $x \mathbin{/\!/} y \cdot y \leq x$) $x = x \mathbin{/\!/} y \cdot y$, pa postoji $z := x \mathbin{/\!/} y$ 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 $0 \cdot z = x$" upravo znači da je $x = 0$. □

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

Dokaz. Prim-brojevi imaju točno dva prirodna djelitelja, a za pozitivne x, svaki djelitelj od x je manji ili jednak x:

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

Za nulu će tom metodom ispasti da ima jedan djelitelj (samu sebe), a zapravo ih ima beskonačno mnogo — ali i tako je $0 \notin \mathbb{P}$. Sada tvrdnja slijedi iz korolara 3.10 i leme 2.54. □

U „stvarnom životu", provjera je li $x \in \mathbb{P}$ odvija se drugačije: nulu, jedinicu i dvojku te sve ostale parne brojeve odvojimo kao posebne 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. Početkom ovog stoljeća čak je pronađen *polinomni* algoritam za provjeru je li zadani broj prim-broj, ali je uvelike kompliciraniji. Kako se mi ovdje ne opterećujemo 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 čija je slika $\mathbb{P}$, 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. 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 koja prima $(n\ i)\ p_n$, i mora vratiti p_{n+1}. Je li ta funkcija izračunljiva?

Kako uopće znamo da je nextprime *totalna*, odnosno da njome možemo 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:

$$\mathsf{nextprime}(p) := \mu q(q \in \mathbb{P} \wedge q > p). \tag{3.10}$$

Upravo napisani izraz zapravo kaže da je $\mathsf{nextprime}^1$ *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 2.36.

Za primitivnu rekurzivnost, moramo nekako ograničiti minimizaciju, odnosno moramo „isprogramirati" dokaz da je $\mathbb{P}$ beskonačan. Uzmimo Euklidov dokaz: ako imamo $p_0, p_1, \ldots, p_n \in \mathbb{P}$, 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, $m = \mathsf{Sc}\big(\overline{\mathsf{Z}}\big(\mathsf{Sc}(n)\big)\big)$ — ali nažalost još ne znamo da je funkcija $\overline{\mathsf{Z}}$ 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 je u tome da ne trebamo pomoću n i p_n izračunati baš m, nego samo neki broj od kojeg je m manji. Ionako nećemo tražiti njegove prim-djelitelje, nego će nam on samo poslužiti kao gornja granica za traženje sljedećeg prim-broja nakon p_n. To znači da umjesto primorijela možemo upotrijebiti faktorijel, za koji znamo da je primitivno rekurzivan (primjer 2.24), a vrijedi $p_n! \geq p_0 \cdot p_1 \cdots p_n$ jer je $p_n!$ umnožak svih brojeva na desnoj strani i eventualno još nekih — koji iznose 1 ili više, pa ne smanjuju umnožak.

$$\mathsf{nextprime}(p) = (\mu q \leq p! + 1)(q \in \mathbb{P} \wedge q > p) \tag{3.11}$$

Dakle $\mathsf{nextprime}^1$ je dobivena ograničenom minimizacijom primitivno rekurzivne relacije do primitivno rekurzivne granice, pa je primitivno rekurzivna po napomeni 2.52 i propoziciji 2.58. A tada je i $\mathsf{prime} = 2 \mathbin{\mathbb{R}} \mathsf{nextprime} \circ \mathsf{I}_2^2$ primitivno rekurzivna po propoziciji 2.22. □

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

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

Dokaz. Opet, tražimo najveći broj t takav da ${p_i}^t \mid n^{\cdot}$. 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 $n^{\cdot} \geq {p_i}^t \geq {p_0}^t = 2^t > t$ (Cantorov osnovni teorem za konačne skupove), pa je t dovoljno tražiti do $n^{\cdot}$ isključivo, ili do n jer su za $n = 0$ ionako svi eksponenti 0. Sve u svemu, vrijedi

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

pa je ex primitivno rekurzivna. □

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 $\vec{x} = (x_1, x_2, \ldots, x_k) \in \mathbb{N}^*$, uz oznaku $c := \langle \vec{x} \rangle$ te pokratu $c[i] := \mathsf{part}(c, i)$ vrijedi:

1. $\mathsf{lh}(c) = k$;
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

$$\mathsf{lh}(c) := (\# d \leq c)(d \in \mathbb{P} \wedge d \mid c), \tag{3.13}$$

$$c[i] := \mathsf{part}(c, i) := \mathsf{pd}\big(\mathsf{ex}(c, i)\big), \tag{3.14}$$

zadovoljavaju sve uvjete. Prije svega, primitivno su rekurzivne: lh je dobivena ograničenim brojenjem primitivno rekurzivne relacije do primitivno rekurzivne granice, a $\mathsf{part} = \mathsf{pd} \circ \mathsf{ex}$.

Što se tiče tvrdnje 1, prema definiciji lh broji prim-djelitelje od c — a kako je $c = \langle \vec{x} \rangle = 2^{x_1+1} \cdot 3^{x_2+1} \cdots p_{k-1}^{x_k+1}$ 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 $\mathsf{ex}(c, i) = x_{i+1} + 1$, pa je $c[i] = \mathsf{pd}(x_{i+1} + 1) = x_{i+1}$.

Za tvrdnju 3, neka je $i \geq k$ proizvoljan. Jedini prim-djelitelji od c su $p_j, j \in [0 \cdot\cdot k\rangle$, među kojima nije p_i. Dakle $(\mu t < c)(p_i^t \nmid c) = 1$, pa je $\mathsf{ex}(c, i) = \mathsf{pd}(1) = 0$ te je i $c[i] = \mathsf{pd}(0) = 0$. □

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

3.2. Funkcije definirane kodiranjem konačnih nizova

Lema 3.15 (Lema o povijesti): Neka je $k \in \mathbb{N}_+$ i G^k (primitivno) rekurzivna funkcija. Tada je i $\overline{G}$ (primitivno) rekurzivna.

(Vrijedi i obrat ove leme, ali nam neće biti potreban. Zapravo, i u lemi 2.53 vrijedi obrat za F_1, dok za F_2 ne vrijedi. Zgodna je vježba pokušati to dokazati.)

Dokaz. Da bismo zapisali $\overline{G}$ primitivno rekurzivno pomoću G, uvrstimo (3.1) u (3.3):

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

Ako i-ti faktor u (3.15) označimo s $H(\vec{x}, i)$, tada je $\mathsf{pow} \circ (\mathsf{prime} \circ I_k^k, \mathsf{Sc} \circ G)$ simbolička definicija od H^k kompozicijom iz funkcija za koje je već dokazano da su primitivno rekurzivne, i funkcije G koja je (primitivno) rekurzivna po pretpostavci — pa je H (primitivno) rekurzivna. Sada je $\overline{G}$ (primitivno) rekurzivna po lemi 2.53. □

Napokon možemo dokazati primitivnu rekurzivnost slike kodiranja.

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

Dokaz. Formalno, definicija slike (totalne funkcije) daje

$$\mathsf{Seq}(c) \iff (\exists\, \vec{x} \in \mathbb{N}^*)(c = \langle \vec{x} \rangle), \tag{3.16}$$

i način za određivanje $\vec{x}$ 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

$$\mathsf{Seq}(c) \iff c = \overline{\mathsf{part}}\big(c, \mathsf{lh}(c)\big); \tag{3.17}$$

pritom smjer ($\Leftarrow$) odmah 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 $\vec{x} = (x_1, x_2, \ldots, x_k) \in \mathbb{N}^k \subset \mathbb{N}^*$. Tada je prema propoziciji 3.14(2),

$$\begin{aligned} \overline{\mathsf{part}}\big(c, \mathsf{lh}(c)\big) &= \langle c[0], c[1], \ldots, c[\mathsf{lh}(c) - 1] \rangle = \\ &= \langle x_{0+1}, x_{1+1}, \ldots, x_{\mathsf{lh}(c)-1+1} \rangle = \langle x_1, x_2, \ldots, x_k \rangle = \langle \vec{x} \rangle = c. \end{aligned} \tag{3.18}$$

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

Sada primitivna rekurzivnost slijedi na uobičajeni način: prema propoziciji 3.14 funkcija part je primitivno rekurzivna, prema lemi 3.15 je tada i $\overline{\mathsf{part}}$ primitivno rekurzivna, a onda je $\chi_{\mathsf{Seq}} = \chi_= \circ \big(\mathsf{I}^1_1, \overline{\mathsf{part}} \circ (\mathsf{I}^1_1, \mathsf{lh})\big)$ 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 omogućuje rad s kodiranim skupom kao *klasom*, gdje konstruktori i komponente čine podlogu za sve ostale metode. Striktno, ulogu konstruktora za $\mathbb{N}^*$ igra familija funkcija $\mathsf{Code}^k, k \in \mathbb{N}_+$, dinamizirana kroz operator povijesti, a ulogu komponenata 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 $\langle\cdots\rangle$ odnosno $\overline{\cdots}$ te lh i part prema njihovoj specifikaciji. Pogledajmo jedan primjer.

Primjer 3.17: *Konkatenacija* je preslikavanje $\mathtt{concat} : \mathbb{N}^* \times \mathbb{N}^* \to \mathbb{N}^*$, zadano s

$$\mathtt{concat}\big((x_1, x_2, \ldots, x_k), (y_1, y_2, \ldots, y_l)\big) := (x_1, x_2, \ldots, x_k, y_1, y_2, \ldots, y_l). \tag{3.19}$$

Pomalo neprecizno, ali sasvim jasno, pišemo $\mathtt{concat}(\vec{x}^k, \vec{y}^l) := (\vec{x}, \vec{y})$. 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 $\mathbb{N}\mathtt{concat}$ definirana je samo na $\mathsf{Seq}\times\mathsf{Seq}$, ali u svrhu primitivne rekurzivnosti, to shvaćamo kao parcijalnu specifikaciju. Također, njeno primitivno rekurzivno proširenje često pišemo infiksno kao operaciju $*$.

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

$$\langle \vec{x} \rangle * \langle \vec{y} \rangle = \langle \vec{x}, \vec{y} \rangle. \tag{3.20}$$

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 za duljinu želimo zbroj duljina od $\vec{x}$ i od $\vec{y}$ — a ako operande od $*$ označimo s x i y, prvih $k := \mathsf{lh}(x)$ 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:

$$\mathsf{G}(x, y, i) := \begin{cases} x[i], & i < \mathsf{lh}(x) \\ y[i \dot{-} \mathsf{lh}(x)], & \text{inače} \end{cases}, \tag{3.21}$$

i pomoću nje

$$x * y := \overline{\mathsf{G}}\big(x, y, \mathsf{lh}(x) + \mathsf{lh}(y)\big). \tag{3.22}$$

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

Jednadžbe (3.21) i (3.22) 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 2.46 (primitivno rekurzivna verzija), a onda i funkcije $\overline{\mathsf{G}}$ po lemi 3.15, pa tako i operacije $*$ dobivene iz nje kompozicijom. □

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 nam omogućuje pisanje ograničenih petlji koje prate jedan podatak. Što ako ih želimo pratiti više, recimo l njih? U napomeni 1.1 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 odvojivi, jer je zamislivo da sljedeće vrijednosti svih l podataka ovise o prethodnim vrijednostima svih njih.

Važna situacija u kojoj se taj fenomen 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: najčešće je sljedbenik te vrijednosti u koraku n, ali pri izvršavanju 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 je implementiramo pomoću obične primitivne rekurzije: umjesto l vrijednosti $\vec{a}^l$ 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, ..., G_l^k i H_1^{k+l+1}, H_2^{k+l+1}, ..., H_l^{k+l+1} (primitivno) rekurzivne funkcije. Tada su funkcije F_1^{k+1}, F_2^{k+1}, ..., F_l^{k+1}, zadane s

$$F_i(\vec{x}, 0) := G_i(\vec{x}), \tag{3.23}$$

$$F_i(\vec{x}, y+1) := H_i\big(\vec{x}, y, F_1(\vec{x}, y), F_2(\vec{x}, y), \ldots, F_l(\vec{x}, y)\big), \tag{3.24}$$

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

Dokaz. Kao što smo već rekli, cilj nam je primitivnom rekurzijom prvo dobiti funkciju $F := \mathrm{Code}^l \circ (F_1, F_2, \ldots, F_l)$, iz koje ćemo onda dobiti svaku F_i kompozicijom s funkcijom part. Inicijalizacija: iz (3.23) imamo

$$F(\vec{x}, 0) = \langle F_1(\vec{x}, 0), \ldots, F_l(\vec{x}, 0) \rangle = \langle G_1(\vec{x}), \ldots, G_l(\vec{x}) \rangle =: G(\vec{x}), \tag{3.25}$$

te je $G = \mathrm{Code}^l \circ (G_1, \ldots, G_l)$ (primitivno) rekurzivna kao kompozicija takvih.
Korak: iz (3.24) je

$$\begin{aligned} F(\vec{x}, y+1) &= \langle F_1(\vec{x}, y+1), \ldots, F_l(\vec{x}, y+1) \rangle = \\ &= \big\langle H_1\big(\vec{x}, y, F_1(\vec{x}, y), \ldots, F_l(\vec{x}, y)\big), \ldots, H_l\big(\vec{x}, y, F_1(\vec{x}, y), \ldots, F_l(\vec{x}, y)\big) \big\rangle = \\ &= \langle H_1(\vec{x}, y, F(\vec{x}, y)[0], \ldots, F(\vec{x}, y)[l-1]), \ldots, H_l(\vec{x}, y, F(\vec{x}, y)[0], \ldots, F(\vec{x}, y)[l-1]) \rangle \\ &=: H\big(\vec{x}, y, F(\vec{x}, y)\big) \end{aligned} \tag{3.26}$$

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

Jednakosti (3.25) i (3.26) kažu nam da je $F = G \mathbin{\mathrm{PR}} H$, dakle funkcija F je dobivena primitivnom rekurzijom iz (primitivno) rekurzivnih funkcija, pa je i sama (primitivno) rekurzivna.

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

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, mogli bismo zapisati (3.26) u obliku

$$H(\vec{x}, y, z) := \langle H_1(\vec{a}), H_2(\vec{a}), \ldots, H_l(\vec{a}) \rangle,$$
$$\text{uz pokratu } \vec{a} := \big(\vec{x}, y, z[0], z[1], \ldots, z[l-1]\big).$$

Treba napomenuti da je to samo kraći *zapis* za (3.26), ne uvođenje pomoćnih funkcija — jer tada bi $\vec{a}$ kao funkcija trebala imati više izlaznih podataka te primati kontekst $\vec{x}$, 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-ov 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. ◁

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 najzanimljiviji je 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 $m < n$, a 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

$$F(\vec{x}, y) := G\big(\vec{x}, \overline{F}(\vec{x}, y)\big), \tag{3.27}$$

također (primitivno) rekurzivna.

Po Dedekindovu teoremu rekurzije (pogledajte [VukTS15, str. 60] za detalje; $\overline{F}(\vec{x}, y)$ ovdje kodira $\varphi|_y$) za svaku totalnu funkciju G^k postoji jedinstvena (totalna) funkcija F^k koja zadovoljava jednadžbu (3.27). 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^l imamo dinamički operator povijesti. Dakle, trebamo dobiti $\overline{F}$ primitivnom rekurzijom (degeneriranom u slučaju $k = 1$). Inicijalizacija: svaka povijest počinje kodom praznog niza,

$$\overline{F}(\vec{x}, 0) = \langle\rangle = 1. \tag{3.28}$$

Za korak, moramo izraziti $\overline{F}(\vec{x}, y+1)$ pomoću $\vec{x}$, y i $z := \overline{F}(\vec{x}, y)$ — iako „kontrolnu varijablu" y zapravo i ne trebamo jer je uvijek možemo dobiti kao $\mathsf{lh}(z)$. Kao što smo, primjerice, operator $\sum$ mogli dobiti iteriranjem operacije $+$ (2.67) na početnoj vrijednosti 0 (2.66), tako operator $\overline{\cdots}$ možemo dobiti iteriranjem operacije $*$ na početnoj vrijednosti $\langle\rangle$. Dakle, vrijedi

$$\overline{F}(\vec{x}, y+1) = \overline{F}(\vec{x}, y) * \langle F(\vec{x}, y)\rangle = \overline{F}(\vec{x}, y) * \big\langle G\big(\vec{x}, \overline{F}(\vec{x}, y)\big)\big\rangle, \tag{3.29}$$

odnosno

$$H(\vec{x}, y, z) := z * \mathsf{Code}^1\big(G(\vec{x}, z)\big). \tag{3.30}$$

Sada je funkcija H (primitivno) rekurzivna prema lemi 3.18 i propoziciji 3.4, pa je i $\overline{F} = C_1^{k-1} \mathbin{\text{PR}} H$ (primitivno) rekurzivna jer je dobivena primitivnom rekurzijom iz (primitivno) rekurzivnih funkcija (za $k = 1$ to je degenerirana primitivna rekurzija $\overline{F} = 1 \mathbin{\text{PR}} H$, pa je $\overline{F}$ (primitivno) rekurzivna po propoziciji 2.22 odnosno po korolaru 2.36). Prema (3.27) je F dobivena kompozicijom iz G, $\overline{F}$ i koordinatnih projekcija, pa je i ona (primitivno) rekurzivna. □

3.2.2. Primjeri korištenja rekurzije s poviješću

Primjer 3.22: Poznata funkcija definirana rekurzijom s poviješću je Fibonaccijev niz:

$$\mathsf{Fib}(n) := n, \text{ za } n < 2; \tag{3.31}$$

$$\mathsf{Fib}(n) := \mathsf{Fib}(n-1) + \mathsf{Fib}(n-2), \text{ inače.} \tag{3.32}$$

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

$$\mathsf{rpart}(c, i) := \begin{cases} c[\mathsf{lh}(c) \dot{-} \mathsf{Sc}(i)], & i < \mathsf{lh}(c) \\ 0, & \text{inače} \end{cases}. \tag{3.33}$$

Sada točkovna definicija od G glasi:

$$G(p) := \begin{cases} \mathsf{lh}(p), & \mathsf{lh}(p) < 2 \\ \mathsf{rpart}(p, 0) + \mathsf{rpart}(p, 1), & \text{inače} \end{cases}. \tag{3.34}$$

Prema teoremu o grananju (primitivno rekurzivna verzija), funkcija rpart, pa onda i funkcija G, je primitivno rekurzivna, a tada je Fib primitivno rekurzivna po propoziciji 3.21, jer je dobivena rekurzijom s poviješću iz G. ◁

Iako smo rekurziju s poviješću uveli koristeći funkcije, promatranjem karakterističnih funkcija dobivamo analogni rezultat i za relacije. Ugrubo, ako pri utvrđivanju vrijedi li $R(\vec{x}, n)$ koristimo samo istinitosti $R(\vec{x}, m)$ za $m < n$, na neki način koji čuva (primitivnu) rekurzivnost, tada je i R (primitivno) rekurzivna. Evo jednog važnog primjera, koji će također poslužiti kao uvod u sljedeću točku, pokazujući da se mogu kodirati razni objekti — ne samo konačni nizovi prirodnih brojeva.

Primjer 3.23: Kodiramo formule logike sudova: propozicijsku varijablu P_i kao $\langle 0, i\rangle$, negaciju $\neg\varphi$ kao $\langle 1, u\rangle$ gdje je u kod od φ, a kondicional $(\varphi \to \psi)$ kao $\langle 2, u, v\rangle$ gdje su u i v kodovi od φ i ψ redom. Ostali veznici se mogu dobiti pomoću negacije i kondicionala na dobro poznat način: pogledajte [VukML09]. Recimo, kod varijable P_0 je $\langle 0, 0\rangle = 2^1 \cdot 3^1 = 6$, a kod formule $(P_0 \to P_0)$ je

$$\langle 2, 6, 6\rangle = 2^3 \cdot 3^7 \cdot 5^7 = 1\,366\,875\,000. \tag{3.35}$$

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 rekurzivno 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:

```
data PF = PropVar Integer | Not PF | Implies PF PF
```
(3.36)

i reprezentacija elementa takvog tipa u memoriji računala bila bi vrlo slična kodiranju koje smo mi napravili. Još jedan primjer, koji nije rekurzivno zadan 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 oblima", 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 rekurzivna 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 rekurzivna. Karakterističnu funkciju te slike, χ_{PF}, definirat ćemo rekurzijom s poviješću. Kao i prije, trebamo naći primitivno rekurzivnu funkciju G koja prima povijest $\overline{\chi_{\mathsf{PF}}}(n)$ i vraća je li n kod formule logike sudova. Kako vraća 0 ili 1 (bool), G možemo shvatiti kao karakterističnu funkciju: $\mathsf{G} = \chi_{\mathsf{R}}$, gdje je

$$\mathsf{R}(p) \iff \text{„}n := \mathsf{lh}(p) \text{ je kod neke formule logike sudova".} \tag{3.37}$$

Dakle, pretpostavimo da imamo n i razmislimo kako bismo odlučili je li kod neke formule — propozicijske varijable, ili negacije, ili kondicionala. Prvi disjunkt možemo napisati kao $\exists i\,(n = \langle 0, i\rangle)$, što nije dovoljno dobro jer je kvantifikacija neograničena. Kako je ograničiti? Ključno je da je **svaki element konačnog niza manji od koda tog niza**:

$$x_i < x_i + 1 < 2^{x_i+1} = p_0^{x_i+1} \le p_i^{x_i+1} \le (\cdots)\cdot p_i^{x_i+1}\cdot(\cdots) = \langle \ldots, x_i, \ldots\rangle. \tag{3.38}$$

Dakle, ako postoji takav i, on je sigurno manji od n, pa 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 $\exists u\big(n = \langle 1, u\rangle \wedge \mathsf{PF}(u)\big)$, i kao i prije možemo ograničiti kvantifikaciju na $(\exists u < n)$, ali što ćemo s rekurzivnim pozivom $\mathsf{PF}(u)$? Njegovu istinitost izvući ćemo iz povijesti p, u kojoj su zapisane sve vrijednosti karakteristične funkcije χ_{PF} na brojevima manjim od n. Dakle, drugi disjunkt je

$$(\exists u < n)\big(n = \langle 1, u\rangle \wedge p[u] = 1\big), \tag{3.39}$$

što je primitivno rekurzivno. Analogno bismo dobili i treći disjunkt (s dvije ograničene kvantifikacije), primitivno rekurzivan dvostrukom primjenom propozicije 2.55 (i brojnih drugih rezultata koji pokazuju da je kvantificirana relacija primitivno rekurzivna).

Tada je R primitivno rekurzivna kao disjunkcija tri primitivno rekurzivne relacije (propozicija 2.44), što znači da je njena karakteristična funkcija $\mathsf{G} = \chi_{\mathsf{R}}$ primitivno rekurzivna. I za kraj, prema propoziciji 3.21, χ_{PF} je tada primitivno rekurzivna, dakle PF^1 je primitivno rekurzivna relacija. ◁

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 standardan za konačne skupove: fiksiramo neki poredak. Konkretno, uzet ćemo kodiranje koje preslikava $\text{INC} \mapsto 0$, $\text{DEC} \mapsto 1$ i $\text{GO TO} \mapsto 2$, kao da smo deklarirali

```
enum ins_type { INC, DEC, GOTO };
```
(3.40)

u programskom jeziku C — time smo dobili injekciju s $\mathcal{I}\mathrm{ns}$ u $\mathbb{N}^*$, čije kodiranje već imamo. Sličnu stvar smo već napravili manje formalno u dokazu leme 1.7: 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* $\mathbb{N}\mathcal{I}\mathrm{ns}(I) =: \ulcorner I \urcorner$, jednadžbama:

$$\begin{aligned}
\ulcorner \text{INC}\ \mathcal{R}_j \urcorner &:= \langle 0, j\rangle &&= \mathsf{codeINC}(j) = 6 \cdot 3^j,\\
\ulcorner \text{DEC}\ \mathcal{R}_j, l \urcorner &:= \langle 1, j, l\rangle &&= \mathsf{codeDEC}(j, l) = 60 \cdot 3^j \cdot 5^l,\\
\ulcorner \text{GO TO}\ l \urcorner &:= \langle 2, l\rangle &&= \mathsf{codeGOTO}(l) = 24 \cdot 3^l,
\end{aligned}$$

za sve $j, l \in \mathbb{N}$. ◁

U desnom stupcu napisani su konstruktori kao aritmetički izrazi, iz kojih se vide neka njihova brojevnoteorijska svojstva, i kao primitivno rekurzivne funkcije od j i/ili l. Injektivnost kodiranja $\ulcorner \cdots \urcorner$ slijedi iz definicije 1.6 i propozicije 3.5. Da mu je slika $\mathsf{Ins} := \mathcal{I}_{\ulcorner \cdots \urcorner}$ primitivno rekurzivna, možemo vidjeti standardnom tehnikom „pokušaja rekodiranja iz komponenti", kao u dokazu korolara 3.16, ili u primjeru 3.23. Komponente j odnosno l te tip definiramo kroz dvije funkcije i tri relacije.

Lema 3.25: Skupovi InsINC, InsDEC i $\mathsf{InsGOTO}$, kodova instrukcija pojedinog tipa, kao i skup svih kodova instrukcija Ins, primitivno su rekurzivni.

Dokaz. Tvrdimo da je

$$\mathsf{InsINC}(i) \iff i = \mathsf{codeINC}(i[1]), \tag{3.41}$$

$$\mathsf{InsDEC}(i) \iff i = \mathsf{codeDEC}(i[1], i[2]), \tag{3.42}$$

$$\mathsf{InsGOTO}(i) \iff i = \mathsf{codeGOTO}(i[1]). \tag{3.43}$$

Dokazujemo samo (3.41); ostale ekvivalencije su sasvim analogne.

Za smjer ($\Rightarrow$), ako je i kod instrukcije tipa INC, recimo $i = \ulcorner \text{INC}\ \mathcal{R}_j \urcorner$, tada po definiciji 3.24 vrijedi $i = \mathsf{codeINC}(j) = \langle 0, j\rangle$, pa je $j = \langle 0, j\rangle[1] = i[1]$ po propoziciji 3.14(2).

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

Primitivna rekurzivnost skupa $\mathsf{Ins} = \mathsf{InsINC} \cup \mathsf{InsDEC} \cup \mathsf{InsGOTO}$ slijedi iz propozicije 2.44. □

Lema 3.26: Definiramo $\mathsf{regn}(t)$ kao adresu registra na koji djeluje instrukcija koda t, ako takva postoji i djeluje na neki registar, a 0 inače. Analogno definiramo $\mathsf{dest}(t)$ 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:

$$\mathsf{regn}(t) = \begin{cases} t[1], & \mathsf{InsINC}(t) \vee \mathsf{InsDEC}(t) \\ 0, & \text{inače} \end{cases}, \qquad \mathsf{dest}(t) = \begin{cases} t[2], & \mathsf{InsDEC}(t) \\ t[1], & \mathsf{InsGOTO}(t) \\ 0, & \text{inače} \end{cases}. \tag{3.44}$$

Ako bolje pogledamo, napisane jednakosti su „obrnuto” napisana definicija 3.24. □

Definicija 3.27: Za proizvoljni RAM-program $P := \left[\ t.\ I_t\ \right]_{t<n}$ definiramo *kod programa* P kao $\ulcorner P \lrcorner := \mathbb{N}\mathcal{P}\mathrm{rog}(P) := \langle \ulcorner I_0 \urcorner, \ulcorner I_1 \urcorner, \ldots, \ulcorner I_{n-1} \urcorner \rangle$. ◁

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 $\ulcorner P^* \urcorner := \langle 3, \ulcorner P \lrcorner \rangle$ 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, vidjet ćemo da možemo jednu jednostavnu transformaciju programa — specijalizaciju — napraviti kao primitivno rekurzivnu funkciju, a sve ostale transformacije pomoću nje.

Primjer 3.28: U dokazu teorema 1.16 napisali smo RAM-programe P_n (1.9) koji računaju konstantne funkcije. Za svaki n možemo izračunati kod tog programa, tako da je preslikavanje $n \mapsto \ulcorner P_n \lrcorner$ primitivno rekurzivno. Doista, svaka instrukcija u tim programima je INC $\mathcal{R}_0$, s kodom $\mathsf{codeINC}(0) = 6$, pa je

$$\ulcorner P_n \lrcorner = \langle 6, 6, \ldots, 6 \rangle \ [n \text{ šestica}] = \langle C_6(0), C_6(1), \ldots, C_6(n-1) \rangle = \overline{C_6}(n). \tag{3.45}$$

$\overline{C_6^1}$ je primitivno rekurzivna po propoziciji 2.21 i lemi 3.15. Isto možemo dobiti dinamizacijom ove šestice kao u primjeru 2.51, ili čak potenciranjem primorijela: $\ulcorner P_n \lrcorner = \overline{I_1^2}(6, n) = \left(\overline{Z}(n)\right)^7$ — vidite li zašto? ◁

Primjer 3.29: U primjeru 1.25 naveden je RAM-program $Q^\flat$ (1.20), čiji je kod

$$\begin{aligned}
e_Q := \ulcorner Q^\flat \urcorner &= \langle \ulcorner \textsc{dec}\ \mathcal{R}_1, 2 \urcorner, \ulcorner \textsc{go to}\ 0 \urcorner, \ulcorner \textsc{dec}\ \mathcal{R}_2, 2 \urcorner, \ulcorner \textsc{dec}\ \mathcal{R}_1, 6 \urcorner, \ulcorner \textsc{inc}\ \mathcal{R}_0 \urcorner, \ldots, \ulcorner \textsc{go to}\ 9 \urcorner \rangle = \\
&= \langle \langle 1,1,2 \rangle, \langle 2,0 \rangle, \langle 1,2,2 \rangle, \langle 1,1,6 \rangle, \langle 0,0 \rangle, \langle 2,3 \rangle, \langle 1,2,9 \rangle, \langle 0,0 \rangle, \langle 2,6 \rangle, \langle 1,3,12 \rangle, \langle 0,0 \rangle, \langle 2,9 \rangle \rangle = \\
&= \langle 4\,500, 24, 13\,500, 2\,812\,500, 6, 648, 1\,054\,687\,500, 6, 17\,496, 395\,507\,812\,500, 6, 472\,392 \rangle = \\
&2^{4501} \cdot 3^{25} \cdot 5^{13501} \cdot 7^{2812501} \cdot 11^{7} \cdot 13^{649} \cdot 17^{1054687501} \cdot 19^{7} \cdot 23^{17497} \cdot 29^{395507812501} \cdot 31^{7} \cdot 37^{472393}. \quad \triangleleft
\end{aligned}$$

Injektivnost preslikavanja $\ulcorner \cdots \urcorner$ slijedi iz injektivnosti od $\ulcorner \cdots \urcorner$ i $\langle \cdots \rangle$: 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 instrukcijā injektivno).

Lema 3.30: Slika kodiranja RAM-programa, $\mathsf{Prog}^1 := \mathcal{I}_{\mathbb{N}\mathcal{P}\mathrm{rog}} = \{\ulcorner P \urcorner \mid P \in \mathcal{P}\mathrm{rog}\}$, primitivno je rekurzivna.

Dokaz. Za RAM-program treba specificirati da se radi o konačnom nizu instrukcija, čija su odredišta (ako postoje) manja ili jednaka duljini programa:

$$\mathsf{Prog}(e) \iff \mathsf{Seq}(e) \wedge \big(\forall i < \mathsf{lh}(e)\big)\big(\mathsf{Ins}(e[i]) \wedge \mathsf{dest}(e[i]) \leq \mathsf{lh}(e)\big). \tag{3.46}$$

Ovdje koristimo činjenicu da je $\mathsf{dest}(t) = 0$ ako instrukcija koda t nema odredište, a uvijek je $0 \leq \mathsf{lh}(e)$. Marljiva evaluacija od $\wedge$ (odnosno mul^2) znači da će se $\mathsf{lh}(e)$ uvijek izračunati, čak i kad e nije kod konačnog niza, ali rezultat čitavog izraza će tada 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. □

3.3.1. Kodiranje stanja registara i RAM-konfiguracija

Konfiguraciju RAM-stroja smo definirali kao jednu funkciju, ali zapravo je jasno da trebamo pratiti dvije odvojene stvari: stanje registara i vrijednost programskog brojača. Vrijednost programskog brojača već jest prirodan broj pa je ne kodiramo, odnosno kodiramo je 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 1.9 kaže da promatramo samo konfiguracije 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, a u svakom koraku izračunavanja se nosač može povećati najviše za jedan element — izvršavanjem instrukcije tipa $\textsc{inc}$ na registru čiji je sadržaj bio 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 proizvoljni niz prirodnih brojeva s konačnim nosačem $(r_0, r_1, r_2, \ldots, 0, 0, 0, \ldots)$ definiramo kod kao

$$\langle r_0, r_1, r_2, \ldots, 0, 0, 0, \ldots \rangle := \prod_{i \in \mathbb{N}} p_i^{r_i}. \tag{3.47}$$

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. Alternativno bismo mogli naći širinu programa 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*) moglo bi promijeniti kod stanja registara za isto izračunavanje (isti niz konfiguracija) pa kodiranje ne bi bilo funkcija.

Treba nam jedan važan 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 takva da za svaki neprazni konačni niz $\vec{x} \in \mathbb{N}^+$ vrijedi $\mathsf{start}(\langle\vec{x}\rangle) = \langle 0, \vec{x}, 0, 0, 0, \ldots\rangle$.

Dokaz. Samo treba napisati rastav na prim-faktore:

$$\mathsf{start}(x) := \prod_{i=0}^{\mathsf{lh}(x)-1} p_{i+1}^{x[i]} = \prod_{i<\mathsf{lh}(x)} \mathsf{pow}\big(\mathsf{prime}\big(\mathsf{Sc}(i)\big), \mathsf{part}(x,i)\big). \tag{3.48}$$

Sada primitivna rekurzivnost slijedi iz propozicija 3.12 i 3.14, leme 2.53 i primjera 2.13. Prim-brojevi su pomaknuti jer adrese ulaznih registara počinju od 1. □

Funkcija start nije injekcija: recimo, $\mathsf{start}(\langle 2,1\rangle) = \mathsf{start}(\langle 2,1,0,0\rangle) = \langle 0,2,1,0,0,\ldots\rangle$, odnosno $\mathsf{start}(72) = \mathsf{start}(2520) = 45$. Ali preslikavanje $\langle\cdots\rangle$ jest injekcija (kao što kodiranje i treba biti), po osnovnom teoremu aritmetike. Također po osnovnom teoremu aritmetike, slika mu je $\mathcal{I}_{\langle\ldots\rangle} = \mathbb{N}_+$, primitivno rekurzivna po primjeru 2.28.

Što se komponenata tiče, lh nema smisla, a za indeksiranje služi funkcija ex iz leme 3.13. Doista, $\mathsf{ex}(\langle r_0, r_1, \ldots, 0, 0, \ldots\rangle, i) = r_i$, jer je to upravo eksponent od p_i u tom kodu. Koristit ćemo je za očitavanje rezultata (sadržaja registra $\mathcal{R}_0$) iz završne konfiguracije.

$$\mathsf{result}(c) := \mathsf{ex}(c, 0) \tag{3.49}$$

Za sāmo 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

$$\langle r_0, r_1, \ldots, r_{j-1}, r_j + 1, r_{j+1}, r_{j+2}, \ldots, 0, 0, \ldots\rangle = \langle r_0, r_1, \ldots, 0, 0, \ldots\rangle \cdot p_j, \tag{3.50}$$

i odmah iz toga (za $r_j > 0$)

$$\langle r_0, r_1, \ldots, r_{j-1}, r_j - 1, r_{j+1}, r_{j+2}, \ldots, 0, 0, \ldots\rangle = \langle r_0, r_1, \ldots, 0, 0, \ldots\rangle \mathbin{/\!/} p_j. \tag{3.51}$$

Sada možemo kodirati dokaz leme 1.10: za preslikavanje $\mathit{nextconf}$ koje konfiguraciju RAM-stroja preslikava u „sljedeću” konfiguraciju (u koju ova prelazi), konstruirat ćemo $\mathbb{N}\mathit{nextconf}$ kao izračunljivu funkciju. Ona ima dva izlaza, pa ćemo je reprezentirati kroz dvije primitivno rekurzivne funkcije — koje će primati 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 $c[\mathcal{R}] := \langle c(\mathcal{R}_0), c(\mathcal{R}_1), \ldots \rangle$. Tada postoje primitivno rekurzivne funkcije $\mathsf{NextReg}^2$ i $\mathsf{NextCount}^3$ takve da za svaku RAM-instrukciju I, i za sve RAM-konfiguracije c i d takve da $c \rightsquigarrow d$ po instrukciji I, vrijedi

$$\mathsf{NextReg}\big(\ulcorner I \urcorner, c[\mathcal{R}]\big) = d[\mathcal{R}], \text{ i} \tag{3.52}$$

$$\mathsf{NextCount}\big(\ulcorner I \urcorner, c[\mathcal{R}], c(\textsc{pc})\big) = d(\textsc{pc}). \tag{3.53}$$

Dokaz. Označimo argumente s $i := \ulcorner I \urcorner \in \mathsf{Ins}$, $r := c[\mathcal{R}] \in \mathbb{N}_+$ te $p := c(\textsc{pc}) \in \mathbb{N}$.

$\mathsf{NextReg}$ 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. $\mathsf{NextCount}$ treba inkrementirati p ako je I promijenila neki registar, ili ga postaviti na njeno odredište ako nije.

$$\mathsf{NextReg}(i, r) := \begin{cases} r \cdot pj, & \mathsf{InsINC}(i) \iff: \mathit{Up} \\ r \mathbin{/\!\!/} pj, & \mathsf{InsDEC}(i) \wedge pj \mid r \iff: \mathit{Down} \\ r, & \text{inače} \end{cases} \tag{3.54}$$

$$\mathsf{NextCount}(i, r, p) := \begin{cases} \mathsf{Sc}(p), & \mathit{Up} \vee \mathit{Down} \\ \mathsf{dest}(i), & \text{inače} \end{cases} \tag{3.55}$$

$$\text{uz pokratu } pj := \mathsf{prime}\big(\mathsf{regn}(i)\big) \tag{3.56}$$

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 funkcijā $\mathsf{NextReg}$ i $\mathsf{NextCount}$, pa su one primitivno rekurzivne po teoremu 2.46.

Za parcijalnu specifikaciju, neka su I i c (odnosno i, r i p) kao na početku dokaza. Tvrdimo: $\mathsf{NextReg}(i, r)$ i $\mathsf{NextCount}(i, r, p)$ kodiraju konfiguraciju u koju c prelazi po I.

Ako je I tipa $\textsc{inc}$, recimo $I = (\textsc{inc}\ \mathcal{R}_j)$, tada vrijedi Up i $pj = \mathsf{prime}(j) = p_j$, pa je $r' = r \cdot p_j$, što prema (3.50) kodira upravo stanje registara nakon izvršavanja I.

Ako je I tipa $\textsc{go to}$, recimo $I = (\textsc{go to}\ l)$, tada ne vrijedi ni Up ni Down pa je $r' = r$, a $p' = \mathsf{dest}(i) = \mathsf{dest}(\ulcorner \textsc{go to}\ l \urcorner) = l$, kao što i treba biti.

Ako je pak I tipa $\textsc{dec}$, recimo $I = (\textsc{dec}\ \mathcal{R}_j, l)$, tada je opet $pj = p_j$ te Up ne vrijedi, a $\mathit{Down} \Leftrightarrow p_j \mid r = c[\mathcal{R}] \Leftrightarrow c(\mathcal{R}_j) > 0$. Ako to vrijedi, $p' = p + 1$ i $r' = r \mathbin{/\!\!/} p_j$, a inače, $p' = \mathsf{dest}(\ulcorner \textsc{dec}\ \mathcal{R}_j, l \urcorner) = l$ i $r' = r$, što i treba biti po semantici instrukcije tipa $\textsc{dec}$. □

3.3.2. Kodiranje RAM-izračunavanja po koracima

Sada raspolažemo 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 $P \in \mathcal{P}\mathrm{rog}$ te vraća niz $(c_n)_{n \in \mathbb{N}}$ koji predstavlja P-izračunavanje s $\vec{x}$.

Izrazimo taj zadatak preko brojevnih funkcija. Umjesto $\vec{x}$ i P prenijet ćemo njihove kodove $x := \langle \vec{x} \rangle \in \mathsf{Seq}' := \mathsf{Seq} \setminus \{\langle\rangle\}$ i $e := \ulcorner P \urcorner \in \mathsf{Prog}$. Tada ne treba prenositi k jer ga uvijek možemo odrediti kao $\mathsf{lh}(x)$. No kako vratiti niz? Nizova konfiguracijā ima neprebrojivo mnogo, pa moramo malo modificirati sučelje. Ukratko, nizove vraćamo „točkovno” tako da zapravo vraćamo njihove članove, dodavši indeks u nizu kao još jedan ulazni podatak — a algoritam s

dva izlazna podatka (kōd stanja registara i vrijednost programskog brojača) po napomeni 1.1 shvaćamo kao dva algoritma s istim ulaznim podacima.

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, $\mathsf{Reg}(\langle\vec{x}\rangle, \ulcorner P \urcorner, n)$ je kod stanja registara, a $\mathsf{Count}(\langle\vec{x}\rangle, \ulcorner P \urcorner, n)$ je vrijednost programskog brojača, nakon n koraka P-izračunavanja s $\vec{x}$.

> Preciznije, ako je P-izračunavanje s $\vec{x}$ niz RAM-konfiguracijā $(c_n)_{n\in\mathbb{N}}$, tada je $\mathsf{Reg}(\langle\vec{x}\rangle, \ulcorner P \urcorner, n) = c_n[\mathcal{R}]$ te $\mathsf{Count}(\langle\vec{x}\rangle, \ulcorner P \urcorner, n) = c_n(\mathrm{PC})$.

Dokaz. To što smo razdvojili reprezentaciju izračunavanja na dvije funkcije ne znači da ih možemo računati odvojenim algoritmima. Prema lemi 1.10 sljedeća konfiguracija ovisi samo o neposredno prethodnoj (izračunavanje je *memoryless* — kao Markovljev lanac, samo deterministično), 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 traženih funkcija u $n = 0$. 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:

$$\mathsf{Reg}(\langle\vec{x}\rangle, e, 0) = \langle 0, x_1, x_2, \ldots, x_k, 0, 0, \ldots\rangle = \mathsf{start}(\langle\vec{x}\rangle), \tag{3.57}$$

pa za točkovnu definiciju inicijalizacijske funkcije za Reg (gdje je $x \in \mathbb{N}$ proizvoljan) uzmimo

$$\mathsf{Reg}(x, e, 0) := \mathsf{G}_1(x, e) := \mathsf{start}(x). \tag{3.58}$$

Inicijalizacija Count je $\mathsf{G}_2 := \mathsf{C}_0^2$, 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 $\mathsf{NextReg}$ i $\mathsf{NextCount}$, s tim da sad moramo voditi računa i o završnim konfiguracijama, ostavljajući ih fiksnima. $\mathsf{NextReg}$ to već čini, jer ako t nije kod instrukcije, (3.54) kaže da je $\mathsf{NextReg}(t, r) = r$ — no za H_2 ćemo morati granati.

$$\mathsf{H}_1(x, e, n, r, p) := \mathsf{NextReg}(e[p], r), \tag{3.59}$$

$$\mathsf{H}_2(x, e, n, r, p) := \begin{cases} \mathsf{NextCount}(e[p], r, p), & p < \mathsf{lh}(e) \\ p, & \text{inače} \end{cases}. \tag{3.60}$$

Funkcije H_1 i H_2 su primitivno rekurzivne po teoremu o grananju 2.46, propoziciji 3.14, lemi 3.32 i primjeru 2.29. Sada je jasno da su Reg i Count, definirane s

$$\mathsf{Reg}(x, e, 0) := \mathsf{G}_1(x, e) = \mathsf{start}(x), \tag{3.61}$$

$$\mathsf{Count}(x, e, 0) := \mathsf{G}_2(x, e) = 0, \tag{3.62}$$

$$\mathsf{Reg}(x, e, n+1) := \mathsf{H}_1\big(x, e, n, \mathsf{Reg}(x, e, n), \mathsf{Count}(x, e, n)\big), \tag{3.63}$$

$$\mathsf{Count}(x, e, n+1) := \mathsf{H}_2\big(x, e, n, \mathsf{Reg}(x, e, n), \mathsf{Count}(x, e, n)\big), \tag{3.64}$$

dobivene simultanom primitivnom rekurzijom iz primitivno rekuzivnih funkcija G_1, G_2, H_1 i H_2 pa 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 broju koraka n. Za $n = 0$, $\mathsf{Count}(x, e, n) = 0$, što je po definiciji vrijednost programskog brojača nakon 0 koraka izračunavanja. Također, iz leme 3.31 slijedi da je $\mathsf{Reg}(x, e, 0)$ 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 $\mathsf{Count}(x, e, n) = c_n(\textsc{pc}) = \mathsf{lh}(e)$ pa ne vrijedi uvjet grananja u definiciji H_2 pozvanoj iz (3.64). Također je prema propoziciji 3.14(3) $e[\mathsf{lh}(e)] = 0 \notin \mathsf{Ins}$, što znači da funkcija H_1 vraća nepromijenjen kod stanja registara (ne vrijedi ni Up ni Down u (3.54)). Iz toga je $\mathsf{Reg}(x, e, n+1) = \mathsf{Reg}(x, e, n)$ i $\mathsf{Count}(x, e, n+1) = \mathsf{Count}(x, e, n)$, što i treba biti jer je $c_{n+1} = c_n$.

Ako pak c_n nije završna, tada je $p := c_n(\textsc{pc}) < \mathsf{lh}(e)$ te je $i := e[p]$ kod instrukcije koja se izvršava u $(n+1)$. koraku. Tada je po pretpostavci indukcije i lemi 3.32,

$$\begin{aligned}\mathsf{Reg}(x, e, n+1) &= \mathsf{H}_1\big(x, e, n, \mathsf{Reg}(x, e, n), p\big) = \\ &= \mathsf{NextReg}\big(e[p], \mathsf{Reg}(x, e, n)\big) = \mathsf{NextReg}\big(i, c_n[\mathcal{R}]\big) = c_{n+1}[\mathcal{R}], \end{aligned} \tag{3.65}$$

i analogno $\mathsf{Count}(x, e, n+1) = c_{n+1}(\textsc{pc})$, čime je proveden korak indukcije. □

3.3.3. Prepoznavanje završne konfiguracije i čitanje rezultata

Pomoću funkcije Count možemo detektirati završnu konfiguraciju.

$$\mathsf{Final}'(x, e, n) :\Longleftrightarrow \mathsf{Count}(x, e, n) = \mathsf{lh}(e) \tag{3.66}$$

Zbog $\chi^3_{\mathsf{Final}'} = \chi_= \circ (\mathsf{Count}, \mathsf{lh} \circ \mathsf{I}^3_2)$, to je primitivno rekurzivna relacija, a za sve $P \in \mathcal{P}rog$, $\vec{x} \in \mathbb{N}^+$ i $n \in \mathbb{N}$, $\mathsf{Final}'(\langle\vec{x}\rangle, \ulcorner P \urcorner, n)$ 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 $\mathsf{Final}'(x, e, n)$, iako e uopće nije kod RAM-programa, niti je x kod nepraznog konačnog niza.

Primjer 3.34: Recimo, za $x = 100$ i $e = 10^{217}$, prvo bismo odredili $\mathsf{start}(100)$. Broj 100 ima dva prim-djelitelja (2 i 5), pa je $\mathsf{lh}(100) = 2$. Iz toga $\mathsf{start}(100) = 3^{100[0]} \cdot 5^{100[1]} = \langle 0, 1, 0, 0, \ldots\rangle = 3 = \mathsf{Reg}(100, 10^{217}, 0)$. Dakle, $\mathcal{R}_1$ kreće od 1, a svi ostali registri od nule. $\mathsf{Count}(100, 10^{217}, 0) = 0$ jer $\textsc{pc}$ uvijek kreće od nule.

Sada odredimo „trenutnu instrukciju": $e[0] = 216 = 2^3 \cdot 3^3 = \langle 2, 2\rangle = \ulcorner \textsc{go to}\ 2 \urcorner \in \mathsf{InsGOTO}$ pa je $\mathsf{Count}(100, 10^{217}, 1) = \mathsf{dest}(216) = 2 = \mathsf{lh}(e)$. Dakle, $\mathsf{Final}'(100, 10^{217}, 1)$ vrijedi — no nema smisla reći da to reprezentira zaustavljanje nekog izračunavanja, jer ne postoje $P \in \mathcal{P}rog$ i $\vec{x} \in \mathbb{N}^+$ takvi da bi to bilo P-izračunavanje s $\vec{x}$. ◁

Primjer 3.34 pokazuje da trebamo biti oprezni s parcijalnim specifikacijama. Dok su za primitivno rekurzivne funkcije one ponekad „nužno zlo" ili barem „manje zlo", za primitivno rekurzivne *relacije* je mnogo prirodnije isključiti sve nespecificirane točke (efektivno, promatrati presjek s područjem specifikacije).

$$\mathsf{Final}(x, e, n) :\Longleftrightarrow \mathsf{Seq}'(x) \wedge \mathsf{Prog}(e) \wedge \mathsf{Final}'(x, e, n) \tag{3.67}$$

Lema 3.35: Relacija Final^3 je primitivno rekurzivna te je $(x, e, n) \in \mathsf{Final}$ ako i samo ako je x kod nekog nepraznog konačnog niza $\vec{x} \in \mathbb{N}^+$, e je kod nekog RAM-programa P, a P-izračunavanje s $\vec{x}$ stane nakon najviše n koraka.

Dokaz. Za primitivnu rekurzivnost: jednočlana relacija $\{\langle\rangle\} = \{1\}$ je primitivno rekurzivna po lemi 2.47. Tada je $\mathsf{Seq}' := \mathsf{Seq} \setminus \{1\}$ primitivno rekurzivna po propoziciji 2.41, a onda i Final po propoziciji 2.44.

Za specifikaciju, smjer slijeva nadesno, raspetljavanjem definicije od $\mathsf{Final}(x, e, n)$ vidimo da mora vrijediti:

- $\mathsf{Seq}(x)$, dakle $x = \langle\vec{x}\rangle$ za neki $\vec{x} \in \mathbb{N}^*$, jer je $\mathsf{Seq} = \mathcal{I}_{\langle\dots\rangle}$;
- $x \neq 1 = \langle\rangle$, dakle $\vec{x}$ je neprazan, jer je $\langle\cdots\rangle$ injekcija;
- $\mathsf{Prog}(e)$, dakle $e = \ulcorner P \urcorner$ za neki RAM-program P, jer je $\mathsf{Prog} = \mathcal{I}_{\ulcorner\dots\urcorner}$;
- $\mathsf{Count}(x, e, n) = \mathsf{lh}(e)$, što prema lemi 3.33 (koju možemo primijeniti zahvaljujući prethodnim trima stavkama) znači da je n-ta konfiguracija u P-izračunavanju s $\vec{x}$ završna — iz čega slijedi da ono stane; štoviše, da je prvi indeks završne konfiguracije u izračunavanju manji ili jednak n.

Zaključivanjem u suprotnom smjeru lako dobijemo i drugi smjer tvrdnje. □

To je u redu ako već imamo (kandidat za) n — ali možemo li *izračunati* n? Svakako, minimizacijom: $\mathsf{step} := \mu\,\mathsf{Final}$ je parcijalno rekurzivna funkcija takva da je $\mathsf{step}(\langle\vec{x}\rangle, \ulcorner P \urcorner) \simeq \mu n\, \mathsf{Final}(\langle\vec{x}\rangle, \ulcorner P \urcorner, n)$ upravo broj koraka P-izračunavanja s $\vec{x}$ do dolaska u završnu konfiguraciju. To je definirano samo za izračunavanja koja stanu: $\mathrm{HALT} := \mathcal{D}_{\mathsf{step}} = \exists_* \mathsf{Final}$ je dvomjesna relacija takva da $\mathrm{HALT}(\langle\vec{x}\rangle, \ulcorner P \urcorner)$ vrijedi ako i samo ako P-izračunavanje s $\vec{x}$ stane.

Napomena 3.36: Zahvaljujući oprezu sa specifikacijom prije, sad možemo biti i precizniji: kad god e nije kod nekog programa, ili x nije kod nepraznog konačnog niza, ne vrijedi $\mathsf{Final}(x, e, n)$ ni za koji n. Dakle, tada $(x, e) \notin \mathrm{HALT}$, odnosno izraz $\mathsf{step}(x, e)$ nema vrijednost. ◁

Uočite 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 $(x, e) \in \mathrm{HALT}$, to ćemo doznati čim nađemo broj koraka n — ali ako $(x, e) \notin \mathrm{HALT}$, to nećemo nikada doznati tim pristupom. Pokazat ćemo, štoviše, da **nijedan pristup ne radi za sve parove** (x, e), iako u nekim slučajevima (poput $e \notin \mathsf{Prog}$) možemo utvrditi $\neg\mathrm{HALT}(x, e)$. 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 $\mathrm{HALT}(x, e)$, izraz $\mathsf{Reg}\big(x, e, \mathsf{step}(x, e)\big)$ ima vrijednost koja kodira stanje registara u završnoj konfiguraciji izračunavanja. Za $\neg\mathrm{HALT}(x, e)$, što uključuje i $\neg\mathsf{Prog}(e)$ i $\neg\mathsf{Seq}'(x)$, taj izraz nema vrijednost. Još treba dokomponirati primitivno rekurzivnu funkciju result iz (3.49), dobivši

$$\mathsf{univ}(x, e) :\simeq \mathsf{result}\big(\mathsf{Reg}\big(x, e, \mathsf{step}(x, e)\big)\big), \tag{3.68}$$

univerzalnu funkciju koja preslikava RAM-program i ulaz za njega u rezultat izračunavanja, i to 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 $\mathsf{univ}(x, e)$ rezultat tog izračunavanja.

2. U svim ostalim slučajevima, $(x, e) \notin \mathcal{D}_{\mathsf{univ}}$ (izraz $\mathsf{univ}(x, e)$ nema vrijednost).

Dokaz. Parcijalna rekurzivnost slijedi iz (3.68), iz primitivne rekurzivnosti korištenih funkcija $\mathsf{result} = \mathsf{ex} \circ (\mathsf{I}_1^1, \mathsf{Z})$ i Reg te parcijalne rekurzivnosti funkcije step.

Za tvrdnju 1, pretpostavke kažu da postoji završna konfiguracija u P-izračunavanju s $\vec{x}$. Prema lemi 3.35, to znači $\exists n\, \mathsf{Final}(x, e, n)$, dakle $(x, e) \in \exists_* \mathsf{Final} = \mathcal{D}_{\mathsf{step}}$, pa postoji $\mathsf{step}(x, e) =: n_0$, i $\mathsf{Reg}(x, e, n_0)$ je kod stanja registara u završnoj konfiguraciji c_{n_0} tog izračunavanja.

$$z := \mathsf{Reg}(x, e, n_0) = c_{n_0}[\mathcal{R}] = \langle c_{n_0}(\mathcal{R}_0), c_{n_0}(\mathcal{R}_1), \ldots \rangle \tag{3.69}$$

Sada je $\mathsf{univ}(x, e) = \mathsf{result}(z) = \mathsf{ex}(\langle c_{n_0}(\mathcal{R}_0), \ldots \rangle, 0) = c_{n_0}(\mathcal{R}_0)$, 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 $\mathsf{Final}(x, e, n)$, iz čega $(x, e) \notin \mathcal{D}_{\mathsf{step}}$. Iz pretpostavke o marljivoj evaluaciji (2.4) slijedi da (x, e) ne može biti ni u domeni od univ. □

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 2.38) i svaku parcijalno rekurzivnu funkciju. Na neki način, univ predstavlja funkcijski *interpreter* za RAM-stroj, nasuprot imperativnom *kompajleru* za simboličke definicije izgrađenom u poglavlju 2. No iz tehničkih i povijesnih razloga, univerzalna funkcija obično se uvodi malo drugačije.

Htjeli bismo napisati što jednostavniju „simboličku definiciju" univerzalne funkcije. Definicija (3.68) ima nezgodno svojstvo da dvaput koristi x i e, odnosno napisana je kao kompozicija dvije funkcije, svaka od kojih ovisi o ulaznim podacima. No vanjska funkcija ne mora primati x i e, ako joj unutarnja funkcija pošalje završnu konfiguraciju, ili nešto iz čega se ona može odrediti. Razlog zašto ih trenutno prima je u tome što joj unutarnja funkcija pošalje samo broj koraka, sam po sebi nedovoljan za određivanje završne konfiguracije. No koristeći kodiranje $\mathbb{N}^*$, poslana vrijednost može biti proizvoljno komplicirana.

Drugo, od osnovnih operatora ($\circ$, PR 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. Tehnikama svođenja iz poglavlja 5 može se pokazati da minimizacija ne može biti u korijenu (ne postoji rekurzivna relacija R^3 takva da je $\mathsf{univ} = \mu \mathsf{R}$) — dakle moramo nakon μ primijeniti još neki operator. To ne može biti PR jer bismo time opet dobili totalnu funkciju, dakle mora biti kompozicija. Najjednostavnija „normalna forma" koja zadovoljava te uvjete je $\mathsf{univ}^2 = \mathsf{U}^1 \circ \mu \check{\mathsf{T}}^3$ za primitivno rekurzivne U i $\check{\mathsf{T}}$ (U je funkcija, $\check{\mathsf{T}}$ je relacija).

Funkciji U trebamo poslati 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 $\mathsf{U} \circ \mu\check{\mathsf{T}}$ bude definirano u tom slučaju.

Dakle, promotrimo izračunavanje $(c_n)_{n\in\mathbb{N}}$ koje stane. Vidjeli smo da ono mora biti oblika $(c_0, c_1, \ldots, c_{n_0}, c_{n_0}, c_{n_0}, \ldots)$, 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 $(c_0, c_1, \ldots, c_{n_0})$ duljine $n_0 + 1$, jer se iz njega beskonačnim ponavljanjem posljednjeg elementa može dobiti i čitavo izračunavanje. Za kodirati pojedinu c_i, trebali bismo uračunati i $\mathsf{Reg}(x, e, i)$ i $\mathsf{Count}(x, e, i)$, no s obzirom na to da nam je u završnoj konfiguraciji po definiciji poznata ova druga vrijednost $(c_{n_0}(\mathrm{PC}) = \mathsf{lh}(e))$ i zapravo nam treba sadržaj registra $\mathcal{R}_0$, pamtit ćemo samo vrijednosti funkcije Reg. Drugim riječima, treba nam povijest $\overline{\mathsf{Reg}}(x, e, n_0 + 1)$, što je izračunljivo jednom kad imamo $n_0 := \mathsf{step}(x, e)$.

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. ◁

Propozicija 1.12, restringirana samo na izračunavanja koja stanu (jer jedino takva znamo kodirati), može se iskazati kao: tromjesna relacija

$$\mathrm{Trace}\big(\vec{x}, P, (c_n)_{n\in\mathbb{N}}\big) :\Longleftrightarrow \text{„}(c_n)_{n\in\mathbb{N}} \text{ je P-izračunavanje s } \vec{x}\text{, koje stane”} \tag{3.70}$$

ima funkcijsko svojstvo. Tada će i $\check{\mathsf{T}}^3 := \mathbb{N}\mathrm{Trace}$ imati funkcijsko svojstvo. Cilj nam je dokazati da je Trace izračunljiva, odnosno da je $\check{\mathsf{T}}$ primitivno rekurzivna.

U skladu s napomenom iz primjera 3.17, Trace ćemo shvatiti kao $(k+2)$-mjesnu relaciju, tako da svaki x_i shvatimo kao zasebni argument. To vodi na promatranje familije relacija $\mathsf{T}_k^{k+2}, k \in \mathbb{N}_+$ — a jednom kad dobijemo primitivnu rekurzivnost od $\check{\mathsf{T}}$, dobit ćemo i primitivnu rekurzivnost svih T_k, jer je

$$\mathsf{T}_k(\vec{x}, e, y) \Longleftrightarrow \check{\mathsf{T}}(\langle\vec{x}\rangle, e, y), \tag{3.71}$$

dakle T_k je dobivena iz $\check{\mathsf{T}}$ kompozicijom s Code^k i koordinatnim projekcijama.

Primjer 3.39: U primjeru 1.22 je naveden primjer makro-programa Q i Q-izračunavanja s $(2, 4)$, koje stane. Kako $Q^\flat$-izračunavanje s $(2, 4)$ prolazi kroz ista stanja registara, možemo iz (1.17) izračunati kod tog izračunavanja (iz definicije 1.20 trebamo zanemariti prijelaze tipa 4, jer oni ne odgovaraju nikakvim prijelazima RAM-stroja, kao i one tipa 1, jer smo tada već stigli do završne konfiguracije):

$$\begin{aligned} c_0 := \langle 5625, 1875, 1875, 625, 625, 625, 125, 125, 25, 50, 50, 10, 20, 20, 4, 8, 8, 8, 8\rangle &= \\ = 2^{5626} \cdot 3^{1876} \cdot 5^{1876} \cdot 7^{626} \cdot 11^{626} \cdot 13^{626} \cdot 17^{126} \cdots 61^9 \cdot 67^9. \end{aligned} \tag{3.72}$$

Recimo, $c_0[6] = \mathsf{part}(c_0, 6) = \mathsf{pd}\big(\mathsf{ex}(c_0, 6)\big) = \mathsf{pd}(126) = 125 = 5^3 = \langle 0, 0, 3, 0, \ldots\rangle$, jer nakon 6 koraka $Q^\flat$-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 $\mathsf{Reg}(\langle 2, 4\rangle, \ulcorner Q^\flat \urcorner, 6) = \langle 0, 0, 3, 0, \ldots\rangle$. Ako izračunamo $\langle 2, 4\rangle = 2^3 \cdot 3^5 = 1944$ i upotrijebimo $e_Q := \ulcorner Q^\flat \urcorner$ iz primjera 3.29, možemo napisati i $\mathsf{Reg}(1944, e_Q, 6) = 5^3$. Vidimo da je c_0 povijest funkcije Reg, konkretno $\check{\mathsf{T}}(1944, e_Q, c_0)$ znači $c_0 = \overline{\mathsf{Reg}}(1944, e_Q, 19)$, gdje je $\mathsf{pd}(19) = 18 = \mathsf{step}(1944, e_Q)$. ◁

Funkcija step jest izračunljiva, ali budući da nam treba *relacija* za minimizaciju, koristit ćemo njen graf $\mathsf{Step}^3 := \mathcal{G}_{\mathsf{step}}$ kao relaciju iz koje možemo dobiti njene vrijednosti.

Napomena 3.40: Važno: **ne možemo** napisati $n = \mathsf{step}(x, e)$ kao točkovnu definiciju relacije Step — to bi simbolički glasilo $\chi_{\mathsf{Step}} = \chi_= \circ \big(\mathsf{I}_3^3, \mathsf{step} \circ (\mathsf{I}_1^3, \mathsf{I}_2^3)\big)$, što nije istina jer te dvije funkcije imaju različite domene: lijeva je totalna, a desna je definirana samo na $\mathrm{HALT} \times \mathbb{N}$. 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. ◁

Ipak, za totalne funkcije takav rezultat vrijedi, i možemo ga već sada 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

$$\mathcal{G}_f(\vec{x}, y) :\Longleftrightarrow \vec{x} \in \mathcal{D}_f \wedge y \simeq f(\vec{x}). \quad \triangleleft \tag{3.73}$$

U definiciji općenito ne smijemo napisati $y = f(\vec{x})$ umjesto $y \simeq f(\vec{x})$ jer bi to impliciralo totalnost od f (pogledajte napomenu 3.40), ali ako je f već totalna, onda možemo.

Napomena 3.42: Iz elementarne matematike znamo da je relacija R graf neke funkcije ako i samo ako ima *funkcijsko svojstvo*: $R(\vec{x}, y_1) \wedge R(\vec{x}, y_2) \Rightarrow y_1 = y_2$. Riječima, „svaka vertikala siječe graf u najviše jednoj točki". ◁

Lema 3.43: Za svaki $k \in \mathbb{N}_+$, za svaku funkciju f^k, vrijedi $\exists_* \mathcal{G}_f = \mathcal{D}_f$ i $\mu \mathcal{G}_f = f$.

Dokaz. Za prvu jednakost, 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 zbog $\mathcal{G}_f\big(\vec{x}, f(\vec{x})\big)$ vrijedi $\vec{x} \in \exists_* \mathcal{G}_f$.

Dokažimo sada drugu jednakost. Prva jednakost kaže da te dvije funkcije imaju istu domenu (2.34); trebamo još 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 $y \neq y_0$ 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 $\mu y\, \mathcal{G}_f(\vec{x}, y) = y_0 = f(\vec{x})$. □

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 lemi 3.43. Dakle, f je parcijalno rekurzivna, a po pretpostavci teorema je totalna, pa je rekurzivna.

Za smjer ($\Leftarrow$): kako je f totalna, uvjet $\vec{x}^k \in \mathcal{D}_f = \mathbb{N}^k$ uvijek vrijedi, i $f(\vec{x})$ uvijek postoji, pa (3.73) postaje $\mathcal{G}_f(\vec{x}, y) \Longleftrightarrow y = f(\vec{x})$, odnosno $\chi_{\mathcal{G}_f}$ je dobivena kompozicijom iz $\chi_=$, f i koordinatnih projekcija. Jednakost i koordinatne projekcije su rekurzivne po korolarima 2.42 i 2.35, a f je rekurzivna po pretpostavci, pa je $\chi_{\mathcal{G}_f}$ rekurzivna po lemi 2.33. □

Istaknimo 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, opisana u [VukIzr15, dodatak], primjer je takve funkcije.

3.4.1. Kodiranje RAM-izračunavanja jednim brojem

Kako step nije totalna, ne možemo primijeniti teorem 3.44, ali možemo neke druge rezultate. Prema lemi 3.43 je $\mathsf{step} = \mu\,\mathsf{Step}$, 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 $\mathsf{Final}(x, e, n_0)$, vrijedi i $\mathsf{Final}(x, e, n)$ za sve $n > n_0$), ali možemo li dobiti jednu pomoću druge? $\mathsf{Final}(x, e, n) \Leftrightarrow (\exists m \leq n)\mathsf{Step}(x, e, m)$ znači da je Final dobivena ograničenom egzistencijalnom kvantifikacijom iz Step, ali nama treba drugi smjer.

Relacija $\mathsf{Step}(x, e, m) \Leftrightarrow m = \mu n\,\mathsf{Final}(x, e, n)$, koju dobijemo doslovnim čitanjem definicije $\mathsf{Step} = \mathcal{G}_{\mathsf{step}}$, čini se kao dobar početak: jedino što joj nedostaje je totalnost ove minimizacije na desnoj strani. Taj problem smo već imali nekoliko puta u točki 3.1.1 i uvijek smo ga uspješno rješavali ograničavanjem minimizacije. Postoji li gornja granica za n do koje je dovoljno provjeravati vrijedi li $\mathsf{Final}(x, e, n)$, 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 $\mathsf{Step}^3 := \mathcal{G}_{\mathsf{step}^2}$ je primitivno rekurzivna.

Dokaz. Kao što smo upravo rekli, cilj nam je dokazati

$$\mathsf{Step}(x, e, m) \iff m = (\mu n \leq m)\mathsf{Final}(x, e, n) \tag{3.74}$$

— iz toga će onda slijediti primitivna rekurzivnost prema (redom) lemi 3.35, propoziciji 2.58, napomeni 2.52 i korolaru 2.42.

Pretpostavimo da vrijedi $\mathsf{Step}(x, e, m)$. Tada vrijedi $(x, e) \in \mathcal{D}_{\mathsf{step}} = \mathrm{HALT}$ i $m = \mathsf{step}(x, e)$. Dakle vrijedi $\mathsf{Final}(x, e, m)$, i ni za koji $n < m$ ne vrijedi $\mathsf{Final}(x, e, n)$, pa je $\mu n\big(n \leq m \to \mathsf{Final}(x, e, n)\big) = \mu n\,\mathsf{Final}(x, e, n) = \mathsf{step}(x, e) = m$.

Ako pak ne vrijedi $\mathsf{Step}(x, e, m)$, tada negiranjem (3.73) vidimo da ili ne vrijedi $\mathrm{HALT}(x, e)$, ili pak postoji $s := \mathsf{step}(x, e)$, ali je različit (veći ili manji) od m. Tvrdimo da ni u kojem od tih slučajeva broj $t := (\mu n \leq m)\mathsf{Final}(x, e, n)$ nije jednak m.

Ako $\neg\mathrm{HALT}(x, e)$, tada ne postoji n takav da vrijedi $\mathsf{Final}(x, e, n)$, pa je

$$t = \mu n(n \leq m \to \bot) = \mu n \neg(n \leq m) = \mu n(n > m) = m + 1 \neq m. \tag{3.75}$$

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 posebno za svaki $n \leq m$ — vrijedi $\neg\mathsf{Final}(x, e, n)$, iz čega opet $t = m + 1 \neq m$. □

Sada se napokon možemo pozabaviti relacijama $\check{\mathsf{T}}$ 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.

$$\mathsf{T}_k(\vec{x}^k, e, y) :\iff (\exists P \in \mathcal{P}\mathrm{rog})(e = \ulcorner P \urcorner \wedge \text{„}y \text{ je kod } P\text{-izračunavanja s } \vec{x}\text{”}) \tag{3.76}$$

Propozicija 3.46: Za svaki $k \in \mathbb{N}_+$, relacija T_k je primitivno rekurzivna.

Dokaz. Kao što smo već rekli, prvo ćemo dokazati primitivnu rekurzivnost relacije $\check{\mathsf{T}}$,

$$\check{\mathsf{T}}(x, e, y) :\iff (\exists \vec{x} \in \mathbb{N}^+)\big(x = \langle \vec{x} \rangle \wedge \mathsf{T}_{\mathsf{lh}(x)}(\vec{x}, e, y)\big). \tag{3.77}$$

Tvrdimo da je njena točkovna definicija

$$\check{\mathsf{T}}(x,e,y) \iff \mathsf{Step}(x,e,n) \wedge y = \overline{\mathsf{Reg}}\big(x,e,\mathsf{Sc}(n)\big), \quad \text{uz pokratu } n := \mathsf{pd}\big(\mathsf{lh}(y)\big).$$

U jednom smjeru, pretpostavimo da vrijedi $\check{\mathsf{T}}(x,e,y)$. Tada prema (3.77) i (3.76) postoje $\vec{x} \in \mathbb{N}^+$ (njegovu duljinu označimo s k) i $P \in \mathcal{P}rog$ 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 $\mathrm{HALT}(x,e)$, pa postoji $n_0 := \mathsf{step}(x,e)$ — i y je upravo povijest stanja registara prvih n_0+1 (od c_0 do c_{n_0}) konfiguracija u tom izračunavanju.

Iz $n_0 = \mathsf{step}(x,e)$ slijedi $\mathsf{Step}(x,e,n_0)$, povijest stanja registara opisana je funkcijom $\overline{\mathsf{Reg}}$, a upravo smo vidjeli da je $\mathsf{lh}(y) = \mathsf{Sc}(n_0)$. Dakle, $n_0 = \mathsf{pd}\big(\mathsf{lh}(y)\big)$, što se upravo tvrdi u točkovnoj definiciji.

U drugom smjeru, pretpostavimo da vrijedi točkovna definicija. Tada iz $(x,e,n) \in \mathsf{Step} = \mathcal{G}_{\mathsf{step}}$ slijedi $(x,e) \in \mathcal{D}_{\mathsf{step}} = \mathrm{HALT}$ i $n := \mathsf{pd}\big(\mathsf{lh}(y)\big) = \mathsf{step}(x,e) = (\mu\,\mathsf{Final})(x,e)$. Specijalno vrijedi $\mathsf{Final}(x,e,n)$, 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 $n = \mathsf{step}(x,e)$.

Također vrijedi $y = \overline{\mathsf{Reg}}\big(x,e,\mathsf{Sc}(n)\big)$, dakle $\mathsf{lh}(y) = \mathsf{Sc}(n) > 0$. To znači da je y povijest stanja registara prvih $n+1$ konfiguracija, što je upravo kod tog izračunavanja.

Nažalost, nismo mogli napisati $y = \overline{\mathsf{Reg}}\big(x,e,\mathsf{lh}(y)\big)$, jer bi to zadovoljavao i kod praznog niza: $1 = \overline{G}\big(\vec{x},\mathsf{lh}(1)\big)$ bez obzira na specifikaciju funkcije G i vrijednosti argumenata $\vec{x}$. Na neki način, 0 je problematična kao $m = \mathsf{lh}(y)$, pa smo umjesto m napisali $\mathsf{Sc}\big(\mathsf{pd}(m)\big) = m\cdot$ u duhu napomene 2.26.

Sada za proizvoljni k primitivna rekurzivnost T_k slijedi iz (3.71). □

Korolar 3.47: Za svaki $k \in \mathbb{N}_+$, relacija T_k ima funkcijsko svojstvo, a projekcija joj je

$$\exists_* \mathsf{T}_k = \{(\vec{x},e) \in \mathbb{N}^{k+1} \mid \mathrm{HALT}(\langle\vec{x}\rangle,e)\} =: \mathrm{Halt}_k. \tag{3.78}$$

Dokaz. Za proizvoljne $\vec{x}$ i e, postoji najviše jedan RAM-program P s kodom e ($\ulcorner\cdots\urcorner$ je injekcija), pa onda postoji jedinstveno P-izračunavanje s $\vec{x}$ (propozicija 1.12), a ako ono stane, postoji jedinstven njegov kod. Ako P ne postoji, ili ako izračunavanje ne stane, ne postoji nijedan y takav da vrijedi $\mathsf{T}_k(\vec{x},e,y)$. Dakle, uvijek postoji *najviše* jedan takav y, a postoji *točno* jedan takav y ako i samo ako je $(\vec{x},e) \in \mathrm{Halt}_k$ — što je upravo tvrdnja koju smo željeli dokazati. □

Skupovi $\mathrm{Halt}_k, k \in \mathbb{N}_+$ čine familiju neizračunljivih relacija, svaka od kojih je fiksne mjesnosti i predstavlja određenu „krišku" univerzalne neizračunljive relacije HALT.

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 $\mu\mathsf{T}_k$, njena domena je Halt_k, i ona preslikava svaki $(\vec{x}^k,\ulcorner P\urcorner) \in \mathrm{Halt}_k$ (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

$$\mathsf{U}(y) := \mathsf{result}\big(\mathsf{rpart}(y,0)\big) = \mathsf{ex}(y[\mathsf{pd}(\mathsf{lh}(y))],0), \tag{3.79}$$

doslovno „sadržaj registra $\mathcal{R}_0$ u posljednjoj konfiguraciji kodiranoj s y", čime dobijemo

$$\mathsf{comp}_k(\vec{x},e) :\simeq \mathsf{U}\big(\mu y\, \mathsf{T}_k(\vec{x},e,y)\big). \tag{3.80}$$

Propozicija 3.48: Funkcija U je primitivno rekurzivna. Za svaki $k \in \mathbb{N}_+$, funkcija comp_k je parcijalno rekurzivna s domenom $\mathcal{D}_{\mathsf{comp}_k} = \mathtt{Halt}_k$ te vrijedi

$$\mathsf{univ}(\langle \vec{x} \rangle, e) \simeq \mathsf{comp}_k(\vec{x}, e). \tag{3.81}$$

Dokaz. Prvo, $\mathsf{U} = \mathsf{result} \circ \mathsf{rpart} \circ (\mathsf{I}_1^1, \mathsf{Z})$ je simbolička definicija od U: funkcija result je točkovno definirana u (3.49), a rpart u (3.33), pomoću primitivno rekurzivnih funkcija (ex, part, pd i lh), pa su result i rpart — a onda i U — primitivno rekurzivne.

Sada je $\mathsf{comp}_k = \mathsf{U} \circ \mu \mathsf{T}_k$ parcijalno rekurzivna jer je dobivena kompozicijom i minimizacijom iz primitivno rekurzivnih U i T_k. Prema korolaru 3.47 domena joj je $\mathtt{Halt}_k$, baš kao i domena funkcije $(\vec{x}, e) \mapsto \mathsf{univ}(\langle \vec{x} \rangle, e)$ (prema (2.4), jer je Code^k totalna). Još treba vidjeti da se te dvije funkcije podudaraju na toj domeni. Za svaki $(\vec{x}, e) \in \mathtt{Halt}_k$ (uz oznake $x := \langle \vec{x} \rangle$ i $n := \mathsf{step}(x, e)$, tako da vrijedi $\mathsf{Step}(x, e, n)$) imamo:

$$\begin{aligned}
\mathsf{univ}(x, e) &= \mathsf{result}\big(\mathsf{Reg}(x, e, n)\big) = \mathsf{result}\big(\overline{\mathsf{Reg}}(x, e, \mathsf{Sc}(n))[n]\big) = \\
&= \mathsf{result}\big(\mathsf{rpart}\big(\overline{\mathsf{Reg}}(x, e, \mathsf{Sc}(n)), 0\big)\big) = \mathsf{U}\big(\overline{\mathsf{Reg}}(x, e, \mathsf{Sc}(n))\big) = \\
&= \mathsf{U}\big(\mu y\big(y = \overline{\mathsf{Reg}}(x, e, \mathsf{Sc}(n))\big)\big) = \mathsf{U}\big(\mu y\big(y = \overline{\mathsf{Reg}}(x, e, \mathsf{Sc}(n)) \wedge \mathsf{Step}(x, e, n)\big)\big) = \\
&= \mathsf{U}\big(\mu y \check{\mathsf{T}}(x, e, y)\big) = \mathsf{U}\big(\mu y \mathsf{T}_k(\vec{x}, e, y)\big) = \mathsf{comp}_k(\vec{x}, e),
\end{aligned} \tag{3.82}$$

iz čega slijedi tražena parcijalna jednakost. □

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 $\mathsf{comp}_k(\vec{x}, e)$ rezultat tog izračunavanja.

2. U ostalim slučajevima ($e = \ulcorner P \urcorner$ tako da P-izračunavanje s $\vec{x}$ ne stane, ili uopće $e \notin \mathsf{Prog}$), izraz $\mathsf{comp}_k(\vec{x}, e)$ nema vrijednost.

Dokaz. Ovo je zapravo lema 3.37, iskazana koristeći parcijalnu jednakost (3.81) — i pojednostavljena jer je uvijek $\langle \vec{x}^k \rangle \in \mathsf{Seq}'$ za $k \in \mathbb{N}_+$, pa to ne treba pisati u uvjete. □

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, tako da za svaku parcijalno rekurzivnu funkciju f mjesnosti k postoji prirodni broj e, takav da za sve $\vec{x} \in \mathbb{N}^k$ vrijede sljedeće dvije tvrdnje:

$$\vec{x} \in \mathcal{D}_f \iff \exists y\, \mathsf{T}_k(\vec{x}, e, y), \tag{3.83}$$

$$f(\vec{x}) \simeq \mathsf{U}\big(\mu y\, \mathsf{T}_k(\vec{x}, e, y)\big). \tag{3.84}$$

Dokaz. Funkcija U je definirana s (3.79) i dokazano je da je primitivno rekurzivna u propoziciji 3.48. Za svaki $k \in \mathbb{N}_+$, relacija T_k je definirana s (3.76) i dokazano je da je primitivno rekurzivna u propoziciji 3.46. Neka je sada f proizvoljna parcijalno rekurzivna funkcija. Prema

teoremu 2.38, postoji RAM-program P koji računa f. Označimo $e := \ulcorner P \urcorner$, i neka je $\vec{x} \in \mathbb{N}^k$ proizvoljan.

Prema definiciji 1.11, ako je $\vec{x} \in \mathcal{D}_f$, tada P-izračunavanje s $\vec{x}$ stane, pa postoji njegov kod y_0. Tada po definiciji (3.76) 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 1.11 P-izračunavanje ne stane, pa nema kod, odnosno ne postoji y takav da vrijedi $T_k(\vec{x}, e, y)$. Time je dokazano (3.83), odnosno dokazano je da u (3.84) lijeva i desna strana imaju vrijednost za iste $\vec{x}$ (jer lijeva ima vrijednost 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 posljednjoj konfiguraciji kodiranoj s y_0 — što je po definiciji 1.11 jednako $f(\vec{x})$, jer ta konfiguracija mora biti završna. □

Korolar 3.51: Za svaku parcijalno rekurzivnu funkciju postoji simbolička definicija u kojoj se operator μ pojavljuje točno jednom.

Dokaz. Zapišimo (3.84) simbolički:

$$f = U \circ \mu T_k \circ (I_1^k, I_2^k, \ldots, I_k^k, C_e^k). \tag{3.85}$$

Vidimo da su U, T_k, I_n^k i C_e^k primitivno rekurzivne, pa se u njihovim simboličkim definicijama ne pojavljuje minimizacija. Dakle, jedina njena pojava služi za dobivanje funkcije μT_k. □

Kad fiksiramo U i sve T_k te pomoću njih definiramo funkcije comp_k (kao što smo učinili), Kleenejev teorem o normalnoj formi možemo izreći jednostavnije (i precizirati što je točno broj e o kojem taj teorem govori).

Definicija 3.52: Za fiksne $k \in \mathbb{N}_+$ i $e \in \mathbb{N}$, k-mjesnu brojevnu funkciju $\vec{x} \mapsto \mathrm{comp}_k(\vec{x}, e)$ označavamo s $\{e\}^k$, ili samo s $\{e\}$ ako joj ne trebamo istaknuti mjesnost.

Broj e zovemo *indeksom* funkcije $\{e\}^k$ (broj k je mjesnost funkcije $\{e\}^k$).

Za funkciju f^k kažemo da *ima indeks* ako postoji $e \in \mathbb{N}$ takav da je $\{e\}^k = f^k$. ◁

Korolar 3.53: Za sve $k \in \mathbb{N}_+$, za sve $e \in \mathbb{N}$, funkcija $\{e\}^k$ je parcijalno rekurzivna.

Dokaz. Simbolički, definicija 3.52 glasi $\{e\}^k := \mathrm{comp}_k \circ (I_1^k, I_2^k, \ldots I_k^k, C_e^k)$. Sada tvrdnja slijedi iz propozicije 3.48. □

Propozicija 3.54: Za sve $k \in \mathbb{N}_+$ i $e \in \mathbb{N}$ vrijedi:

1. Ako je $e \in \mathrm{Prog}$, tada je $\{e\}^k$ jedinstvena funkcija koju računa RAM-algoritam P^k, gdje je P jedinstveni RAM-program takav da je $\ulcorner P \urcorner = e$. (Geslo: „$P^k$ računa $\{\ulcorner P \urcorner\}^k$.")

2. Ako $e \notin \mathrm{Prog}$, tada je $\{e\}^k = \otimes^k$.

Dokaz. Ako je $e \in \mathrm{Prog}$, tada postoji $P \in \mathcal{P}\mathrm{rog}$ takav da je $\ulcorner P \urcorner = e$, i jedinstven je jer je $\ulcorner \cdots \urcorner$ injekcija. Po korolaru 1.14, postoji jedinstvena funkcija f^k koju P^k računa. Po definiciji 1.11, 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),

$f(\vec{x}) = \mathrm{comp}_k(\vec{x}, e) = \{e\}^k(\vec{x})$. Za sve pak $\vec{x} \notin \mathcal{D}_f$, P-izračunavanje s $\vec{x}$ ne stane, pa po korolaru 3.49(2) izraz $\mathrm{comp}_k(\vec{x}, e) \simeq \{e\}^k(\vec{x})$ nema vrijednost, baš kao ni $f(\vec{x})$. Dakle uvijek je $f(\vec{x}) \simeq \{e\}(\vec{x})$, odnosno $f = \{e\}$.

S druge strane, ako $e \notin \mathrm{Prog}$, tada opet po korolaru 3.49(2) izraz $\mathrm{comp}(\vec{x}, e) \simeq \{e\}(\vec{x})$ nema vrijednost, ali ovaj put neovisno o $\vec{x} \in \mathbb{N}^k$. Drugim riječima, tada je $\mathcal{D}_{\{e\}^k} = \varnothing^k$, odnosno jedino je moguće $\{e\}^k = \otimes^k$. □

Korolar 3.55: Svaka funkcija $f \in \mathcal{C}\mathrm{omp}$ je oblika $\{e\}^k$ za neke $k \in \mathbb{N}_+$ i $e \in \mathbb{N}$.

Dokaz. Za k stavimo mjesnost funkcije f, a za e kod nekog RAM-programa koji računa f (koji postoji jer je f RAM-izračunljiva).
Sada $f = \{e\}^k$ slijedi iz propozicije 3.54(1) i korolara 1.14. □

Korolar 3.56: Svaka parcijalno rekurzivna funkcija ima indeks.

Dokaz. Neka je $k \in \mathbb{N}_+$ te f^k parcijalno rekurzivna funkcija. Prema teoremu 2.38, $f \in \mathcal{C}\mathrm{omp}$. Sada prema korolaru 3.55 postoje k' i e takvi da je $f^k = \{e\}^{k'}$. Jednake funkcije moraju imati iste mjesnosti, pa je zapravo $k' = k$, odnosno $f^k = \{e\}^k$. □

Napomena 3.57: Ponekad se govori: „..., dakle funkcija f ima indeks, označimo ga s e_1". Tu frazu nije dobro koristiti, jer sugerira jedinstvenost indeksa, što je očito pogrešno: programerska intuicija nam kaže da svaki RAM-program ima beskonačno mnogo ekvivalentnih RAM-programa (možemo dodavati irelevantne ili nedostupne instrukcije). Kako kasnije najčešće ne koristimo nikakva svojstva broja e_1 osim da je indeks od f, zapravo taj izričaj treba shvatiti kao pokratu za „..., dakle funkcija f ima indeks; odaberimo jedan njen indeks, fiksirajmo ga u daljnjem razmatranju, i nazovimo ga e_1", kao što ćemo uglavnom pisati ubuduće.

Alternativno, možemo smatrati da smo uzeli *najmanji* indeks za f, koji sigurno postoji ako f ima indeks, i jednoznačno je određen zbog dobre uređenosti od $\mathbb{N}$, ali takav pristup ima jedan bitni 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 funkcijā, koja nije izračunljiva. Čak i da su funkcije totalne (što ne moraju biti), morali bismo provjeriti jednakost za sve $\vec{x}$ iz $\mathbb{N}^k$, kojih je beskonačno mnogo. ◁

Drugim, riječima, tromjesna relacija

$$\mathrm{index}(e, k, f) :\Longleftrightarrow \{e\}^k = f \tag{3.86}$$

između $\mathbb{N}$, $\mathbb{N}_+$ i $\mathcal{C}\mathrm{omp}$ nema funkcijsko svojstvo po prvoj varijabli — ali ima po trećoj: za svaki broj e i mjesnost k, propozicija 3.54 precizno opisuje funkciju $\{e\}^k$.

Tako index može poslužiti kao neka vrsta kodiranja: kad želimo algoritmu dati izračunljivu funkciju kao ulazni podatak, ili je vratiti kao izlazni podatak, možemo prenijeti neki njen indeks e (k se obično vidi iz konteksta). To nije doista kodiranje, jer nije jednoznačna funkcija ako kažemo „bilo koji indeks", a nije izračunljiva funkcija ako kažemo „najmanji indeks" — kao što je opisano u napomeni 3.57. Ipak, ako takav rad s indeksima shvatimo kao parcijalnu specifikaciju, to može funkcionirati.

Primjer 3.58: Zamislimo da imamo funkciju $F : \mathcal{C}omp_2 \rightharpoonup \mathcal{C}omp_3$, koja preslikava dvomjesne RAM-izračunljive funkcije u tromjesne RAM-izračunljive funkcije. Kao i prije, htjeli bismo izračunljivost funkcije F opisati pomoću izračunljivosti prateće funkcije $\mathbb{N}F$, koja prima indeks funkcije f i vraća indeks funkcije $F(f)$. Zbog napomene 3.57 to nije potpuni opis funkcije $\mathbb{N}F$, ali važno svojstvo koje želimo sačuvati je da *za svaki* $e \in \mathbb{N}$ bude

$$\{\mathbb{N}F(e)\}^3 = F(\{e\}^2). \tag{3.87}$$

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* takva funkcija $\mathbb{N}F$ koja je izračunljiva u nekom smislu i zadovoljava (3.87), kažemo da je F izračunljiva u istom tom smislu.

Takve funkcije možemo i komponirati: ako imamo $G : \mathcal{C}omp_3 \rightharpoonup \mathcal{C}omp_1$ 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. ◁

3.4.3. Restrikcije i grananja parcijalnih funkcija

Sheme poput one iz primjera 3.58 možemo kombinirati s „pravim" kodiranjima: recimo, za svaki $k \in \mathbb{N}_+$, funkcija $\mathcal{a}pply_k : \mathbb{N}^k \times \mathcal{C}omp_k \rightharpoonup \mathbb{N}$, 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.

$$\mathbb{N}\mathcal{a}pply_k(x, e) :\simeq \mathsf{univ}(x, e), \quad \text{za } \mathsf{lh}(x) = k \tag{3.88}$$

Napomena 3.59: Nažalost, iz (3.88) još uvijek ne slijedi da je $\mathbb{N}\mathcal{a}pply$ 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

$$(G|_R)(\vec{x}) \simeq G(\vec{x}) + \mu y\big(\mathsf{Sc}(y) = \chi_R(\vec{x})\big) \tag{3.89}$$

— ali zapravo će to trivijalno slijediti iz rezultata koji ćemo uskoro dokazati. ◁

Na početku točke 2.4 bilo je riječi o teškoćama koje marljiva evaluacija donosi 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, pa parcijalnost bilo koje od njih znači parcijalnost čitave kompozicije.

Dakle, način da se izvučemo je da našoj funkciji If ne damo već izračunane *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 $y?g(\vec{x}):h(\vec{x})$ imamo $(y?g:h)(\vec{x})$. Indeksi (kao „pokazivači na funkcije") nam omogućuju da tu tehniku doista provedemo.

Teorem 3.60 (Teorem o grananju, parcijalno rekurzivna verzija): Neka su $k, l \in \mathbb{N}_+$, neka su G_0^k, G_1^k, G_2^k, ..., G_l^k parcijalno rekurzivne funkcije, a R_1^k, R_2^k, ..., R_l^k u parovima disjunktne rekurzivne relacije, sve iste mjesnosti.

Tada je funkcija $F := \mathrm{if}\{R_1 : G_1, R_2 : G_2, \ldots, R_l : G_l, G_0\}$ također parcijalno rekurzivna.

Dokaz. Za svaki $i \in [0 \cdot\cdot\, l]$, funkcija G_i^k je parcijalno rekurzivna, pa prema korolaru 3.56 ima indeks; fiksirajmo jedan i označimo ga s e_i (pogledajte napomenu 3.57 za značenje ove fraze). Prema teoremu 2.46 (rekurzivna verzija), funkcija $H^k := \mathrm{if}\{R_1 : C_{e_1}^k, R_2 : C_{e_2}^k, \ldots, R_l : C_{e_l}^k, C_{e_0}^k\}$ je rekurzivna (konstante su primitivno rekurzivne pa su rekurzivne, a uvjeti su u parovima disjunktni i rekurzivni po pretpostavci). Tvrdimo da je

$$F(\vec{x}) \simeq \mathrm{comp}_k\big(\vec{x}, H(\vec{x})\big). \tag{3.90}$$

Doista, neka je $\vec{x} \in \mathbb{N}^k$ proizvoljan. Ako vrijedi $R_i(\vec{x})$ za neki (zbog disjunktnosti jedinstveni) $i \in [1 \cdot\cdot\, l]$, tada je po teoremu 2.46, $H(\vec{x}) = C_{e_i}(\vec{x}) = e_i$, pa je

$$\mathrm{comp}_k\big(\vec{x}, H(\vec{x})\big) \simeq \mathrm{comp}_k(\vec{x}, e_i) \simeq \{e_i\}(\vec{x}) \simeq G_i(\vec{x}) \simeq F(\vec{x}). \tag{3.91}$$

Ako pak ne vrijedi $R_i(\vec{x})$ ni za koji i, tada je opet po teoremu 2.46, $H(\vec{x}) = C_{e_0}(\vec{x}) = e_0$, pa je kao i prije $\mathrm{comp}_k\big(\vec{x}, H(\vec{x})\big) \simeq G_0(\vec{x}) \simeq F(\vec{x})$.

Sada parcijalna rekurzivnost slijedi iz (3.90), jer je F dobivena kompozicijom iz parcijalno rekurzivnih funkcija comp_k, H i koordinatnih projekcija. □

U teoremu 2.46 nismo mogli ispustiti G_0, jer njena podrazumijevana vrijednost $\otimes^k$ nije rekurzivna (nije uopće totalna). Ali $\otimes^k$ jest parcijalno rekurzivna, tako da ovdje možemo ispustiti granu „inače" — samo nam treba neki indeks za $\otimes^k$, da bi H bila totalna funkcija.

Za to možemo iskoristiti propoziciju 3.54(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, $\overline{\mathrm{part}}\big(0, \mathrm{lh}(0)\big) = \overline{\mathrm{part}}(0, 0) = \langle\rangle = 1 \neq 0$.

Korolar 3.61: Neka su $k, l \in \mathbb{N}_+$, neka su G_1^k, G_2^k, ..., G_l^k parcijalno rekurzivne funkcije, a R_1^k, R_2^k, ..., R_l^k u parovima disjunktne rekurzivne relacije, sve iste mjesnosti.

Tada je i funkcija $F := \mathrm{if}\{R_1 : G_1, R_2 : G_2, \ldots, R_l : G_l\}$ parcijalno rekurzivna.

Dokaz. Tvrdnja slijedi iz teorema 3.60, uvrštavajući za G_0 parcijalno rekurzivnu (primjer 2.31) funkciju $\otimes^k$, odnosno njen indeks $e_0 = 0$ u dokaz. □

Korolar 3.62: Neka je $k \in \mathbb{N}_+$, neka je G^k parcijalno rekurzivna funkcija i R^k rekurzivna relacija iste mjesnosti. Tada je i restrikcija $G|_R$ parcijalno rekurzivna.

Dokaz. Tvrdnja slijedi iz korolara 3.61, uvrštavajući $l := 1$, $G_1 := G$ te $R_1 := R$. Funkcija $\mathrm{if}\{R : G\}$ je upravo jednaka $G|_R$: domena joj je $\mathcal{D}_G \cap R$, a vrijednosti su joj jednake vrijednostima funkcije G na tom skupu. □

4. Turing-izračunljivost

Uveli smo dva modela izračunljivosti brojevnih funkcija — RAM-izračunljivost i parcijalnu rekurzivnost — i dokazali da su ekvivalentni, usprkos bitno različitim pristupima (imperativnom odnosno funkcijskom) definiciji algoritma. To je dobar argument za Church–Turingovu tezu, 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. Osnovna razlika je u konačnosti odnosno beskonačnosti skupova relevantnih za izračunavanje.

Primjerice, u RAM-modelu, skup mogućih stanja pojedinog registra je beskonačan. Često je problematično shvaćanje beskonačnosti kao „jako mnogo, pa još malo više". Zapravo, bliže definiciji beskonačnosti bilo bi „jako mnogo, 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 $\mathsf{univ}(\langle 2,4\rangle, e_0)$, odnosno konstrukciji rezultata $\mathrm{Q}^\flat$-izračunavanja s $(2,4)$, na univerzalnom RAM-stroju (dobivenom kompiliranjem simboličke definicije funkcije univ).

Stvarna računala, naravno, takve složene strukture (zamislimo e_0 kao *bytecode*, neku vrstu strojnog koda) 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 vrijedi univerzalni princip da na konačnim skupovima možemo dopustiti bilo kakve transformacije kao elementarne korake, reprezentirajući ih tablicama — dok s beskonačnim skupovima moramo biti oprezni, ograničavajući transformacije samo na one izračunljive. Kako su ti beskonačni skupovi u pravilu trivijalno izomorfni s $\mathbb{N}$, kao osnovne korake dopuštamo samo prijelaz 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 adresā relevantnih registara za fiksni RAM-algoritam ima konačno mnogo.

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 promatrane ćelije povećati ili smanjiti (ako nije 0) za jedan. To je *jezični model* izračunavanja, tako nazvan jer odgovara onom kako (barem 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) je bilo vrlo stvarno i nezgodno, i uostalom jedan od glavnih razloga za prijelaz na 64-bitnu arhitekturu.

Sličan primjer koji je nedavno došao do granice svojih mogućnosti je internetski protokol IPv4, ilustrativan jer daje uvid u 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 se koraka (DNS + *router*) može u idealnom slučaju adresirati $t^2 = 2^{64}$ računala. Više nam vjerojatno neće nikada trebati jer ćemo u međuvremenu prijeći na IPv6, ali u n koraka mogli bismo adresirati t^n računala, zamišljajući generaliziranu IP-adresu kao n-znamenkasti broj u bazi t — ili $32n$-znamenkasti broj u bazi 2. Jedina razlika kod jezičnog modela je u tome što umjesto binarnog zapisa adrese korisimo unarni, u kojem su elementarne operacije samo inkrement i dekrement (koji ostavlja nulu fiksnom).

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, i kao takva neće nam biti važna. Ono što nam jest važno je da u jezičnom modelu adresiranje pojedine ćelije zahtijeva netrivijalne algoritme, a bilo kakva promjena sadržaja ćelije je elementarna operacija — upravo suprotno od RAM-modela.

Drugi aspekt koji smo u potpunosti zanemarili u brojevnom modelu je povijesni. Danas, kad smo na svakom koraku okruženi računalima, a većina nās 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 nastao je prije modernog brojenja godina, a neki babilonski algoritmi potječu sa samih početaka pisane povijesti. Od kakve je koristi algoritam ako nema računala na kojem se može izvršavati?

Naravno, algoritme su izvršavali ljudi. Iako ljudski mozak po svojoj prirodi nije savršen supstrat za doslovno slijeđenje instrukcija kroz višeznamenkaste brojeve koraka, moderno

obrazovanje svjedoči da se može tome naučiti. Doista, rješavanje većine školskih matematičkih zadataka se može svesti na provođenje nekog algoritma. Mnoge takve zadatke računala rješavaju brže i točnije od ljudi, što se vidi kroz 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 daleko prije izuma računala. 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 funkcionira. No da bi to uspjelo, koraci algoritma moraju biti takvi da njegovo provođenje ne zahtijeva nikakav angažman pored onog specificiranog algoritmom — niti ikakvo vanjsko znanje osim poznavanja ulaznih podataka, i nekoliko elementarnih vještina za koje smatramo da su svojstvene svim radno sposobnim ljudima: čitanje i pisanje simbola iz fiksnog konačnog skupa te odlučivanje na osnovi pročitanog.

Britanski matematičar Alan Mathison Turing prvi je uspješno formalizirao taj koncept. U svom članku [Tur37], 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 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 primjerice λ-račun.

Ako modeliramo ljude, odakle onda Turingovi *strojevi*? Turing je bio svjestan fenomena da je 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 netrivijalno znanje o ljudskoj biologiji (mogu biti sestre jedna drugoj, ali ne mogu biti majke jedna drugoj) — a ipak nam se svaka od te dvije semantike nametne 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 (neki detalji su kasnije promijenjeni 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 osnovni opis algoritma lišili svega suvišnog.

4.1. Izračunljivost jezičnih funkcija

U literaturi postoje brojne varijante Turingova stroja — mi slijedimo [Sip13], uz male modifikacije kako bismo lakše dokazivali teoreme. Za početak ponovimo definicije.

Ulazna abeceda, ili samo *abeceda*, je konačan neprazan skup. Obično je označavamo sa Σ i smatramo fiksnom. Njene elemente zovemo *znakovima* — neodređene znakove (znakovne varijable) pišemo malim grčkim slovima s početka alfabeta (α, β, γ), dok konkretne znakove pišemo u fontu fiksne širine: $\mathtt{a}$, $\mathtt{b}$, $\mathtt{0}$, $\mathtt{1}$. *Riječ* (nad Σ) je bilo koji konačan niz znakova, najčešće označen slovom w, v ili u. Riječi pišemo konkatenacijom (nizanjem) znakova: recimo, riječ $(\mathtt{0}, \mathtt{1}, \mathtt{1})$ pišemo kao $\mathtt{011}$. Dakle, skup svih riječi je skup Σ^* svih konačnih nizova znakova. Duljinu riječi w označavamo s $|w|$. Prazni niz (duljine 0) 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 $\varphi : \Sigma^* \rightharpoonup \Sigma^*$.

Napomena 4.1: Kao i mjesnost kod brojevnih funkcija i relacija, tako i abecedu kod jezika i jezičnih funkcija smatramo dijelom njihova identiteta; preslikavanje nad $\{\mathtt{a}, \mathtt{b}\}$ koje riječi pridružuje njen reverz (obrnuto čitanu riječ), različito je od preslikavanja nad $\{\mathtt{a}, \mathtt{c}\}$ zadanog istim pravilom. Ili, jezik svih riječi koje se sastoje samo od znakova $\mathtt{a}$ i $\mathtt{b}$ je različit kao jezik nad $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ i kao jezik nad $\{\mathtt{a}, \mathtt{b}, \mathtt{d}\}$ — 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 tih 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:

- konačnu *radnu abecedu* $\Gamma \supset \Sigma$, s istaknutim elementom $\sqcup \in \Gamma \setminus \Sigma$ (*praznina*);
- konačni skup *stanja* Q, s elementima $q_0 \in Q$ (*početno* stanje) i $q_z \in Q$ (*završno* stanje);
- konačnu *funkciju prijelaza* $\delta : (Q \setminus \{q_z\}) \times \Gamma \to Q \times \Gamma \times \{-1, 1\}$. $\triangleleft$

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 Turingova stroja sve ćelije osim ulaznih biti inicijalizirane na $\sqcup$. Po analogiji s $\mathbb{N}_+ = \mathbb{N} \setminus \{0\}$ 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 $w = \alpha_0\alpha_1 \cdots \alpha_{|w|-1} \in \Sigma^*$ je trojka $(q_0, 0, w\sqcup\ldots)$, čija je traka definirana s $(w\sqcup\ldots)_i := (\alpha_i$ ako je $i < |w|$, a inače $\sqcup)$.

Za konfiguracije $c = (q, n, t)$ i $d = (q', n', t')$ Turingova stroja $\mathcal{T}$ kažemo da c *prelazi* u d, pišući $c \rightsquigarrow d$, ako je c završna i $c = d$ (pišemo $c \circlearrowleft$), 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 trenutno stanje i sadržaj trenutne ćelije 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 $v\sqcup\ldots$ za neku riječ $v \in \Sigma^*$, kažemo da je v izlaz Turingova stroja s ulazom w.

Lema 4.4: Svaka konfiguracija Turingova stroja prelazi u točno jednu konfiguraciju.

Dokaz. Neka je $\mathcal{T}$ Turingov stroj te $c = (q, n, t)$ 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 $q' := p$, $n' := \max\{n + d, 0\}$ i niz $t' := \left(\left\{\begin{smallmatrix}\beta, & j=n\\ t(j), & \text{inače}\end{smallmatrix}\right.\right)_{j\in\mathbb{N}}$ 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, a 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 $\big(q_z, n, \varphi(w)\sqcup\ldots\big)$ za neki $n \in \mathbb{N}$ (pozicija nije bitna).
- Ako $w \notin \mathcal{D}_\varphi$, tada $\mathcal{T}$-izračunavanje s w ne stane.

Jezična funkcija φ 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 ne i da svaki Turingov stroj računa neku jezičnu funkciju. Traka u završnoj konfiguraciji ne mora biti oblika $v\sqcup\ldots$ za $v \in \Sigma^*$: može sadržavati znakove iz $\Gamma_+ \setminus \Sigma$, ili sadržavati neki znak iz Σ nakon prve praznine. Za takve „patološke" Turingove strojeve nećemo reći da računaju ikakvu funkciju — zapravo ih nećemo uopće promatrati, ali ako želimo iskazati neku tvrdnju univerzalno po svim Turingovim strojevima, dobro je imati na umu da i takvi postoje.

Primjer 4.6: Nad $\Sigma := \{a, b\}$ promotrimo funkciju $\varphi_h : \Sigma^* \rightharpoonup \Sigma^*$, čija je domena skup svih riječi parne duljine, a $\varphi_h(\alpha_1\alpha_2\cdots\alpha_{2k}) := \alpha_1\alpha_2\cdots\alpha_k$ (prva polovica riječi).

Primjerice, za $w_1 := \mathtt{aababa}$ vrijedi $|w_1| = 6$, pa je $w_1 \in \mathcal{D}_{\varphi_h}$ te $\varphi_h(w_1) = \mathtt{aab}$.

S druge strane, za $w_2 := \mathtt{bbb}$ vrijedi $|w_2| = 3$, pa $w_2 \notin \mathcal{D}_{\varphi_h}$.

Funkcija φ_h je Turing-izračunljiva: računa je Turingov stroj

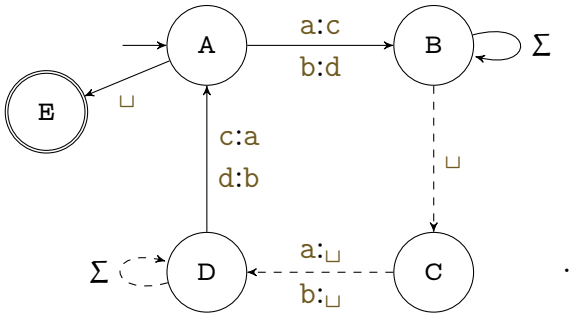
$$\mathcal{T}_h := (\{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}, \mathtt{E}, \mathtt{F}\}, \Sigma, \{\sqcup, a, b, c, d\}, \sqcup, \delta_h, \mathtt{A}, \mathtt{E}), \tag{4.1}$$

čija je funkcija prijelaza (kao konačna funkcija) zadana tablicom

δ_h	$\sqcup$	a	b	c	d
A	$(\mathtt{E}, \sqcup, +1)$	$(\mathtt{B}, c, +1)$	$(\mathtt{B}, d, +1)$	$(\mathtt{F}, c, -1)$	$(\mathtt{F}, d, -1)$
B	$(\mathtt{C}, \sqcup, -1)$	$(\mathtt{B}, a, +1)$	$(\mathtt{B}, b, +1)$	$(\mathtt{F}, c, -1)$	$(\mathtt{F}, d, -1)$
C	$(\mathtt{F}, \sqcup, -1)$	$(\mathtt{D}, \sqcup, -1)$	$(\mathtt{D}, \sqcup, -1)$	$(\mathtt{F}, c, -1)$	$(\mathtt{F}, d, -1)$
D	$(\mathtt{F}, \sqcup, -1)$	$(\mathtt{D}, a, -1)$	$(\mathtt{D}, b, -1)$	$(\mathtt{A}, a, +1)$	$(\mathtt{A}, b, +1)$
F	$(\mathtt{F}, \sqcup, -1)$	$(\mathtt{F}, a, -1)$	$(\mathtt{F}, b, -1)$	$(\mathtt{F}, c, -1)$	$(\mathtt{F}, d, -1)$

. (4.2)

Kao što vidimo, takav način zadavanja Turingova stroja nije naročito čitljiv — zato se obično crtaju dijagrami. Recimo, $\mathcal{T}_h$ bismo mogli prikazati dijagramom



. (4.3)

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 prikazujemo strelicom s više oznaka $\alpha_1{:}\beta_1, \ldots, \alpha_k{:}\beta_k$. Ako se znak ne mijenja ($\alpha = \beta$), umjesto $\alpha{:}\alpha$ pišemo samo α. Oznaku za skup $S \subseteq \Gamma$ možemo 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 Turingova stroja imati jedno stanje q_x (za $\mathcal{T}_h$ to je F), takvo da δ svaku „nemoguću situaciju" (q, α) preslika u $(q_x, \alpha, -1)$. Specijalno to vrijedi i za $q = q_x$ (za sve $\alpha \in \Gamma$) pa Turingov stroj, ako se ikad nađe u nekoj od tih nedozvoljenih konfiguracija, više nikada neće stati.

Efektivno, to je „stanje greške" koje ne moramo (kao ni prijelaze prema njemu) 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 Turingova stroja u konkretnom izračunavanju obično se označavaju skraćeno: ispod trenutno čitanog znaka napišemo trenutno stanje, odnosno umjesto $\big(q, n, (t_m)_{m \in \mathbb{N}}\big)$ pišemo $t_0 t_1 \ldots \underset{q}{t_n} t_{n+1} \ldots$ — u takvom zapisu ne pišemo (ali podrazumijevamo) $\sqcup \ldots$ na kraju.

Koristeći tu notaciju, početna konfiguracija $\mathcal{T}_h$ s ulazom abaa je $\underset{A}{a}baa$.

Tada je $\mathcal{T}_h$-izračunavanje s abaa

$$\underset{A}{\mathtt{abaa}} \rightsquigarrow \underset{B}{\mathtt{cbaa}} \rightsquigarrow \underset{B}{\mathtt{cbaa}} \rightsquigarrow \underset{B}{\mathtt{cbaa}} \rightsquigarrow \underset{B}{\mathtt{cbaa}\sqcup} \rightsquigarrow \underset{C}{\mathtt{cbaa}} \rightsquigarrow \underset{D}{\mathtt{cba}} \rightsquigarrow \underset{D}{\mathtt{cba}} \rightsquigarrow \underset{D}{\mathtt{cba}} \rightsquigarrow$$
$$\rightsquigarrow \underset{A}{\mathtt{aba}} \rightsquigarrow \underset{B}{\mathtt{ada}} \rightsquigarrow \underset{B}{\mathtt{ada}\sqcup} \rightsquigarrow \underset{C}{\mathtt{ada}} \rightsquigarrow \underset{D}{\mathtt{ad}} \rightsquigarrow \underset{A}{\mathtt{ab}\sqcup} \rightsquigarrow \underset{E}{\mathtt{ab}\sqcup\sqcup} \circlearrowleft, \tag{4.4}$$

iz čega se vidi izlazni podatak $\mathtt{ab} = \varphi_h(\mathtt{abaa})$. S druge strane, $\mathcal{T}_h$-izračunavanje s aba

$$\underset{A}{\mathtt{aba}} \rightsquigarrow \underset{B}{\mathtt{cba}} \rightsquigarrow \underset{B}{\mathtt{cba}} \rightsquigarrow \underset{B}{\mathtt{cba}\sqcup} \rightsquigarrow \underset{C}{\mathtt{cba}} \rightsquigarrow \underset{D}{\mathtt{cb}} \rightsquigarrow \underset{D}{\mathtt{cb}} \rightsquigarrow \underset{A}{\mathtt{ab}} \rightsquigarrow \underset{B}{\mathtt{ad}\sqcup} \rightsquigarrow \underset{C}{\mathtt{ad}} \rightsquigarrow \underset{F}{\mathtt{ad}} \rightsquigarrow \underset{F}{\mathtt{ad}} \rightsquigarrow \cdots \tag{4.5}$$

očito ne stane, što je u skladu s tim da $\mathtt{aba} \notin \mathcal{D}_{\varphi_h}$ (jer je $|\mathtt{aba}| = 3$ neparan broj). ◁

4.1.1. Prateće funkcije

Kao što je već najavljeno, cilj je pokazati ekvivalentnost Turing-modela s RAM-modelom i funkcijskim modelom izračunljivosti. Za početak ćemo koristeći pristup sličan onome iz poglavlja 3 dokazati da su Turing-izračunljive jezične funkcije „parcijalno rekurzivne”.

Prvo moramo precizirati što to znači. Neka je Σ abeceda i φ jezična funkcija nad njom. Očito φ ne možemo dobiti kompozicijom, primitivnom rekurzijom i minimizacijom iz inicijalnih (brojevnih) funkcija. I da smislimo neke „inicijalne jezične funkcije”, kompozicija je jasna (čak jednostavnija nego u brojevnom slučaju, jer su sve funkcije jednomjesne), ali kako definirati primitivnu rekurziju kad sljedbenik riječi nije jedinstven? Kako definirati minimizaciju kad kanonski leksikografski uređaj na Σ^* nije dobar uređaj? Kako uopće definirati izračunljive jezike („relacije”) kad karakteristična funkcija jezika nije ni brojevna 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 $\sigma := \mathbb{N}\Sigma^*$ (po uzoru na kodiranje $\mathbb{N}^*$ označavamo $\langle w \rangle := \sigma(w)$) te ćemo pomoću tog kodiranja definirati prateću funkciju $\mathbb{N}\varphi = \sigma \circ \varphi \circ \sigma^{-1}$, za koju znamo što znači da je parcijalno rekurzivna. Štoviše, pazit ćemo da σ bude *bijekcija* između Σ^* i $\mathbb{N}$, tako da će prateće funkcije totalnih funkcija biti opet totalne funkcije. Onda će i *rekurzivne* jezične funkcije biti dobro definirane — kao totalne izračunljive jezične funkcije, odnosno one čije su prateće funkcije rekurzivne.

Prvo kodirajmo abecedu Σ. Kako je to konačan skup iskoristit ćemo enum-tehniku, kao što smo napravili kod kodiranja tipova instrukcija RAM-stroja (3.40). Ovdje počinjemo brojiti od 1 (želimo $0 \notin \mathcal{I}_{\mathbb{N}\Sigma}$) iz tehničkog razloga koji će uskoro biti jasan.

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 Σ ...”, podrazumijevajući („Zermelov teorem za konačne skupove”) da kodiranje *postoji*. U praksi, znakovi će obično biti dio standarda Unikod, pa ih u nedostatku pametnijeg kriterija možemo poredati po rednim brojevima u Unikodu.

Definicija 4.7: Neka je Σ abeceda. Označimo $b := \operatorname{card} \Sigma$.

Kodiranje od Σ je bilo koja bijekcija $\mathbb{N}\Sigma : \Sigma \leftrightarrow [1 \cdot\cdot\, b]$. ◁

Sada bismo mogli kao u definiciji 3.27 kodirati riječi kao kodove konačnih nizova kodova njihovih znakova — ali to ne bi bilo bijektivno. Osim toga, kodovi RAM-instrukcija mogli su biti proizvoljno veliki, dok za fiksnu abecedu kodovi znakova mogu biti najviše b — što 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 višeznamenkaste brojeve, je **zapis u bazi** b. Problem su početne nule ($(001121)_3 = (1121)_3$), što smo riješili početkom kodiranja znakova od 1, ali smo zato dobili uključenu gornju granicu. Možemo li doista imati znamenku b u bazi b? Drugi problem su jednočlane abecede: obični pozicijski brojevni sustavi definiraju se samo za baze $b \geq 2$. 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 zato nemamo znamenku 0, i ne smijemo je imati ako želimo jedinstvenost zapisa — no to upravo rješava prvi problem: ne možemo imati nule na početku ako ih nemamo uopće.

Definicija 4.8: Neka je $b \in \mathbb{N}_+$, $n \in \mathbb{N}$ te $z_0, z_1, \ldots, z_{n-1} \in [1 \cdot\cdot\, b]$. Zapis u *pomaknutoj bazi* b je

$$(z_{n-1} \cdots z_0)_b := \sum_{i<n} z_i \cdot b^i, \tag{4.6}$$

dakle isti kao obični zapis u bazi b, samo sa znamenkama iz $[1 \cdot\cdot\, b]$ umjesto iz $[0 \cdot\cdot\, b\rangle$. ◁

Lema 4.9: Za svaki $b \in \mathbb{N}_+$, svaki $x \in \mathbb{N}$ ima jedinstveni zapis u pomaknutoj bazi b.

Dokaz. Dokaz egzistencije je isti kao i za običnu bazu b: uzastopno dijelimo x s b dok ne dobijemo nulu pa 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 primitivno rekurzivnoj verziji teorema o grananju.

$$\mathsf{mod}'(x, b) := \begin{cases} b, & b \mid x \\ x \bmod b, & \text{inače} \end{cases} \tag{4.7}$$

Za dokaz jedinstvenosti treba vidjeti da brojevi različitih duljina zapisa, kao i oni iste duljine zapisa koji se u nekoj znamenci razlikuju, moraju biti različiti. Obje te tvrdnje slijede iz

$$\begin{aligned}(t z_{m-1} z_{m-2} \cdots z_1 z_0)_b &= t \cdot b^m + z_{m-1} \cdot b^{m-1} + z_{m-2} \cdot b^{m-2} + \cdots + z_1 \cdot b + z_0 \leq \\ &\leq t \cdot b^m + b \cdot b^{m-1} + b \cdot b^{m-2} + \cdots + b \cdot b + b < t \cdot b^m + b^m + 1 \cdot b^{m-1} + \cdots + 1 \cdot b^2 + 1 \cdot b + 1 \leq \\ &\leq (t+1) b^m + z'_{m-1} \cdot b^{m-1} + \cdots + z'_2 \cdot b^2 + z'_1 \cdot b^1 + z'_0 \cdot b^0 = \left((t+1) z'_{m-1} \cdots z'_2 z'_1 z'_0\right)_b \end{aligned} \tag{4.8}$$

— prva za $t := 0$, a druga za $t :=$ (prva znamenka slijeva u kojoj se zapisi razlikuju).

To po tranzitivnosti zapravo znači da smo zapise poredali po duljini, a zapise iste duljine leksikografski (tzv. *shortlex ordering*). □

Primjer 4.10: *Shortlex ordering* $\{a, b\}^*$ je: $\underset{0}{\varepsilon}, \underset{1}{a}, \underset{2}{b}, \underset{3}{aa}, \underset{4}{ab}, \underset{5}{ba}, \underset{6}{bb}, \underset{7}{aaa}, \underset{8}{aab}, \underset{9}{aba}, \ldots$. ◁

Definicija 4.11: Neka je Σ abeceda te $\mathbb{N}\Sigma$ njeno kodiranje.

Definiramo kodiranje skupa Σ^* svih riječi nad Σ, s

$$\mathbb{N}\Sigma^*(w) := \langle \alpha_{n-1} \cdots \alpha_0 \rangle := \big(\mathbb{N}\Sigma(\alpha_{n-1}) \cdots \mathbb{N}\Sigma(\alpha_0)\big)_b = \sum_{i<n} \mathbb{N}\Sigma(\alpha_i) \cdot b^i, \tag{4.9}$$

za svaku riječ $w = \alpha_{n-1} \cdots \alpha_0 \in \Sigma^*$ (uz oznake $n := |w|$ i $b := \operatorname{card} \Sigma$). ◁

Primjer 4.12: U primjeru 4.6 uveli smo abecedu $\Sigma = \{a, b\}$ i riječ $w_1 = aababa$ nad njom. Za tu abecedu je $b = 2$; fiksirajmo kodiranje $\mathbb{N}\Sigma(a) := 1$, $\mathbb{N}\Sigma(b) := 2$. Tada je $\langle w_1 \rangle = (112121)_2 = 73$. Također je $w_1 \in \mathcal{D}_{\varphi_h}$, pa je $73 \in \mathcal{D}_{\mathbb{N}\varphi_h}$; a iz $\varphi_h(w_1) = aab$ vidimo $\mathbb{N}\varphi_h(73) = \langle aab \rangle = (112)_2 = 8$.

Za broj 153 pretvaranje u pomaknutu bazu 2 daje $\begin{smallmatrix} 153 & 76 & 37 & 18 & 8 & 3 & 1 \\ 1 & 2 & 1 & 2 & 2 & 1 & 1 \end{smallmatrix}$, dakle $153 = (1122121)_2 = \langle aabbaba \rangle$. Jer je $|aabbaba| = 7$ neparan broj, $aabbaba \notin \mathcal{D}_{\varphi_h}$, pa $153 \notin \mathcal{D}_{\mathbb{N}\varphi_h}$. ◁

Za dekodiranje trebamo, analogno propoziciji 3.14, duljinu zapisa te ekstrakciju pojedine znamenke u zadanoj pomaknutoj bazi.

Lema 4.13: Postoje primitivno rekurzivne funkcije slh^2, sdigit^3, $\mathsf{sconcat}^3$ i $\mathsf{sprefix}^3$, takve da za sve $b \in \mathbb{N}_+$, za sve $m, n \in \mathbb{N}$ vrijedi:

$\mathsf{slh}(n, b)$ je duljina zapisa broja n u pomaknutoj bazi b.

$\mathsf{sdigit}(n, i, b)$ za $i < \mathsf{slh}(n, b)$, je vrijednost i-te znamenke zapisa broja n u pomaknutoj bazi b, gdje brojimo od nule slijeva.

$\mathsf{sconcat}(m, n, b)$ je broj čiji se zapis u pomaknutoj bazi b dobije konkatenacijom istih takvih zapisa za m i n redom. Pišemo ga i kao $m \mathbin{\widehat{\mathsf{b}}} n$.

$\mathsf{sprefix}(n, i, b)$ za $i \le \mathsf{slh}(n, b)$, je broj čiji je zapis u pomaknutoj bazi b prefiks (početak) duljine i istog takvog zapisa broja n.

$$\mathsf{slh}(n, b) := (\mu t \le n)\big(\textstyle\sum_{i \le t} b^i > n\big) \tag{4.10}$$

$$\mathsf{sconcat}(m, n, b) := m \mathbin{\widehat{\mathsf{b}}} n := m \cdot b^{\mathsf{slh}(n,b)} + n \tag{4.11}$$

$$\mathsf{sprefix}(n, i, b) := (\mu m \le n)(\exists t \le n)\big(n = m \mathbin{\widehat{\mathsf{b}}} t \wedge \mathsf{slh}(m, b) = i\big) \tag{4.12}$$

$$\mathsf{sdigit}(n, i, b) := \mathsf{mod}'\big(\mathsf{sprefix}\big(n, \mathsf{Sc}(i), b\big), b\big) \tag{4.13}$$

Dokaz. Za slh, treba uočiti da zapisā duljine i ima točno b^i. Dakle, samo trebamo zbrajati takve brojeve dok ne prijeđemo n. Očito je $\mathsf{slh}(n, b) \le n$. (4.10)

Za konkatenaciju, m pomaknemo udesno za duljinu od n, i pribrojimo n. (4.11)

Pomoću konkatenacije možemo dobiti $\mathsf{sprefix}$: prefiks je ono što se može konkatenirati s nečim tako da se dobije početni zapis n. Očito su dijelovi manji ili jednaki n. (4.12)

Sada možemo napisati i sdigit: to je zadnja znamenka onog prefiksa čija je duljina za jedan veća od pozicije znamenke. (4.13) □

Propozicija 4.14: Neka je Σ abeceda s kodiranjem $\mathbb{N}\Sigma$. Tada je $\mathbb{N}\Sigma^*$ bijekcija između Σ^* i $\mathbb{N}$.

Dokaz. Označimo $b := \operatorname{card}\Sigma$. Ako je $w \neq w'$ za $w, w' \in \Sigma^*$, tada su w i w' ili različitih duljina (pa je $\mathsf{slh}(\langle w\rangle, b) \neq \mathsf{slh}(\langle w'\rangle, b)$), ili su jednake duljine ali se na nekom mjestu razlikuju (pa je $\mathsf{sdigit}(\langle w\rangle, i, b) \neq \mathsf{sdigit}(\langle w'\rangle, i, b)$ za neki i). U svakom slučaju $\langle w\rangle \neq \langle w'\rangle$, pa je $\mathbb{N}\Sigma^*$ injekcija.

Za surjektivnost, neka je $x \in \mathbb{N}$ proizvoljan. Prema lemi 4.9, postoje znamenke $\vec{z} \in [1 \cdot\cdot\, b]^*$ takve da je $x = (\vec{z})_b$. Ako sad za svaku z_i označimo $\alpha_i := \mathbb{N}\Sigma^{-1}(z_i) \in \Sigma$, riječ $\vec{\alpha}$ sastavljena od tih znakova ima upravo kod x. □

Definicija 4.15: Neka je Σ abeceda, $\mathbb{N}\Sigma$ njeno kodiranje te $\varphi : \Sigma^* \rightharpoonup \Sigma^*$ jezična funkcija nad njom. *Prateća funkcija* od φ je brojevna funkcija $\mathbb{N}\varphi$ s domenom $\mathbb{N}\Sigma^*[\mathcal{D}_\varphi]$, zadana s

$$\mathbb{N}\varphi(\langle w\rangle) :\simeq \langle \varphi(w)\rangle. \tag{4.14}$$

Kako je svaki prirodni broj kod jedinstvene riječi iz Σ^*, definicija je dobra. ◁

4.1.2. Kodiranje stanja, trake i funkcije prijelaza

Kroz čitavu ovu točku imat ćemo fiksiranu abecedu Σ_0, njeno kodiranje $\mathbb{N}\Sigma_0$ uz oznaku $b' := \operatorname{card}\Sigma_0$, 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 je računa. 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 $a := \operatorname{card} Q_0$ i fiksirati bijekciju $\mathbb{N}Q_0 : Q_0 \leftrightarrow [0 \cdot\cdot\, a\rangle$. 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 fiksni broj — recimo, $\mathbb{N}Q_0(q_z) := 1$.

Hm, hoće li onda $\mathbb{N}Q_0$ biti dobro definirana: što ako je $q_0 = q_z$? Definicija Turingova stroja ne sprečava tu mogućnost. Ipak, $q_0 \neq q_z$ smijemo pretpostaviti bez smanjenja općenitosti.

Lema 4.16: Za svaki Turingov stroj $\mathcal{T}$ koji računa neku jezičnu funkciju φ, postoji *ekvivalentni* (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, $(q_0, 0, w\sqcup\ldots) = (q_z, 0, w\sqcup\ldots)$, ujedno je i završna konfiguracija, pa 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. Jedan takav je

$$\mathcal{T}' := (\{0, 1\}, \Sigma, \Sigma \cup \{\sqcup\}, \sqcup, \delta', 0, 1), \tag{4.15}$$

$$\delta'(0, \alpha) := (1, \alpha, 1) \text{ za sve } \alpha \in \Sigma \cup \{\sqcup\}. \tag{4.16}$$

Tada za svaku $w \in \Sigma^*$, $\mathcal{T}'$-izračunavanje s w je $(0, 0, w\sqcup\ldots) \rightsquigarrow (1, 1, w\sqcup\ldots) \circlearrowleft$, odakle je izlazni podatak opet w, odnosno $\mathcal{T}'$ računa identitetu. □

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. Stoga označimo $b := \operatorname{card}\Gamma_0$ i proširimo bijekciju $\mathbb{N}\Sigma_0 : \Sigma_0 \leftrightarrow [1 \cdot\cdot\, b']$ do bijekcije $\mathbb{N}\Gamma_0 : \Gamma_0 \leftrightarrow [0 \cdot\cdot\, b\rangle$, tako da $\sqcup$ preslikamo u 0, a ostale elemente iz $\Gamma_0 \setminus \Sigma_0$ bijektivno u skup $\langle b' \cdot\cdot\, b\rangle$.

Ovaj put smo upotrijebili nulu, kao $\mathbb{N}\Gamma_0(\sqcup)$. To opravdava frazu „traka je s konačnim nosačem" (praznine su kodirane nulama, pa ne-nulā 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 takvo 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?

Možemo, 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 $(001121)_3 = (1121)_3 = 43$, no to upravo znači da je broj 43 prirodno gledati kao kod trake $\texttt{abaa}\sqcup\sqcup\sqcup\ldots = \texttt{abaa}\sqcup\ldots$ — drugim riječima, ovdje imamo **obični zapis u (ne pomaknutoj) bazi** b.

Definicija 4.17: Za proizvoljnu traku $t : \mathbb{N} \to \Gamma_0$ (koja je skoro svuda $\sqcup$), definiramo *kod trake*

$$\langle t\rangle = \langle t_0 t_1 t_2 \ldots \sqcup\sqcup\ldots\rangle := \sum_{i\in\mathbb{N}} \mathbb{N}\Gamma_0(t_i)\cdot b^i, \text{ gdje je } b := \operatorname{card}\Gamma_0. \tag{4.17}$$

Zbog konačnosti 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. ◁

S ovom idejom postoje dvije smetnje pri konstrukciji početne konfiguracije. Prvo, **moramo promijeniti bazu** iz $\operatorname{card}\Sigma_0 = b'$ u $\operatorname{card}\Gamma_0 = b$, jer na traci se mogu naći i znakovi s većim kodovima, kao i praznina (**uvijek je** $b > b'$, zbog $\Gamma \supset \Sigma$). Drugo, u zapadnom svijetu pišemo riječi slijeva nadesno (i tako doživljavamo traku), ali smo zapis brojeva preuzeli iz arapskog jezika 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.

Računarci s iskustvom mrežnog programiranja znaju za taj problem. Radi se o *byteorder*-dilemi, sasvim analognoj upravo opisanom problemu, iz sličnih razloga. Većina modernih procesora pamti višebajtne podatke tako da na početnoj adresi podatka stoji najmanje značajni bajt, jer se jedino tako postiže neovisnost o veličini ćelije (nakon `y=25;` na adresi `&y` je bajt 25, neovisno o tome je li varijabla `y` tipa `char`, `int` ili `long long`), što pojednostavljuje računanje.

No taj redoslijed (*little-endian*) je obrnut u odnosu na to kako smo navikli pisati brojeve u dekadskom zapisu — i kako smo uostalom napisali taj broj (2 pa onda 5) u naredbi koja inicijalizira `y`. To posebno dolazi do izražaja pri mrežnom programiranju: standardi propisuju da se višebajtni brojevi preko mreže prenose redoslijedom *big-endian*, kako bi bilo lakše pratiti što se događa u slučaju greške. Zato mnoge standardne biblioteke imaju funkcije (`ntoh`/`hton`) za pretvaranje redoslijeda bajtova iz mrežnog (čitljivijeg) u procesorski (operativniji) i obrnuto. Mi ćemo napraviti slično, usput prenoseći zapis iz jedne baze u drugu.

Lema 4.18: Postoji primitivno rekurzivna funkcija Recode[3] takva da za svaku riječ $w \in \Sigma_0^*$ vrijede jednakosti

$$\mathsf{Recode}\big(\langle w\rangle, b', b\big) = \langle w_{\sqcup}\ldots\rangle, \tag{4.18}$$

$$\mathsf{Recode}\big(\langle w_{\sqcup}\ldots\rangle, b, b'\big) = \langle w\rangle. \tag{4.19}$$

Dokaz. Kao što smo već rekli, $\mathsf{Recode}(n, b_1, b_2)$ samo treba ekstrahirati znamenke od n u bazi b_1 te ih slagati obrnutim redom u bazi b_2. 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 b_1 pomaknuta.

$$\mathsf{Recode}(n, b_1, b_2) := \sum_{i<\mathsf{slh}(n,b_1)} \mathsf{sdigit}(n, i, b_1) \cdot {b_2}^i \tag{4.20}$$

Neka je $w = \alpha_0\alpha_1\cdots\alpha_{l-1} \in \Sigma^*$ proizvoljna ($l := |w|$). Tada, ako označimo $n := \langle w\rangle$, vrijedi $\mathsf{slh}(n, b') = l$ i $d_i := \mathsf{sdigit}(n, i, b') = \mathbb{N}\Sigma_0(\alpha_i) = \mathbb{N}\Gamma_0(\alpha_i)$ za sve $i < l$. Iz toga

$$\mathsf{Recode}(n, b', b) = \sum_{i<l} d_i \cdot b^i = \sum_{i=0}^{l-1} \mathbb{N}\Gamma_0(\alpha_i) \cdot b^i = \langle w_{\sqcup}\ldots\rangle, \tag{4.21}$$

i analogno (4.19). Ključno je bilo da se $\mathbb{N}\Sigma_0$ i $\mathbb{N}\Gamma_0$ podudaraju na svim znakovima $\alpha \in \Sigma_0$.

Primitivna rekurzivnost slijedi iz lema 2.53 i 4.13 te primjera 2.13. □

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 znakovi napisani, i neka je $w := bbc$. Tada je $\langle w\rangle = (223)_3 = 27$ te $\langle w_{\sqcup}\ldots\rangle = (322)_7 = 163$, dakle $\mathsf{Recode}(27, 3, 7) = 163$ i $\mathsf{Recode}(163, 7, 3) = 27$. ◁

Na redu je kodiranje funkcije prijelaza δ_0. Mogli bismo se zabrinuti, 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 dvomjesne koordinatne funkcije (napomena 1.1). Drugo, već imamo kodiranja za ulaze i prva dva izlaza, $\mathbb{N}Q_0 : Q_0 \leftrightarrow [0 \cdot\cdot\, a\rangle$ i $\mathbb{N}\Gamma_0 : \Gamma_0 \leftrightarrow [0 \cdot\cdot\, b\rangle$, 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 funkcije newstate, newsymbol i direction koje preslikavaju $\big(\mathbb{N}Q_0(q), \mathbb{N}\Gamma_0(\gamma)\big)$ 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 $(k, g) \in [0 \cdot\cdot\, a\rangle \times [0 \cdot\cdot\, b\rangle$, ako je $k \neq 1$ označimo $(q', \gamma', d) := \delta_0\big(\mathbb{N}Q_0^{-1}(k), \mathbb{N}\Gamma_0^{-1}(g)\big)$ te definirajmo $\mathsf{newstate}(k, g) := \mathbb{N}Q_0(q')$, $\mathsf{newsymbol}(k, g) := \mathbb{N}\Gamma_0(\gamma')$ i $\mathsf{direction}(k, g) := 1 + d$. Također, za $k = 1$, za svaki $g \in [0 \cdot\cdot\, b\rangle$, dodefinirajmo $\mathsf{newstate}(1, g) := \mathsf{direction}(1, g) := 1$ i $\mathsf{newsymbol}(1, g) := g$.

Koliko god to komplicirana pravila bila, njima definirane funkcije su konačne (sve tri imaju domenu $[0 \cdot\cdot\, a\rangle \times [0 \cdot\cdot\, b\rangle$ s ab elemenata), pa prema korolaru 2.50 za svaku od njih postoji primitivno rekurzivna funkcija koja se s njom podudara na $[0 \cdot\cdot\, a\rangle \times [0 \cdot\cdot\, b\rangle$ (proširenje nulom). Sada je iz definicije tih triju funkcija jasno da vrijedi tvrdnja leme (sjetimo se, $1 = \mathbb{N}Q_0(q_z)$, a $[0 \cdot\cdot\, a\rangle$ i $[0 \cdot\cdot\, b\rangle$ su slike od $\mathbb{N}Q_0$ i $\mathbb{N}\Gamma_0$ redom). □

Primjer 4.21: U primjeru 4.6 smo vidjeli funkciju prijelaza δ_h, definiranu s (4.2).

Ako stanja i znakove kodiramo kao

Q_h	A	B	C	D	E	F
$\mathbb{N}Q_h$	0	3	4	5	1	2

i

Γ_h	a	b	c	d	⊔
$\mathbb{N}\Gamma_h$	1	2	3	4	0

,

tada su funkcije newstate, newsymbol i direction redom zadane tablicama

	0	1	2	3	4	5	⋯
0	1	3	3	2	2	0	⋯
1	1	1	1	1	1	0	⋯
2	2	2	2	2	2	0	⋯
3	4	3	3	2	2	0	⋯
4	2	5	5	2	2	0	⋯
5	2	5	5	0	0	0	⋯
6	0	0	0	0	0	0	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

,

	0	1	2	3	4	5	⋯
0	0	3	4	3	4	0	⋯
1	0	1	2	3	4	0	⋯
2	0	1	2	3	4	0	⋯
3	0	1	2	3	4	0	⋯
4	0	0	0	3	4	0	⋯
5	0	1	2	1	2	0	⋯
6	0	0	0	0	0	0	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

i

	0	1	2	3	4	5	⋯
0	2	2	2	0	0	0	⋯
1	1	1	1	1	1	0	⋯
2	0	0	0	0	0	0	⋯
3	0	2	2	0	0	0	⋯
4	0	0	0	0	0	0	⋯
5	0	0	0	2	2	0	⋯
6	0	0	0	0	0	0	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

. (4.22)

◁

4.1.3. Kodiranje Turing-izračunavanja

Upravo dokazana lema 4.20 donekle je analogna lemi 3.32. Vrijeme je za analogon leme 3.33. Za razliku od tih lema, koje su definirale *uniformne* funkcije što 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

$$\mathsf{State}(\langle w\rangle, n) := \mathbb{N}Q_0(q_n), \quad \mathsf{Position}(\langle w\rangle, n) := p_n, \quad \mathsf{Tape}(\langle w\rangle, n) := \langle t_n\rangle, \tag{4.23}$$

gdje je $\big((q_n, p_n, t_n)\big)_{n\in\mathbb{N}}$ $\mathcal{T}_0$-izračunavanje s w, primitivno su rekurzivne.

Dokaz. Kao i u dokazu leme 3.33, upotrijebit ćemo simultanu primitivnu rekurziju. Za inicijalizaciju trebamo funkcije G_1, G_2 i G_3, koje daju vrijednosti traženih funkcija za $n = 0$. Stanje počinje od q_0 kodiranog nulom, pozicija počinje od nule (lijevi rub), a traka počinje od $w_\sqcup\ldots$, čiji kod se iz $x = \langle w\rangle$ može dobiti primjenom funkcije Recode.

$$\mathsf{G}_1(x) := \mathsf{G}_2(x) := 0, \qquad \mathsf{G}_3(x) := \mathsf{Recode}(x, b', b). \tag{4.24}$$

Dakle, $\mathsf{G}_1 = \mathsf{G}_2 = \mathsf{Z}$ (inicijalna), a G_3 je primitivno rekurzivna po lemi 4.18 i propoziciji 2.21.

Za korak, trebamo funkcije H_1, H_2 i H_3, od kojih svaka prima po pet argumenata: kod riječi s kojom se računa $x = \langle w\rangle$, broj dosad napravljenih koraka izračunavanja n, kod prethodnog stanja $q = \mathbb{N}Q_0(q_n)$, prethodnu poziciju $p = p_n$ i kod prethodne trake $t = \langle t_n\rangle$. Redom trebaju vratiti $\mathbb{N}Q_0(q_{n+1})$, p_{n+1} i $\langle t_{n+1}\rangle$. U tome će nam pomoći funkcije iz leme 4.20, 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. Ovdje ne možemo koristiti sdigit jer se na traci mogu nalaziti i praznine, a i jer se mjesne vrijednosti broje zdesna u zapisu — ali standardni postupak za ekstrakciju znamenke (4.28) funkcionira.

Sada H_1 samo treba vratiti $\mathsf{newstate}(q, g)$. H_2 treba vratiti p pomaknut za $\mathsf{direction}(q, g)$, 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 znamenkom vrijednosti $\mathsf{newsymbol}(q, g)$. To radimo tako da od t oduzmemo trenutnu mjesnu vrijednost te znamenke $g \cdot b^p$ pa dodamo mjesnu vrijednost nove. Dakle, primitivno rekurzivne funkcije

$$\mathsf{H}_1(x, n, q, p, t) := \mathsf{newstate}(q, g), \tag{4.25}$$

$$\mathsf{H}_2(x, n, q, p, t) := \mathsf{pd}\big(p + \mathsf{direction}(q, g)\big), \tag{4.26}$$

$$\mathsf{H}_3(x, n, q, p, t) := t \dot{-} g \cdot b^p + \mathsf{newsymbol}(q, g) \cdot b^p, \tag{4.27}$$

$$\text{uz pokratu } g := t \mathbin{/\!/} b^p \bmod b, \tag{4.28}$$

zadovoljavaju specifikaciju s početka odlomka. Sada su State, $\mathsf{Position}$ i Tape definirane simultanom primitivnom rekurzijom iz G_1, G_2, G_3, H_1, H_2 i H_3, pa su primitivno rekurzivne po propoziciji 3.19. Tvrdnju leme sada dokazujemo matematičkom indukcijom po n, sasvim analogno kao u RAM-modelu.

Za $n = 0$, početno stanje je q_0, pa je $\mathsf{State}(x, 0) = \mathsf{G}_1(x) = \mathsf{Z}(x) = 0 = \mathbb{N}Q_0(q_0)$ doista njegov kod. Početna pozicija je $0 = \mathsf{Z}(x) = \mathsf{G}_2(x) = \mathsf{Position}(x, 0)$. Početna traka je $w_\sqcup \ldots$, čiji kod $\langle w_\sqcup \ldots \rangle$ je prema (4.18) jednak $\mathsf{Recode}(\langle w \rangle, b', b) = \mathsf{G}_3(\langle w \rangle) = \mathsf{Tape}(\langle w \rangle, 0)$.

Sada pretpostavimo da je nakon k koraka, $q := \mathsf{State}(\langle w \rangle, k)$ kod stanja, $p := \mathsf{Position}(\langle w \rangle, k)$ pozicija, a $t := \mathsf{Tape}(\langle w \rangle, k)$ 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 $\mathsf{newstate}(q, g) = \mathsf{direction}(q, g) = 1$ i $\mathsf{newsymbol}(q, g) = g$. Tada H_1 daje q, H_2 daje $\mathsf{pd}(p + 1) = \mathsf{pd}\big(\mathsf{Sc}(p)\big) = p$, a H_3 daje $t \dot{-} g \cdot b^p + g \cdot b^p = t$ (jer je $g = t \mathbin{/\!/} b^p \bmod b \le t \mathbin{/\!/} b^p \le \frac{t}{b^p}$, pa je $g \cdot b^p \le t$, odnosno $\dot{-}$ je zapravo $-$). Dakle $\mathsf{State}(\langle w \rangle, k+1) = \mathsf{State}(\langle w \rangle, k)$, i slično za $\mathsf{Position}$ i Tape, odnosno završna konfiguracija ostaje ista, kao što i treba.

Ako ta konfiguracija nije završna, tad je $q \in [0 \cdot\cdot a\rangle \setminus \{1\}$, pa su vrijednosti $\mathsf{newstate}$, $\mathsf{newsymbol}$ i $\mathsf{direction}$ zadane preko funkcije δ_0 te 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 $\mathsf{direction}(q, g) = d + 1$, pa je

$$\begin{aligned}\mathsf{Position}(\langle w \rangle, k+1) &= \mathsf{H}_2\big(\langle w \rangle, k, \mathsf{State}(\langle w \rangle, k), \mathsf{Position}(\langle w \rangle, k), \mathsf{Tape}(\langle w \rangle, k)\big) = \\ &= \mathsf{H}_2(\langle w \rangle, k, q, p, t) = \mathsf{pd}\big(p + \mathsf{direction}(q, g)\big) = \mathsf{pd}(p + d + 1) = \\ &= \max\{p + d + 1 - 1, 0\} = \max\{\mathsf{Position}(\langle w \rangle, k) + d, 0\}. \end{aligned} \tag{4.29}$$

Slično se dobiju i odgovarajuće vrijednosti za $\mathsf{State}(\langle w \rangle, k+1)$ i $\mathsf{Tape}(\langle w \rangle, k+1)$. □

Pomoću funkcije State^2 možemo prepoznati završnu konfiguraciju — u njoj će vrijednost te funkcije biti jednaka $\mathbb{N}Q_0(q_z) = 1$. Štoviše, ako $\mathcal{T}_0$-izračunavanje s w stane, tada je broj koraka nakon kojeg se to dogodi upravo najmanji broj za koji vrijedi ta jednakost. (Nemamo probleme kao u primjeru 3.34, jer nam je Turingov stroj fiksan 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

$$\mathsf{stop}(\langle w \rangle) :\simeq \text{„broj koraka nakon kojeg } \mathcal{T}_0\text{-izračunavanje s } w \text{ stane”}, \tag{4.30}$$

parcijalno je rekurzivna (s domenom $\mathcal{D}_{\mathsf{stop}} = \mathcal{D}_{\mathbb{N}\varphi_0} = \{\langle w \rangle \mid w \in \mathcal{D}_{\varphi_0}\} =: \langle \mathcal{D}_{\varphi_0} \rangle$).

Dokaz. Tvrdimo da je

$$\mathsf{stop}(x) \simeq \mu n\big(\mathsf{State}(x,n) = 1\big) \tag{4.31}$$

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 — označimo s $n_0 := \mathsf{stop}(x)$ broj koraka nakon kojeg se to dogodi. Prema lemi 4.22 vrijedi $\mathsf{State}(x, n_0) = \mathbb{N}Q_0(q_z) = 1$, ali isto tako za sve $n < n_0$ vrijedi $\mathsf{State}(x,n) \neq 1$ — jer stanje još nije završno, a $\mathbb{N}Q_0$ je injekcija. Dakle n_0 je jednak $\mu n\big(\mathsf{State}(x,n) = 1\big)$.

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 $\mathsf{State}(x,n) = 1$ (opet, jer je $\mathbb{N}Q_0$ injekcija) te izraz $\mu n\big(\mathsf{State}(x,n) = 1\big)$ nema vrijednost, baš kao ni $\mathsf{stop}(x)$. □

Teorem 4.24: Neka je Σ_0 abeceda, $\mathbb{N}\Sigma_0$ njeno kodiranje te 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 $b' := \operatorname{card}\Sigma_0$ i $b := \operatorname{card}\Gamma_0$. 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 te izračunavanje s proizvoljnom riječju), dobijemo (među ostalim) funkcije Recode, Tape i stop, sa svojstvima iz odgovarajućih lema. Tvrdimo da je

$$\mathbb{N}\varphi_0(x) \simeq \mathsf{Recode}\big(\mathsf{Tape}\big(x, \mathsf{stop}(x)\big), b, b'\big), \tag{4.32}$$

pa je $\mathbb{N}\varphi_0$ parcijalno rekurzivna kao kompozicija takvih. Sama parcijalna jednakost (4.32) je zapravo (4.14) 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 vrijednost, a niti desna strana od (4.32) nema vrijednost — $\mathcal{T}_0$ računa φ_0, pa $w \notin \mathcal{D}_{\varphi_0}$ znači da $\mathcal{T}_0$-izračunavanje s w ne stane. Tada $x \notin \mathcal{D}_{\mathsf{stop}}$ prema lemi 4.23, a onda po korolaru 2.7 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, a prema lemi 4.23 to se dogodi nakon $s := \mathsf{stop}(\langle w \rangle) = \mathsf{stop}(x)$ koraka. Prema lemi 4.22, kod trake završne konfiguracije je onda $t := \mathsf{Tape}(x, s)$, što je $\langle \varphi_0(w) \sqcup \ldots \rangle$ jer $\mathcal{T}_0$ računa φ_0. Sada je prema (4.19) $\mathsf{Recode}(t, b, b') = \langle \varphi_0(w) \rangle$, kao što i treba biti. □

4.2. Pretvorba RAM-stroja u Turingov stroj

Dokažimo sada obrat teorema 4.24. Za parcijalno rekurzivne funkcije imamo kompajler u RAM-strojeve, koji možemo iskoristiti i za kompiliranje u Turingove strojeve. U modernom računarstvu, dolaskom nove arhitekture često ne moramo iznova kompilirati izvorni kod. Ako već imamo kod na sličnoj razini (RAM-program), možemo ga neposredno prevesti („transpilirati") u kompilirani 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 $\sigma = \langle \cdots \rangle = \mathbb{N}\Sigma_0^*$ zapisom u pomaknutoj bazi $b = \operatorname{card}\Sigma_0$, jezičnu funkciju φ_0 nad Σ_0 takvu da je $\mathbb{N}\varphi_0 \in \mathcal{C}\mathrm{omp}_1$ 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 („stavi x u $\mathcal{R}_1$") te simulira redom korake P_0-izračunavanja s x. Ako/kad $\mathcal{S}_0$ uđe u završnu konfiguraciju c, $\mathcal{T}_0$ će sadržaj $y := c(\mathcal{R}_0)$ „dekodirati" u riječ $v := \varphi_0(w)$, obrisati ostatke izračunavanja s trake i premjestiti v na početak. 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: $\mathbb{N}\varphi_0 = \sigma \circ \varphi_0 \circ \sigma^{-1}$ znači $\varphi_0 = \sigma^{-1} \circ \mathbb{N}\varphi_0 \circ \sigma$.

Stroj $\mathcal{T}_0$ ćemo konstruirati u pet dijelova (*fragmenata*), koji provode razne faze opisanog postupka simulacije stroja $\mathcal{S}_0$. Da bismo ga opisali, potrebno je precizirati njegova stanja, radnu abecedu Γ te prijelaze.

Stanja ćemo uvoditi implicitno i postepeno, specificiranjem funkcije 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.

U radnoj abecedi se očito moraju nalaziti svi znakovi iz Σ_0, kodirani po $\mathbb{N}\Sigma_0$ brojevima iz $[1 \cdot\cdot b]$. Za svaki $t \in [1 \cdot\cdot b]$ označimo s $\alpha_t := \mathbb{N}\Sigma_0{}^{-1}(t)$ odgovarajuću „znamenku".

Za demarkaciju upotrijebljenog dijela trake koristimo graničnik $\$$. Separator $\#$ odvaja dio za ulaz od dijela za računanje (simulaciju). Traka će pri početku rada izgledati otprilike kao $\$\sqcup^* v \# u \sqcup \ldots$, a pri njegovu kraju kao $\$ v \sqcup^* u \$ \sqcup \ldots$, gdje je $v \in \Sigma_0^*$ dio za ulaz odnosno izlaz, a $u \in (B^m)^*$ dio za reprezentaciju konfiguracije od $\mathcal{S}_0$. Dakle $\Gamma := \Sigma_0 \uplus \{\$, \#\} \uplus B^m$, gdje je B^m skup koji ćemo kasnije definirati — taj skup sadržavat će i prazninu $\sqcup$.

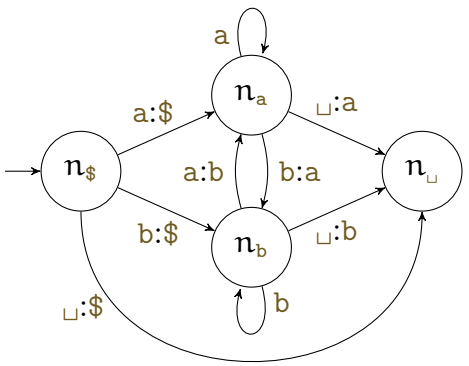
Cilj prve faze je pomaknuti riječ za jednu ćeliju desno, stavivši graničnik na početak.

Lema 4.25: Postoji fragment Turingova stroja koji prevodi početnu konfiguraciju $\underset{n_\$}{w} = (n_\$, 0, w \sqcup \ldots)$ u konfiguraciju $\underset{n_\sqcup}{\$ w \sqcup} = (n_\sqcup, |w| + 1, \$ w \sqcup \ldots)$.

Dokaz. Obično se takav pomak radi s desnog kraja, jer inače moramo izvan trake pamtiti znakove koje prenosimo. No sjetimo se uvoda: za Turingov stroj pomaci su teški a pamćenje znakova lagano — jer je Γ konačan, znakove možemo pamtiti u stanjima stroja. Imat ćemo po jedno stanje n_α za prijenos svakog znaka $\alpha \in \Sigma_0 \uplus \{\$, \sqcup\}$.

Tada, ako u stanju n_α čitamo znak β, zamjenjujemo ga s α i pamtimo u stanju n_β ($\delta 1$). To objašnjava i zašto je početno stanje $n_\$$. Nakon što pročitamo i pomaknemo čitavu riječ, nađemo se u stanju $n_\sqcup$ na prvoj praznini nakon nje. □

Primjer 4.26: Za $\Sigma_0 := \{a, b\}$, prvi fragment je prikazan dijagramom



(4.33)

Za ulaz aab izračunavanje počinje s $\underset{n_\$}{\mathtt{aab}} \rightsquigarrow \underset{n_a}{\$\mathtt{ab}} \rightsquigarrow \underset{n_a}{\$\mathtt{ab}} \rightsquigarrow \underset{n_b}{\$\mathtt{aa}\sqcup} \rightsquigarrow \underset{n_\sqcup}{\$\mathtt{aab}\sqcup}$. ◁

$$\delta(n_\alpha, \beta) := (n_\beta, \alpha, 1), \text{ za sve } \alpha \in \Sigma_0 \uplus \{\$\},\ \beta \in \Sigma_0 \uplus \{\sqcup\} \tag{δ1}$$

Sljedeći korak je konstruirati početnu konfiguraciju od $\mathcal{S}_0$, desno od separatora # na poziciji $|w|+1$. Znamo da to mora biti element od $(B^m)^*$, pa pokušajmo s unarnim kodiranjem: cilj nam je na kraj riječi staviti # te $r_1^{\langle w\rangle}$ za jedan konkretni element $r_1 \in B^m$.

Lema 4.27: Postoji fragment Turingova stroja koji prevodi konfiguraciju $\underset{n_\sqcup}{\$w\sqcup} = (n_\sqcup, |w|+1, \$w\sqcup\ldots)$ u konfiguraciju $\underset{p_0}{\$\sqcup^{|w|}\#r_1^{\langle w\rangle}} = (p_0, |w|+2, \$\sqcup^{|w|}\#r_1^{\langle w\rangle}\sqcup\ldots)$.

$$\delta(n_\sqcup, \sqcup) := (q_0, \#, -1) \tag{δ2a}$$

$$\delta(q_0, \alpha_t) := (q_1, \alpha_{t-1}, 1), \text{ za sve } t \in [2 \cdot\cdot\, b] \tag{δ2b}$$

$$\delta(q_1, \alpha) := (q_1, \alpha, 1), \text{ za sve } \alpha \in \Sigma_0 \uplus \{\#, r_1\} \tag{δ2c}$$

$$\delta(q_1, \sqcup) := (q_0, r_1, -1) \tag{δ2d}$$

$$\delta(q_0, \gamma) := (q_0, \gamma, -1), \text{ za } \gamma \in \{\#, r_1\} \tag{δ2e}$$

$$\delta(q_0, \alpha_1) := (q_0, \alpha_b, -1) \tag{δ2f}$$

$$\delta(q_0, \gamma) := (q_2, \gamma, 1), \text{ za } \gamma \in \{\$, \sqcup\} \tag{δ2g}$$

$$\delta(q_2, \alpha_b) := (q_1, \sqcup, 1) \tag{δ2h}$$

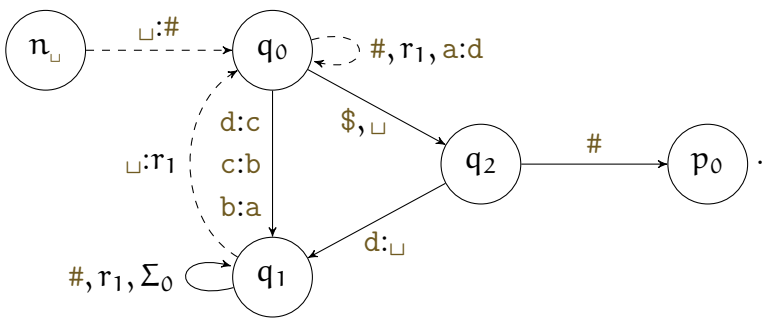
$$\delta(q_2, \#) := (p_0, \#, 1) \tag{δ2i}$$

Dokaz. Prvo postavimo separator na trenutnu poziciju (δ2a). Zatim broj $x := \langle w\rangle$ čitamo u pomaknutoj bazi b pomoću znamenaka $\alpha_t, t \in [1 \cdot\cdot\, b]$ te pišemo unarno (u pomaknutoj bazi 1) pomoću znamenke r_1. To se može riješiti tako da „dekrementiramo" w, za svaki uspješni dekrement dodajući po jedan r_1 na kraj.

Zapis u pomaknutoj bazi dekrementiramo slično kao i zapis u običnoj bazi. Ako zadnji znak nije α_1, smanjimo njegov kod za 1 (δ2b) i odemo desno (δ2c) do prvog razmaka, koji zamijenimo s r_1 (δ2d) i vraćamo se ponovo lijevo (δ2e). Ako naiđemo na α_1, zamijenimo ga „najvećom znamenkom" α_b i nastavljamo smanjivati dalje lijevo (δ2f). Ako smanjujući dođemo do lijevog ruba broja (δ2g), prvu znamenku mu obrišemo (δ2h). Na kraju odemo desno iza separatora, u stanje p_0 (δ2i).

Za dokaz da to stvarno dovede do željene konfiguracije dovoljno je primijetiti da se nakon svakog prolaza lijevo-desno nađemo ponovo u stanju q_0, s pozicijom pri desnom kraju trake oblika $\$\sqcup^{|w|-|w'|}w'\#r_1^{\langle w\rangle-\langle w'\rangle}$. 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, a na kraju onda moramo imati $\$\sqcup^{|w|}\#r_1^{\langle w\rangle}$, jer je tada $w' = \varepsilon$ s kodom $\langle\varepsilon\rangle = 0$. □

Primjer 4.28: Za $\Sigma_0 := \{a, b, c, d\}$, drugi fragment je prikazan dijagramom



(4.34)

Za riječ aab imamo šetnju kroz konfiguracije (pišemo samo neke od njih):

$$
\begin{aligned}
&\underset{n_\sqcup}{\$aab\sqcup} \rightsquigarrow \underset{q_0}{\$aab\#} \rightsquigarrow \underset{q_1}{\$aaa\#} \rightsquigarrow \underset{q_1}{\$aaa\#\sqcup} \rightsquigarrow \underset{q_0}{\$aaa\#r_1} \rightsquigarrow^4 \underset{q_0}{\$ddd\#r_1} \rightsquigarrow \underset{q_2}{\$ddd\#r_1} \rightsquigarrow \underset{q_1}{\$\sqcup dd\#r_1} \rightsquigarrow^4 \\
&\rightsquigarrow^4 \underset{q_1}{\$\sqcup dd\#r_1\sqcup} \rightsquigarrow \underset{q_0}{\$\sqcup dd\#r_1r_1} \rightsquigarrow^2 \underset{q_0}{\$\sqcup dd\#r_1^2} \rightsquigarrow \underset{q_1}{\$\sqcup dc\#r_1^2} \rightsquigarrow^3 \underset{q_1}{\$\sqcup dc\#r_1^2\sqcup} \rightsquigarrow^4 \underset{q_0}{\$\sqcup dc\#r_1^3} \rightsquigarrow^* \underset{q_0}{\$\sqcup db\#r_1^4} \rightsquigarrow^* \\
&\rightsquigarrow^* \underset{q_0}{\$\sqcup da\#r_1^5} \rightsquigarrow^* \underset{q_0}{\$\sqcup cd\#r_1^6} \rightsquigarrow^* \underset{q_0}{\$\sqcup ca\#r_1^9} \rightsquigarrow^* \underset{q_0}{\$\sqcup ba\#r_1^{13}} \rightsquigarrow^* \underset{q_0}{\$\sqcup aa\#r_1^{17}} \rightsquigarrow^* \underset{q_0}{\$\sqcup\sqcup d\#r_1^{18}} \rightsquigarrow^* \underset{q_0}{\$\sqcup\sqcup a\#r_1^{21}} \rightsquigarrow \\
&\quad\rightsquigarrow \underset{q_0}{\$\sqcup\sqcup d\#r_1^{21}} \rightsquigarrow \underset{q_2}{\$\sqcup\sqcup d\#r_1^{21}} \rightsquigarrow \underset{q_1}{\$\sqcup\sqcup\sqcup\#r_1^{21}} \rightsquigarrow^* \underset{q_1}{\$\sqcup\sqcup\sqcup\#r_1^{21}\sqcup} \rightsquigarrow^* \underset{q_0}{\$\sqcup\sqcup\sqcup\#r_1^{22}} \rightsquigarrow \underset{q_2}{\$\sqcup\sqcup\sqcup\#r_1^{22}} \rightsquigarrow \underset{p_0}{\$\sqcup\sqcup\sqcup\#r_1^{22}},
\end{aligned}
$$

što je točno kako treba biti jer je $b = 4$, pa je $\langle aab\rangle = (112)_4 = 22$. „Invarijanta petlje" zadovoljena je u svim istaknutim konfiguracijama gdje se q_0 nalazi ispod zadnjeg znaka prije separatora: npr. za $\$\sqcup cd\#r_1^6$ je $w' = cd$, čiji je kod $\langle cd\rangle = (34)_4 = 16$, a $\langle w\rangle - \langle w'\rangle = 22 - 16 = 6$. ◁

4.2.1. Registri kao tragovi trake

Vrijeme je da opišemo kako ćemo točno reprezentirati konfiguraciju RAM-stroja $\mathcal{S}_0$ 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 prvih m registara.

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"

$$
B^m := \prod_{i<m} \{\circ, \bullet\} = \left\{ \begin{bmatrix} \beta_0 \\ \vdots \\ \beta_{m-1} \end{bmatrix} \,\middle|\, (\forall i < m)(\beta_i \in \{\circ, \bullet\}) \right\}, \tag{4.35}
$$

tako da se dio trake desno od separatora sastoji od m „redaka" (*tragova*), u svakom od kojih piše $\bullet^t\circ\cdots$, gdje je t sadržaj odgovarajućeg registra. U tom kontekstu, B^m je „prostor stupaca" sa svim (2^m njih) stupcima znakova $\bullet$ i $\circ$ visine m. Recimo,

$$
B^4 := \left\{ \begin{matrix}\circ\\\circ\\\circ\\\circ\end{matrix}, \begin{matrix}\bullet\\\circ\\\circ\\\circ\end{matrix}, \begin{matrix}\circ\\\bullet\\\circ\\\circ\end{matrix}, \begin{matrix}\bullet\\\bullet\\\circ\\\circ\end{matrix}, \begin{matrix}\circ\\\circ\\\bullet\\\circ\end{matrix}, \begin{matrix}\bullet\\\circ\\\bullet\\\circ\end{matrix}, \begin{matrix}\circ\\\bullet\\\bullet\\\circ\end{matrix}, \begin{matrix}\bullet\\\bullet\\\bullet\\\circ\end{matrix}, \begin{matrix}\circ\\\circ\\\circ\\\bullet\end{matrix}, \begin{matrix}\bullet\\\circ\\\circ\\\bullet\end{matrix}, \begin{matrix}\circ\\\bullet\\\circ\\\bullet\end{matrix}, \begin{matrix}\bullet\\\bullet\\\circ\\\bullet\end{matrix}, \begin{matrix}\circ\\\circ\\\bullet\\\bullet\end{matrix}, \begin{matrix}\bullet\\\circ\\\bullet\\\bullet\end{matrix}, \begin{matrix}\circ\\\bullet\\\bullet\\\bullet\end{matrix}, \begin{matrix}\bullet\\\bullet\\\bullet\\\bullet\end{matrix} \right\}. \tag{4.36}
$$

Konkretno, ako (za $m = 4$) u nekom trenutku imamo konfiguraciju $(1, 4, 0, 5, 0, 0, \ldots)$, uz $|w| = 7$, na traci će biti

$$
\$\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\#\begin{matrix}\bullet\circ\circ\circ\circ\circ\circ\circ\circ\circ\\\bullet\bullet\bullet\bullet\circ\circ\circ\circ\circ\circ\\\circ\circ\circ\circ\circ\circ\circ\circ\circ\circ\\\bullet\bullet\bullet\bullet\circ\circ\circ\circ\circ\circ\end{matrix}\cdots, \tag{4.37}
$$

iz čega vidimo dvije stvari. Prvo, $\sqcup$ mora biti prvi element naveden u B^m, koji ima sve kružiće prazne. Tako postižemo 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,

$$
\sqcup := \begin{matrix}\circ\\\circ\\\circ\\\vdots\\\circ\end{matrix}, \qquad r_1 := \begin{matrix}\circ\\\bullet\\\circ\\\vdots\\\circ\end{matrix}. \tag{4.38}
$$

Primjer 4.29: Nad abecedom $\Sigma := \{a, b\}$ kodiranom s $\{a \mapsto 1, b \mapsto 2\}$ pogledajmo jezičnu funkciju „dopisivanje a zdesna", $\varphi_a(w) := wa$. Baza je $b = 2$, pa je prateća funkcija $\mathbb{N}\varphi_a(\langle w\rangle) = \langle wa\rangle = \langle w\rangle \mathbin{\widehat{2}} \langle a\rangle = \langle w\rangle \cdot 2^{\mathsf{slh}(\langle a\rangle,2)} + \langle a\rangle = 2^{|a|} \cdot \langle w\rangle + 1$, odnosno $\mathbb{N}\varphi_a(x) = 2x + 1$. Očito je $\mathbb{N}\varphi_a \in \mathcal{C}\mathrm{omp}_1$: računa je jednostavni RAM-program

$$P_a := \begin{bmatrix} 0.\ \textsc{inc}\ \mathcal{R}_0 \\ 1.\ \textsc{dec}\ \mathcal{R}_1, 4 \\ 2.\ \textsc{inc}\ \mathcal{R}_0 \\ 3.\ \textsc{go to}\ 0 \end{bmatrix} \tag{4.39}$$

širine $m = 2$. Za ulaznu riječ ab je $\langle ab\rangle = (12)_2 = 4$, pa je konfiguracija Turingova stroja kroz faze (početna, nakon prve faze i nakon druge faze)

$$\underset{n_\$}{ab} \rightsquigarrow^* \$\underset{n_\sqcup}{ab\sqcup} \rightsquigarrow^* \$\underset{p_0}{\circ\circ\circ\#\bullet\bullet\bullet\bullet\circ\circ\circ\circ}. \quad \triangleleft \tag{4.40}$$

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, za P_0 duljine $n := n_{P_0}$ imat ćemo stanja $p_i, i \in [0 \cdot\cdot\, n]$, pri čemu će ulazak u stanje p_n značiti završetak P_0-izračunavanja.

Definicija 4.30: Neka je P^1 RAM-algoritam širine m te $c = (r_0, r_1, \ldots, r_{m-1}, 0, 0, \ldots, pc)$ konfiguracija RAM-stroja s programom P. Tada je m-*reprezentacija* od c konfiguracija Turingova stroja sa stanjem p_{pc}, trakom oblika $\$\sqcup^k\#v\sqcup\ldots$ i pozicijom većom od k — pri čemu je v riječ nad B^m u čijem i-tom tragu piše $\bullet^{r_i}\circ^{|v|-r_i}$, za sve $i \in [0 \cdot\cdot\, m\rangle$. $\triangleleft$

Recimo, (4.37) je primjer trake 4-reprezentacije konfiguracije $(1, 4, 0, 5, 0, 0, \ldots)$. Također, zadnja konfiguracija u (4.40) je 2-reprezentacija početne konfiguracije RAM-stroja s programom P_a iz (4.39), s ulazom ab.

Ipak, stanja p_i nisu dovoljna za izvršavanje RAM-instrukcija: za svaki $i < n$ imamo još jedno stanje s_i, koje tome služi. Stanje p_i je „pripremno stanje" za s_i, s jedinim zadatkom fiksiranja pozicije odmah nakon znaka #.

Da bismo implementirali prijelaze iz stanja s_i, moramo naučiti „adresirati bitove" na traci m-reprezentacije. Svaki element od B^m vidimo kao m-bitnu riječ, koristeći za promjenu pojedinog bita istu tehniku kao „pravi" procesor: čitavu procesorsku riječ zamijenimo drugom, koja se podudara s njom u svim bitovima osim onog koji želimo promijeniti. Primjerice, ako na traci (4.37) želimo „dekrementirati $\mathcal{R}_1$", zamijenit ćemo njen trinaesti znak $(\circ\bullet\circ\bullet)^\top$ znakom $(\circ\circ\circ\bullet)^\top$, devetim na popisu (4.36).

Definicija 4.31: Neka je $j \in [0 \cdot\cdot\, m\rangle$, 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 $i \in [0 \cdot\cdot\, m\rangle \setminus \{j\}$.

U dijagramima, na strelici od q prema q' pišemo $\beta : \beta'@j$, ili $\beta@j$ ako je $\beta' = \beta$. $\triangleleft$

4.2.2. Simulacija RAM-izračunavanja

Lema 4.32: Neka je $P_0 =: \left[\ i.\ I_t\ \right]_{t<n}$ (širine m). Tada za svaki $i < n$ postoji fragment Turingova stroja, koji prevodi svaku m-reprezentaciju svake $\mathcal{S}_0$-konfiguracije c kojoj je $c(\textsc{pc}) = i$, u neku m-reprezentaciju $\mathcal{S}_0$-konfiguracije d, gdje $c \rightsquigarrow d$ po programu P_0.

Dokaz. Traženi fragment počinje s radom u stanju p_i na poziciji većoj ili jednakoj poziciji separatora k, a treba završiti u stanju $p_{i'}$ za $i' := d(\text{PC})$ na poziciji s istim svojstvom. Za početak, dok god čitamo znak iz B^m, pomičemo se lijevo (δ3a). Kad pročitamo #, pomaknemo se desno (δ3b).

$$\delta(p_i, \gamma) := (p_i, \gamma, -1), \text{ za sve } \gamma \in B^m \tag{δ3a}$$

$$\delta(p_i, \#) := (s_i, \#, 1) \tag{δ3b}$$

Tako dođemo u stanje s_i, na poziciju $k+1$. Zbog $c(\text{PC}) = i < n = n_P$ konfiguracija c sigurno nije završna, pa postoji instrukcija I_i. Dalji postupak ovisi o njenom tipu.

Ako je I_i tipa INC, recimo $i.$ INC $\mathcal{R}_j$, trebamo otići do kraja znakova • u tragu j (δ3c) i dopisati još jedan • tamo, a zatim se pripremiti za sljedeću instrukciju, onu rednog broja $i+1$ (δ3d).

$$\delta_{(j)}(s_i, \bullet) := (s_i, \bullet, 1) \tag{δ3c}$$

$$\delta_{(j)}(s_i, \circ) := (p_{i+1}, \bullet, -1) \tag{δ3d}$$

Ako je I_i tipa DEC, recimo $i.$ DEC $\mathcal{R}_j, l$, trebamo kao i za INC doći do kraja znakova • (δ3e), a kada pročitamo ∘, pomaknuti se lijevo u novo „stanje odluke" t_i (δ3f). Ako u tom stanju čitamo #, znači da je $c(\mathcal{R}_j) = 0$, pa prelazimo na instrukciju rednog broja l (δ3g). Ako ne, tada u tragu j zamijenimo • s ∘ i prelazimo na sljedeću instrukciju (δ3h).

$$\delta_{(j)}(s_i, \bullet) := (s_i, \bullet, 1) \tag{δ3e}$$

$$\delta_{(j)}(s_i, \circ) := (t_i, \circ, -1) \tag{δ3f}$$

$$\delta(t_i, \#) := (p_l, \#, 1) \tag{δ3g}$$

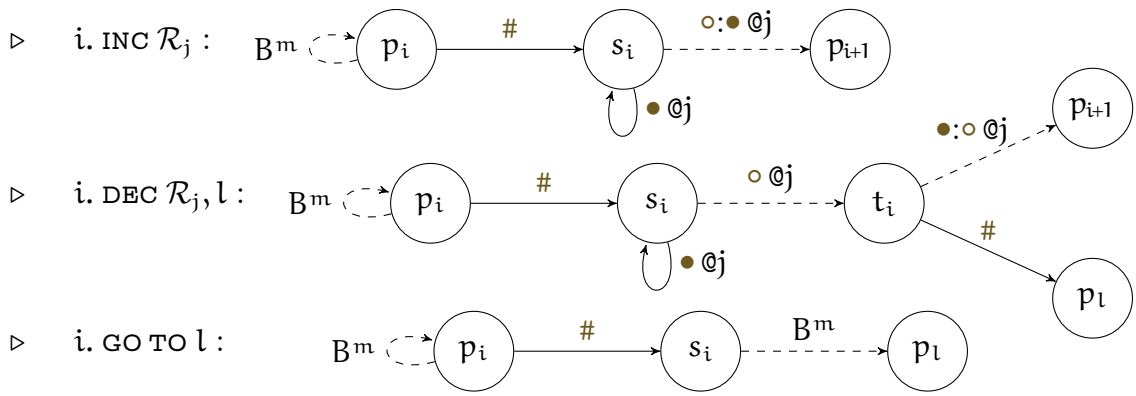
$$\delta_{(j)}(t_i, \bullet) := (p_{i+1}, \circ, -1) \tag{δ3h}$$

Ako je I_i tipa GO TO, recimo $i.$ GO TO l, samo odemo na pripremu za instrukciju I_l (δ3i).

$$\delta(s_i, \gamma) := (p_l, \gamma, -1), \text{ za sve } \gamma \in B^m \tag{δ3i}$$

Lako je vidjeti da je pozicija ulaskom u $p_{i'}$ doista veća ili jednaka k — recimo, primjenom (δ3h) pozicija će biti $k + c(\mathcal{R}_j) - 1 \geq k$, jer je u tom slučaju $c(\mathcal{R}_j) > 0$ — dakle, postignuta konfiguracija jest m-reprezentacija od d. □

Tipove fragmenata iz dokaza možemo prikazati dijagramima:



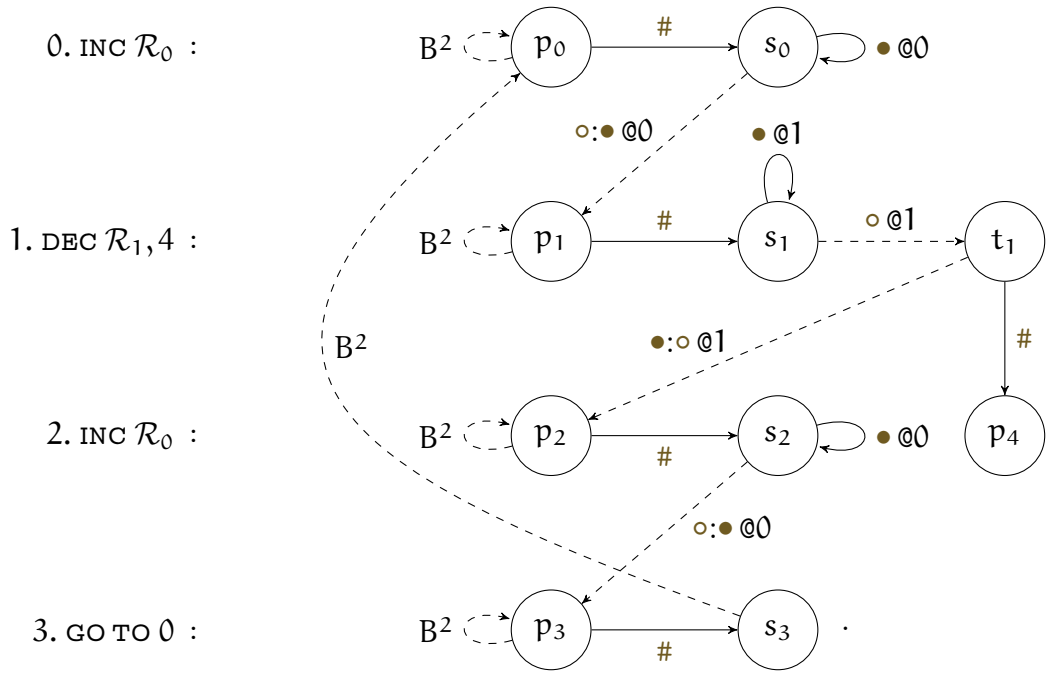
Propozicija 4.33: Postoji fragment Turingova stroja koji za sve $x \in \mathbb{N}$, počevši od bilo koje m-reprezentacije početne $\mathcal{S}_0$-konfiguracije s ulazom x, dostigne neku m-reprezentaciju završne $\mathcal{S}_0$-konfiguracije P_0-izračunavanja s x ako je $x \in \mathcal{D}_{\mathbb{N}\varphi_0}$, a inače nikada ne stigne u stanje p_n.

Dokaz. Za svaki $x \in \mathbb{N}$, označimo sa $(c_l)_{l\in\mathbb{N}}$ P_0-izračunavanje s x i indukcijom po l dokažimo da dosad sastavljeni fragment Turingova stroja dostigne m-reprezentaciju od c_l. Baza je očita, jer je c_0 upravo početna konfiguracija. Korak (ako c_l nije završna) je primjena leme 4.32. Dakle, ako P_0-izračunavanje stane u l_0 koraka, fragment će postići m-reprezentaciju završne konfiguracije c_{l_0}, došavši u stanje p_n.

No vrijedi i obrat: pri simulaciji jednog RAM-prijelaza $c \rightsquigarrow d$, 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 p_n je da doista neka RAM-konfiguracija u izračunavanju preslika PC u n, što znači da P-izračunavanje s x stane. □

Primjer 4.34: U primjeru 4.29 smo vidjeli RAM-program P_a (4.39) duljine 4 i širine 2, koji računa prateću funkciju dopisivanja a na kraj riječi, $\mathbb{N}\varphi_a(x) := 2x + 1$.

Treći fragment Turingova stroja koji odgovara P_a prikazan je dijagramom



(4.41)

P_a-izračunavanje s $\langle ab \rangle = 4$ je

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	⋯
$c_i(\mathcal{R}_0)$	0	0	1	2	2	2	3	4	4	4	5	6	6	6	7	8	8	8	9	9
$c_i(\mathcal{R}_1)$	4	3	3	3	3	2	2	2	2	1	1	1	1	0	0	0	0	0	0	0
$c_i(\mathcal{R}_{2..})$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$c_i(\text{PC})$	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	4	5	5

(4.42)

s izlaznim podatkom $9 = 2 \cdot 4 + 1$, i odgovara mu šetnja kroz 2-reprezentacije

$$
\begin{aligned}
&\$\circ\circ\#\bullet\bullet\bullet\bullet\circ\circ\circ\circ \underset{p_0}{\rightsquigarrow^*} \$\circ\circ\#\bullet\bullet\bullet\bullet\circ\circ\circ \underset{p_1}{\rightsquigarrow^*} \$\circ\circ\#\bullet\bullet\bullet\circ\circ \underset{p_2}{\rightsquigarrow^*} \$\circ\circ\#\bullet\bullet\bullet\circ \underset{p_3}{\rightsquigarrow^*} \$\circ\circ\#\bullet\bullet\bullet \underset{p_0}{\rightsquigarrow^*} \$\circ\circ\#\bullet\bullet\bullet\circ \underset{p_1}{\rightsquigarrow^*} \$\circ\circ\#\bullet\bullet\bullet\circ\circ \underset{p_2}{\rightsquigarrow^*} \\
&\underset{p_2}{\rightsquigarrow^*} \$\circ\circ\#\bullet\bullet\bullet\bullet\bullet\circ \underset{p_3}{\rightsquigarrow^*} \$\circ\circ\#\bullet\bullet\bullet\bullet\circ\circ \underset{p_0}{\rightsquigarrow^*} \$\circ\circ\#\bullet\bullet\bullet\circ\circ\circ \underset{p_1}{\rightsquigarrow^*} \$\circ\circ\#\bullet\bullet\circ\circ\circ\circ \underset{p_2}{\rightsquigarrow^*} \$\circ\circ\#\bullet\circ\circ\circ\circ\circ \underset{p_3}{\rightsquigarrow^*} \\
&\underset{p_0}{\rightsquigarrow^*} \$\circ\circ\#\bullet\bullet\bullet\bullet\bullet\circ\circ \underset{p_1}{\rightsquigarrow^*} \$\circ\circ\#\bullet\bullet\bullet\bullet\bullet\circ \underset{p_2}{\rightsquigarrow^*} \$\circ\circ\#\bullet\bullet\bullet\bullet\bullet \underset{p_3}{\rightsquigarrow^*} \$\circ\circ\#\bullet\circ\circ\circ\circ\circ\circ \underset{p_0}{\rightsquigarrow^*} \$\circ\circ\#\bullet\bullet\bullet\bullet\bullet\circ \underset{p_1}{\rightsquigarrow^*} \$\circ\circ\#\bullet\bullet\bullet\bullet\bullet\bullet \underset{p_2}{\rightsquigarrow^*} \\
&\underset{p_3}{\rightsquigarrow^*} \$\circ\circ\#\bullet\bullet\bullet\bullet\bullet\bullet\circ \underset{p_0}{\rightsquigarrow^*} \$\circ\circ\#\bullet\bullet\bullet\bullet\bullet\bullet \underset{p_1}{\rightsquigarrow^*} \$\circ\circ\#\bullet\bullet\bullet\bullet\bullet\bullet\circ \underset{p_2}{\rightsquigarrow^*} \$\circ\circ\#\bullet\bullet\bullet\bullet\bullet\bullet\bullet \underset{p_3}{\rightsquigarrow^*} \$\circ\circ\#\bullet\circ\circ\circ\circ\circ\circ\circ \underset{p_0}{\rightsquigarrow^*} \$\circ\circ\#\bullet\bullet\bullet\bullet\bullet\bullet\circ \underset{p_1}{\rightsquigarrow^*} \$\circ\circ\#\bullet\bullet\bullet\bullet\bullet\bullet\bullet \underset{p_4}{\rightsquigarrow^*}, \qquad (4.43)
\end{aligned}
$$

koja predstavlja prirodni nastavak one u (4.40). ◁

Još je potrebno dekodirati y (ako postoji) u riječ $v := \varphi_0(w)$ i postaviti je na početak trake. Postupak je sličan onome u drugoj fazi, samo u obrnutom smjeru. Imamo broj y zapisan unarno (pomaknuta baza 1) u tragu 0, a 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, to nam upravo omogućuje definicija 4.31.

Drugo i važnije, kod dekrementiranja smo znali da riječ neće postati dulja, pa smo prazninama skraćivali zapis broja koliko je već potrebno, dok ga nismo dekrementirali sasvim do nule. Kod inkrementiranja nemamo unaprijed određenu gornju granicu za duljinu izlazne riječi $v := \varphi_0(w)$, pa ćemo morati obrisati separator # i koristiti čitavu traku za njenu konstrukciju. Ipak, već upotrijebljeni dio trake će biti dovoljan, jer sadrži barem barem $y = \langle \varphi_0(w) \rangle$ znakova $\bullet$ u tragu 0.

Lema 4.35: Za svaku riječ w vrijedi $|w| \leq \langle w \rangle$.

Dokaz. Označimo s b broj znakova abecede Σ nad kojom je riječ w. Svaki pribrojnik u sumi (4.9) je pozitivan, jer je svaka znamenka $\mathbb{N}\Sigma(\alpha_i) \in [1 \cdot\cdot\, b]$ te je $b > 0$ zbog $\Sigma \neq \varnothing$. 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 $|w|$.

Za $b = 1$ vrijedi jednakost (jer je jedina moguća znamenka 1). □

Štoviše, lema 4.35 pokazuje da riječ možemo dekodirati korak po korak: u svakom trenutku će nam u tragu 0 ostati y' znakova $\bullet$, a desno od početnog graničnika će biti napisana riječ u' čiji je kod $\langle u' \rangle = y - y'$, pa će sigurno stati u prostor od $k + y - y' \geq y - y'$ ćelija oslobođen skidanjem $y - y'$ znakova $\bullet$.

4.2.3. Konstrukcija izlazne riječi

Lema 4.36: Postoji fragment Turingova stroja koji prevodi bilo koju m-reprezentaciju završne konfiguracije $(p_n, k, \$\sqcup^{|w|}\#t\sqcup\ldots)$, gdje je $k > |w|$ i t u tragu 0 ima oblik $\bullet^y\circ^{|t|-y}$, u konfiguraciju $(l_\sqcup, |v|, \$v\sqcup\ldots)$, gdje je $\langle v \rangle = y$.

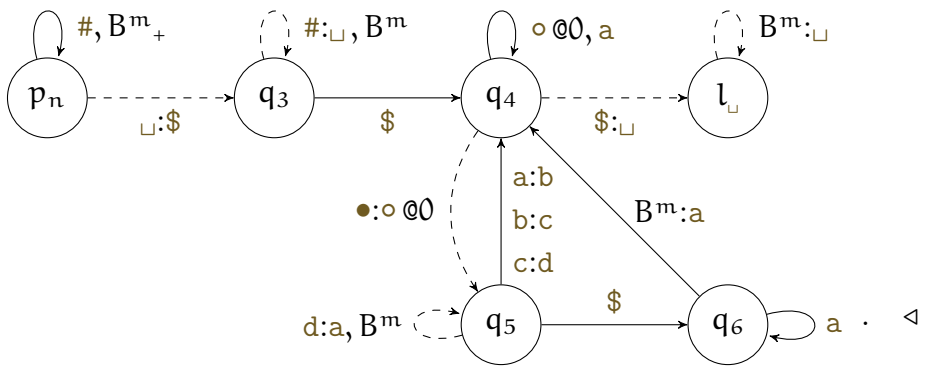
Dokaz. Prvo odemo do kraja upotrijebljenog dijela trake (δ4a) i tamo zapišemo krajnji graničnik (δ4b). Zatim odemo do početka trake (δ4c), brišući pritom separator (δ4d). Tada se okrenemo (δ4e) i idemo desno kroz znakove s $\circ$ u tragu 0 (δ4f) — dok ne dođemo do prvog $\bullet$,

koji obrišemo (δ4g) i prenesemo lijevo (δ4h) do prvog znaka iz Σ_0. Ako je taj znak „najveći", zamjenjujemo ga „najmanjim" (δ4i) i nastavljamo ići lijevo, sve dok ne dođemo do nekog znaka koji možemo „inkrementirati" (δ4j), ili do početnog graničnika (δ4k). U ovom posljednjem slučaju, odemo na kraj riječi koja se mora sastojati samo od znakova α_1 (δ4l) i dopišemo još jedan na kraj (δ4m). Odemo po novi $\bullet$ iz traga $0\ldots$ kada ih sve potrošimo odlazimo lijevo brišući krajnji graničnik i sve ostatke izračunavanja s trake (δ4n), dok ne udarimo u zadnji znak od v (ili početni graničnik ako je $v = \varepsilon$).

Argument zašto to funkcionira je praktički isti kao u dokazu leme 4.27: nakon svakog „prolaza kroz petlju" imamo traku oblika $\$ut\$$ pri čemu t ima $\langle v\rangle - \langle u\rangle$ znakova $\bullet$ u tragu 0. Na početku je $u = \varepsilon$ pa je to istina ($\langle v\rangle - \langle\varepsilon\rangle = y - 0 = y$), svakim prolazom ostaje istina jer se $\langle u\rangle$ povećava za 1 dok se broj znakova $\bullet$ u tragu 0 smanjuje za 1 — pa na izlasku iz petlje mora biti $\langle v\rangle - \langle u\rangle = 0$, odnosno zbog injektivnosti kodiranja riječi $u = v$. Još je samo preostalo isprazniti i ostale tragove iz t te obrisati zadnji $\$$. □

$$\delta(p_n, \gamma) := (p_n, \gamma, 1), \text{ za sve } \gamma \in \{\#\} \cup B^m \setminus \{\sqcup\} \tag{δ4a}$$
$$\delta(p_n, \sqcup) := (q_3, \$, -1) \tag{δ4b}$$
$$\delta(q_3, \gamma) := (q_3, \gamma, -1), \text{ za sve } \gamma \in B^m \tag{δ4c}$$
$$\delta(q_3, \#) := (q_3, \sqcup, -1) \tag{δ4d}$$
$$\delta(q_3, \$) := (q_4, \$, 1) \tag{δ4e}$$
$$\delta_{(0)}(q_4, \circ) := (q_4, \circ, 1) \tag{δ4f}$$
$$\delta_{(0)}(q_4, \bullet) := (q_5, \circ, -1) \tag{δ4g}$$
$$\delta(q_5, \gamma) := (q_5, \gamma, -1), \text{ za sve } \gamma \in B^m \tag{δ4h}$$
$$\delta(q_5, \alpha_b) := (q_5, \alpha_1, -1) \tag{δ4i}$$
$$\delta(q_5, \alpha_t) := (q_4, \alpha_{t+1}, 1), \text{ za sve } t \in [1 \cdot\cdot\, b\rangle \tag{δ4j}$$
$$\delta(q_5, \$) := (q_6, \$, 1) \tag{δ4k}$$
$$\delta(q_6, \alpha_1) := (q_6, \alpha_1, 1) \tag{δ4l}$$
$$\delta(q_6, \gamma) := \delta(q_4, \alpha_1) := (q_4, \alpha_1, 1), \text{ za sve } \gamma \in B^m \tag{δ4m}$$
$$\delta(q_4, \$) := \delta(l_\sqcup, \gamma) := (l_\sqcup, \sqcup, -1), \text{ za sve } \gamma \in B^m \tag{δ4n}$$

Primjer 4.37: Za $\Sigma_0 := \{a, b, c, d\}$, četvrti fragment možemo prikazati dijagramom



(4.44)

Korolar 4.38: Postoji fragment Turingova stroja koji prevodi konfiguraciju $(l_{\sqcup}, |v|, \$v_{\sqcup}\ldots)$ u završnu konfiguraciju $(l_{\$}, 0, v_{\sqcup}\ldots)$.

Dokaz. Samo treba pomaknuti čitavu izlaznu riječ za jednu ćeliju ulijevo, brišući početni graničnik — drugim riječima, napraviti suprotno od (δ1):

$$\delta(l_\beta, \alpha) := (l_\alpha, \beta, -1), \text{ za sve } \alpha \in \Sigma_0 \uplus \{\$\},\ \beta \in \Sigma_0 \uplus \{\sqcup\}. \qquad \square \tag{δ5}$$

Teorem 4.39: Neka je Σ_0 abeceda, $\mathbb{N}\Sigma_0$ njeno kodiranje te φ_0 funkcija nad njom.
Ako je $\mathbb{N}\varphi_0$ RAM-izračunljiva, 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 $m := m_{P_0^1}$. Tvrdimo da tada funkciju φ_0 računa Turingov stroj

$$\mathcal{T}_0 := (Q, \Sigma_0, \Gamma, \sqcup, \delta, n_{\$}, l_{\$}), \tag{4.45}$$

s komponentama

$$Q := \dot{\bigcup_{i\leq 6}}\{q_i\} \uplus \dot{\bigcup_{i\leq n}}\{p_i, s_i, t_i\} \uplus \dot{\bigcup_{\alpha\in\Sigma_0\uplus\{\$,\sqcup\}}}\{n_\alpha, l_\alpha\} \uplus \{q_x\}, \tag{4.46}$$

$$\Gamma := \Sigma_0 \uplus \{\$, \#\} \uplus B^m, \text{ gdje je } B^m := \textstyle\prod_{i<m}\{\circ, \bullet\}, \tag{4.47}$$

$$\sqcup := (\circ, \circ, \ldots, \circ)^\top \in B^m \text{ (zapisana kao stupac)}, \tag{4.48}$$

i funkcijom prijelaza δ zadanom jednakostima (δ1)–(δ5), tako da se svi parovi $(q, \alpha) \in (Q \setminus \{l_{\$}\}) \times \Gamma$ nenavedeni 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 $(n_{\$}, 0, w_{\sqcup}\ldots)$ prvi fragment od $\mathcal{T}_0$ (lema 4.25) prebaciti u $(n_{\sqcup}, k+1, \$w_{\sqcup}\ldots)$, a nju će pak drugi fragment prebaciti u $(p_0, k+2, \$_{\sqcup}{}^k\,\#r_1^{\langle w\rangle}{}_{\sqcup}\ldots)$, gdje je $r_1 = (\circ, \bullet, \circ, \ldots, \circ)^\top$ (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, s izlaznim podatkom $y := \mathbb{N}\varphi_0(x) = \langle\varphi_0(w)\rangle$. Prema propoziciji 4.33, $\mathcal{T}_0$-izračunavanje s w (treća faza) će doći u konfiguraciju $(p_n, k', \$_{\sqcup}{}^k\,\#t_{\sqcup}\ldots)$, gdje je $k' > k$ i t u tragu 0 ima oblik $\bullet^y\circ^{|t|-y}$. Tu će pak konfiguraciju četvrti fragment (lema 4.36) prebaciti u $(l_{\sqcup}, |v|, \$v_{\sqcup}\ldots)$, gdje je $v := \varphi_0(w)$. Peti će fragment (korolar 4.38) napokon tu konfiguraciju prevesti u završnu konfiguraciju $(l_{\$}, 0, v_{\sqcup}\ldots)$, s izlaznom riječju $v = \varphi_0(w)$ na traci.

	stanje	pozicija	traka
početna konfiguracija	$n_{\$}$	0	$w_{\sqcup}\ldots$
nakon prve faze	$n_{\sqcup}$	$\lvert w\rvert + 1$	$\$w_{\sqcup}\ldots$
nakon druge faze	p_0	$\lvert w\rvert + 2$	$\$_{\sqcup}{}^{\lvert w\rvert}\#r_1^{\langle w\rangle}{}_{\sqcup}\ldots$
nakon treće faze	p_n	$> \lvert w\rvert$	$\$_{\sqcup}{}^{\lvert w\rvert}\#\overset{\bullet}{?}{}^{\langle v\rangle}\overset{\circ}{?}\ldots$
međukorak četvrte faze	q_4	1	$\$_{\sqcup}{}^{\lvert w\rvert+1}\overset{\bullet}{?}{}^{\langle v\rangle}\overset{\circ}{?}\ldots\$_{\sqcup}\ldots$
nakon četvrte faze	$l_{\sqcup}$	$\lvert v\rvert$	$\$v_{\sqcup}\ldots$
završna konfiguracija	$l_{\$}$	0	$v_{\sqcup}\ldots$

(4.49)

Ako pak $w \notin \mathcal{D}_{\varphi_0}$, tada će se prve dvije faze izvršiti jednako, ali iz $x \notin \mathcal{D}_{\mathbb{N}\varphi_0}$ po definiciji 1.11 slijedi da P_0-izračunavanje s x nikad neće stati, pa po propoziciji 4.33 niti $\mathcal{T}_0$-izračunavanje neće doći do p_n. To znači da ono nikad neće stati (doći do $l_{\$}$), jer svaki put od $n_{\$}$ do $l_{\$}$ vodi kroz stanje p_n. $\square$

Uočimo sličnost provedenog postupka s dokazom propozicije 1.32. Tamo smo makro-stroju omogućili da u izoliranoj okolini izvrši neki konkretni RAM-program, a ovdje smo to omogućili Turingovu stroju. Kao i u makro-programu (1.28) iz definicije 1.31, postupak se sastoji od pet faza: „otvaranje okvira" pomicanjem udesno, prijenos (kodiranje) argumenata u simulaciju (konstrukcija početne konfiguracije), sama simulacija (izvršavanje pojedinih instrukcija RAM-programa) te, ako ona stane, prijenos (dekodiranje) njene povratne vrijednosti iz završne konfiguracije i „zatvaranje okvira" pomicanjem ulijevo.

Primjer 4.40: Primjer 4.29 sad možemo pratiti kroz čitavo izračunavanje:

početna konfiguracija	$\underset{n_\$}{\texttt{ab}}$	(4.50)
uokvirivanje ulaza	$\rightsquigarrow^* \underset{q_0}{\$\texttt{ab}\#}$	(4.51)
kodiranje ulaza: $\langle \texttt{ab} \rangle = (12)_2 = 4$	$\rightsquigarrow^* \underset{p_0}{\$\sqcup\sqcup\#\circ\circ\circ\circ\bullet\bullet\bullet\bullet}$	(4.52)
računanje $2 \cdot 4 + 1 = 9$; raspisano u (4.43)	$\rightsquigarrow^* \underset{p_4}{\$\sqcup\sqcup\#\bullet\bullet\bullet\bullet\bullet\bullet\bullet\bullet\circ\circ\circ\circ\circ\circ\circ\circ\circ}$	(4.53)
uokvirivanje izlaza	$\rightsquigarrow^* \underset{q_4}{\$\sqcup\sqcup\sqcup\bullet\bullet\bullet\bullet\bullet\bullet\bullet\bullet\circ\circ\circ\circ\circ\circ\circ\circ\circ\$}$	(4.54)
dekodiranje izlaza: $9 = (121)_2 = \langle \texttt{aba} \rangle$	$\rightsquigarrow^* \underset{q_4}{\$\texttt{aba}\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\$}$	(4.55)
završna konfiguracija	$\rightsquigarrow^* \underset{l_\$}{\texttt{aba}} \circlearrowleft$.	(4.56)

Dodati a na kraj riječi Turingov stroj očito može daleko jednostavnije — ali ovaj primjer zapravo pokazuje kako može računati bilo koju funkciju čiju prateću funkciju RAM-stroj može računati. Suština računanja odvija se u primjeru 4.34 — ovo je samo hrpa „papirologije" prije i poslije, koju treba riješiti da bi taj primjer bio koristan. ◁

Ukratko, dokazali smo da je za *jezične* funkcije izračunljivost u jezičnom modelu jednako snažna kao izračunljivost njihovih pratećih funkcija u brojevnom modelu. Time smo napokon opravdali neovisnost o korištenom *encodingu* i riješili „filozofski problem" proizvoljnosti znakova abecede.

Korolar 4.41: 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 $\mathbb{N}\varphi$ parcijalno rekurzivna, tada je po teoremu 2.38 RAM-izračunljiva. Tada je po teoremu 4.39 (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. Zamjenom $\mathbb{N}\Sigma$ i $\mathbb{N}'\Sigma$ se dokaže i drugi smjer, dakle $\mathbb{N}\varphi$ je parcijalno rekurzivna ako i samo ako je $\mathbb{N}'\varphi$ parcijalno rekurzivna. □

4.2.4. Turing-izračunljivost brojevnih funkcija

Sve dosad napravljeno može se iskazati u jednom vrlo općenitom obliku.

Propozicija 4.42: Neka je Σ proizvoljna abeceda te $\mathbb{N}\Sigma$ proizvoljno njeno kodiranje.
Tada je $\varphi \mapsto \mathbb{N}\varphi$ bijekcija između skupa $\mathcal{TC}\mathrm{omp}_\Sigma$ svih Turing-izračunljivih jezičnih funkcija nad Σ, i skupa $\mathcal{C}\mathrm{omp}_1$ svih jednomjesnih RAM-izračunljivih funkcija.

Dokaz. Teorem 4.24 (zajedno s teoremom 2.38) kaže da za svaku $\varphi \in \mathcal{TC}omp_\Sigma$ vrijedi $\mathbb{N}\varphi \in \mathcal{C}omp_1$. Dakle preslikavanje koje promatramo doista „ide kamo treba".

Za injektivnost, neka su $\varphi_1, \varphi_2 \in \mathcal{TC}omp_\Sigma$ 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, ali se razlikuju na nekoj riječi w iz te domene, tada (opet po injektivnosti kodiranja riječi) vrijedi

$$\mathbb{N}\varphi_1(\langle w \rangle) = \langle \varphi_1(w) \rangle \neq \langle \varphi_2(w) \rangle = \mathbb{N}\varphi_2(\langle w \rangle), \tag{4.57}$$

pa su $\mathbb{N}\varphi_1$ i $\mathbb{N}\varphi_2$ različite jer se razlikuju na elementu $\langle w \rangle$.

Za surjektivnost, uzmimo proizvoljnu $\mathsf{F}^1 \in \mathcal{C}omp_1$ i tražimo $\varphi \in \mathcal{TC}omp_\Sigma$ takvu da je $\mathsf{F} = \mathbb{N}\varphi$. Očito, u domeni joj moraju biti upravo sve riječi w za koje je $\langle w \rangle \in \mathcal{D}_\mathsf{F}$, a svaku takvu riječ mora preslikavati u (jedinstvenu) riječ v čiji kod je $\mathsf{F}(\langle w \rangle)$. Time je jezična funkcija φ potpuno određena, a iz $\mathbb{N}\varphi = \mathsf{F} \in \mathcal{C}omp_1$ po teoremu 4.39 slijedi $\varphi \in \mathcal{TC}omp_\Sigma$. Dobivenu funkciju φ označavamo s $\mathbb{N}^{-1}\mathsf{F}$. □

Precizno, za $\sigma := \mathbb{N}\Sigma^*$, imamo vezu $\mathsf{F} = \mathbb{N}\varphi = \sigma \circ \varphi \circ \sigma^{-1} \iff \varphi = \mathbb{N}^{-1}\mathsf{F} = \sigma^{-1} \circ \mathsf{F} \circ \sigma$ te vrijedi $\mathbb{N}\,\mathbb{N}^{-1}\mathsf{F} = \mathsf{F}$ i $\mathbb{N}^{-1}\mathbb{N}\varphi = \varphi$.

Korolar 4.43: Neka je Σ abeceda s kodiranjem $\mathbb{N}\Sigma$ te F^1 brojevna funkcija.

Tada je F parcijalno rekurzivna ako i samo ako je $\mathbb{N}^{-1}\mathsf{F}$ Turing-izračunljiva.

Dokaz. Ako je $\mathsf{F} = \mathbb{N}\,\mathbb{N}^{-1}\mathsf{F}$ parcijalno rekurzivna, tada je po teoremu 2.38 RAM-izračunljiva, pa je po teoremu 4.39 $\mathbb{N}^{-1}\mathsf{F}$ Turing-izračunljiva. S druge strane, ako je $\mathbb{N}^{-1}\mathsf{F}$ Turing-izračunljiva, onda je po teoremu 4.24 $\mathbb{N}\,\mathbb{N}^{-1}\mathsf{F} = \mathsf{F}$ parcijalno rekurzivna. □

Vidimo da korolar 4.43 vrijedi bez obzira na abecedu i kodiranje. Važan je posebni slučaj jednočlane abecede, kojoj je kodiranje jedinstveno i podudara se s duljinom (lema 4.35).

Definicija 4.44: *Unarna abeceda* je $\Sigma_\bullet := \{\bullet\}$, s kodiranjem $\mathbb{N}\Sigma_\bullet(\bullet) := 1$. (Tada je $\langle \cdots \rangle = |\cdots|$.)

Za jednomjesnu brojevnu funkciju F^1, jezičnu funkciju $\mathbb{N}^{-1}\mathsf{F}$ nad $\Sigma_\bullet$ zovemo *unarnom reprezentacijom* od F i označavamo je $\bullet\mathsf{F}$. ◁

Svaka riječ u nad unarnom abecedom ($u \in \Sigma_\bullet^*$) je oblika $u = \bullet^n$, gdje je $n = |u| = \langle u \rangle$. Sada definicija preslikavanja $\mathbb{N}^{-1}$ kaže da je $\bullet\mathsf{F}(\bullet^n) = \bullet^{\mathsf{F}(n)}$ ako je $n \in \mathcal{D}_\mathsf{F}$, a $\bullet^n \notin \mathcal{D}_{\bullet\mathsf{F}}$ inače. Po napomeni 1.4, pišemo $\bullet\mathsf{F}(\bullet^n) \simeq \bullet^{\mathsf{F}(n)}$.

Primjer 4.45: $\bullet\mathsf{factorial}\big(\bullet\mathsf{prime}(\bullet)\big) = \bullet\mathsf{factorial}(\bullet\bullet\bullet) = \bullet\bullet\bullet\bullet\bullet\bullet$, jer je $p_1! = 3! = 6$.
Za inicijalne funkcije, $\bullet\mathsf{I}_1^1(w) = w$, $\bullet\mathsf{Sc}(w) = w\bullet$, a $\bullet\mathsf{Z}(w) = \varepsilon$ za sve $w \in \Sigma_\bullet^*$. ◁

Korolar 4.46: Neka je F^1 brojevna funkcija.

Tada je F parcijalno rekurzivna ako i samo ako je $\bullet\mathsf{F}$ Turing-izračunljiva.

Dokaz. Već rekosmo, ovo je posebni slučaj korolara 4.43, za unarnu abecedu. □

4.3. Izračunljivost jezikā

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? U brojevnom modelu izračunljivost L gledamo preko kodova: definiramo jednomjesnu brojevnu relaciju

$$\langle L \rangle := \{\langle w \rangle \mid w \in L\} = \mathbb{N}\Sigma^*[L], \tag{4.58}$$

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, prazni jezik $\varnothing$ je primitivno rekurzivan jer je $\chi_{\langle \varnothing \rangle} = \chi_\varnothing = \mathsf{Z}$ primitivno rekurzivna (čak inicijalna). Također, univerzalni jezik Σ^* je primitivno rekurzivan, jer je $\langle \Sigma^* \rangle = \mathbb{N}\Sigma^*[\Sigma^*] = \mathcal{I}_{\mathbb{N}\Sigma^*} = \mathbb{N}$, čija je karakteristična funkcija $\mathsf{C}_1^1 = \mathsf{Sc} \circ \mathsf{Z}$.

U jezičnom modelu, moramo vidjeti što znači da neki Turingov stroj računa χ_L. To je Turingov stroj nad Σ, i ulaz $w \in \Sigma^*$ mu se daje na isti način kao i običnom Turingovu stroju: kroz početnu konfiguraciju $(q_0, 0, w_\sqcup \ldots)$. 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 takve Turingove strojeve obično zadajemo kao $(Q, \Sigma, \Gamma, \sqcup, \delta, q_0, q_a, q_r)$ — imaju *dva* završna stanja kojima 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 dosadašnjim Turingovim strojevima, samo što je konfiguracija sa stanjem q_r (kao i ona s q_a) završna: prelazi u samu sebe, a ne po funkciji δ. Međutim, 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) i zovemo ih *odlučiteljima* (*deciders*). Svaki odlučitelj $\mathcal{T}$ dijeli Σ^* na dva dijela,

$$L(\mathcal{T}) := \{w \in \Sigma^* \mid \mathcal{T}\text{-izračunavanje s } w \text{ stane u stanju } q_a\} \text{ i} \tag{4.59}$$

$$\big(L(\mathcal{T})\big)^c = \{w \in \Sigma^* \mid \mathcal{T}\text{-izračunavanje s } w \text{ stane u stanju } q_r\}, \tag{4.60}$$

i kažemo da *prepoznaje* $L(\mathcal{T})$.

Nije preteško vidjeti da su te dvije karakterizacije, brojevna i jezična, povezane — pogotovo jer već imamo napravljen najveći dio posla.

Teorem 4.47: Neka je L jezik (nad nekom abecedom Σ).
Ako postoji Turingov odlučitelj koji prepoznaje L, tada je relacija $\langle L \rangle$ rekurzivna.

Dokaz. Pretpostavimo da je $\mathcal{T}$ odlučitelj za L i provedimo s tim strojem postupak iz točke 4.1. Navest ćemo samo detalje koje je potrebno promijeniti.

Prvo, kako imamo dva završna stanja, moramo fiksirati njihove kodove: recimo, $\mathbb{N}Q(q_a) := 1$, a $\mathbb{N}Q(q_r) := 2$. (Dokažite analogon leme 4.16, da bez smanjenja općenitosti možemo pretpostaviti $q_0 \neq q_a$ i $q_0 \neq q_r$ — a po definiciji odlučitelja mora biti $q_a \neq q_r$.)

Drugo, kako nam δ sad nije definirana na oba završna stanja, treba promijeniti uvjet u lemi analognoj lemi 4.20, u $q \notin \{q_a, q_r\}$. Tu se ništa bitno ne mijenja, osim što će direction-tablica (vidjeti primjer 4.21) imati dva retka jedinicā, a ne samo jedan.

I treće, umjesto parcijalno rekurzivne funkcije stop imat ćemo funkciju zadanu sa

$$\mathsf{stop}'(x) := \mu n\big(\mathsf{State}(x, n) \in \{1, 2\}\big), \tag{4.61}$$

za koju ćemo moći zaključiti da je rekurzivna. Naime, parcijalno je rekurzivna kao minimizacija rekurzivne (lema 4.22 i korolar 2.48) relacije. Totalna je (pa smo u (4.61) mogli koristiti ':=') po definiciji odlučitelja: ako je $x = \langle w \rangle$, tada mora za neki n biti $\mathsf{State}(x, n) \in \{1, 2\}$, jer $\mathcal{T}$-izračunavanje s w mora stati — i $\mathsf{stop}'(x) = \mathsf{stop}'(\langle w \rangle)$ je upravo broj koraka nakon kojeg ono stane.

Tvrdimo da je

$$x \in \langle L \rangle \iff \mathsf{State}\big(x, \mathsf{stop}'(x)\big) = 1, \tag{4.62}$$

iz čega slijedi rekurzivnost od $\langle L \rangle$ po lemi 2.33 i korolaru 2.35.

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_a — označimo s n_0 broj koraka nakon kojeg se to dogodi. Tada je $\mathsf{State}(x, n_0) = 1 \in \{1, 2\}$, dok za sve $n < n_0$ konfiguracija nakon n koraka nije završna (sasvim analogno propoziciji 1.13 — jer završne konfiguracije prelaze isključivo u same sebe, postoji najviše jedna završna konfiguracija u izračunavanju) pa $\mathsf{State}(x, n) \notin \{1, 2\}$. Dakle $n_0 = \mathsf{stop}'(x)$, pa je $\mathsf{State}\big(x, \mathsf{stop}'(x)\big) = \mathsf{State}(x, n_0) = 1$.

U drugom smjeru, pretpostavimo $\mathsf{State}\big(x, \mathsf{stop}'(x)\big) = 1$. 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) $\mathsf{State}(x, n) \notin \{1, 2\}$ za $n < n_0$, a $\mathsf{State}(x, n_0) = 1 \in \{1, 2\}$, pa je $\mathsf{stop}'(x) = n_0$. Sada pretpostavka glasi $\mathsf{State}(x, n_0) = 1$, odnosno završno stanje je q_a, pa $\mathcal{T}$ prihvaća w. To znači da je $w \in L$, odnosno $x = \langle w \rangle \in \langle L \rangle$. □

Teorem 4.48: 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 2.38 postoji RAM-program P koji je računa; i totalna je, pa P-izračunavanje stane sa svakim ulazom. Na P bismo htjeli primijeniti postupak iz točke 4.2 — dakle moramo $\chi_{\langle L \rangle}$ prikazati kao $\mathbb{N}\varphi$ za neku jezičnu funkciju φ nad Σ. Možemo li to?

Svakako: dokaz propozicije 4.42 kaže da je $\varphi := \mathbb{N}^{-1}\chi_{\langle L \rangle} = \sigma^{-1} \circ \chi_{\langle L \rangle} \circ \sigma$, uz oznaku $\sigma := \mathbb{N}\Sigma^*$. Funkcija φ je totalna kao kompozicija tri totalne, a Turing-izračunljiva je prema korolaru 4.43 (ako uzmemo isto kodiranje pomoću kojeg smo izračunali $\langle L \rangle$). Pogledajmo detaljnije kako ona djeluje. Ako joj damo riječ $w \in \Sigma^* \setminus L$, tada ($\sigma$ je injekcija) vrijedi $\langle w \rangle \notin \langle L \rangle$, pa je $\chi_{\langle L \rangle}(\langle w \rangle) = 0$. Tada je

$$\varphi(w) = \sigma^{-1}\big(\chi_{\langle L \rangle}(\sigma(w))\big) = \sigma^{-1}\big(\chi_{\langle L \rangle}(\langle w \rangle)\big) = \sigma^{-1}(0) = \varepsilon. \tag{4.63}$$

Dakle, riječi izvan L ona preslikava u praznu riječ. Riječi *unutar* L onda ne smije preslikavati u ε zbog injektivnosti σ^{-1}, a jer je totalna, mora ih preslikavati nekamo. Dakle, $\varphi(w)$ je neprazna riječ za svaku $w \in L$. Vidimo da, baš kao i u $\mathbb{N}$, imamo prirodnu reprezentaciju `bool` u Σ^*: prazna riječ je lažna, neprazne su istinite. Mnogi programski jezici koji definiraju istinitost stringova, definiraju je upravo na taj način.

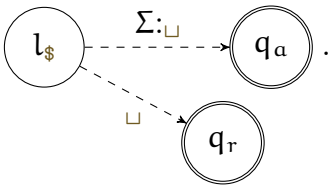
Naš Turingov stroj koji računa φ dobiven je transpiliranjem RAM-programa P koji računa $\chi_{\langle L \rangle}$. Srećom, u završnoj konfiguraciji pozicija mu je 0, pa je vrlo lako vidjeti je li izlazna riječ prazna: znak koji čitamo tada je ili praznina ili element ulazne abecede. Proglasimo $l_{\$}$

običnim stanjem, i dodajmo prijelaze

$$\delta(l_\$, \alpha) := (q_a, \sqcup, -1), \text{ za sve } \alpha \in \Sigma, \tag{δ5a}$$

$$\delta(l_\$, \sqcup) := (q_r, \sqcup, -1). \tag{δ5b}$$

Po teoremu 4.39, dobiveni stroj će „računati" (pod navodnicima jer odlučitelji zapravo ne služe za računanje funkcija) funkciju φ, pa će za $w \in L$ rezultat biti neprazan. To znači da će stroj na kraju izračunavanja pročitati znak iz Σ (možemo ga zamijeniti razmakom da traka bude sasvim prazna; to je jedini znak izlaza jer zbog leme 4.35 vrijedi $|\text{izlaz}| \leq \langle\text{izlaz}\rangle = \langle\varphi(\text{ulaz})\rangle = \chi_{\langle L\rangle}(\langle\text{ulaz}\rangle) \leq 1$) i ući u stanje q_a (δ5a). Za $w \notin L$ rezultat će biti ε pa će stroj pročitati prazninu i ući u stanje q_r (δ5b). Budući da za svaku riječ $w \in \Sigma^*$ vrijedi jedna od te dvije mogućnosti, zaključujemo da smo doista konstruirali odlučitelj. □

Dijagramatski, na kraj Turingova stroja dodali smo fragment 

.

4.4. Turing-izračunljivost višemjesnih funkcija

Dosadašnji rezultati pokazuju da je za jezične funkcije, i za *jednomjesne* brojevne funkcije, svejedno računamo li ih na Turingovu stroju ili na nekom brojevnom modelu izračunavanja (npr. RAM-stroju). Da bismo isto dokazali i za *višemjesne* brojevne funkcije, trebamo precizirati reprezentaciju njihovih ulaza.

Već smo u uvodu nagovijestili, koristit ćemo kontrakciju — u abecedu ćemo dodati separator / kojim ćemo razdvojiti više ulaznih podataka: npr. $(1, 0, 5, 0)$ prenijet ćemo kao •//•••••/.

Definicija 4.49: *Binarna abeceda* je abeceda $\Sigma_\beta := \{\bullet, /\}$. Za svaki neprazni konačni niz prirodnih brojeva $\vec{x} = (x_1, x_2, \ldots, x_k)$, definiramo *binarnu reprezentaciju* kao

$$\beta(\vec{x}) := \bullet^{x_1}/\bullet^{x_2}/\cdots/\bullet^{x_k} \in \Sigma_\beta^*. \tag{4.64}$$

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 4.39), ali zapravo je možemo promatrati u bilo kojem smjeru.

Propozicija 4.50: Funkcija β je bijekcija između $\mathbb{N}^+$ i Σ_β^*.

Dokaz. Najlakše je konstruirati inverznu funkciju i pokazati da je to doista inverz. Dakle, za proizvoljnu riječ $u \in \Sigma_\beta^*$, pitamo se kojeg niza je ona kod. Kao i uvijek kod konačnih nizova, trebamo odrediti njegovu duljinu k, a zatim pojedine članove $x_1, x_2, \ldots, x_k$. Duljina je sljedbenik broja pojavljivanja separatora / u riječi u (jer u (4.64) ima $k-1$ separatora) i to je uvijek pozitivan broj, dakle niz je neprazan. Prvi član mu možemo odrediti brojeći kružiće do prvog separatora (ili do kraja riječi ako je $k = 1$), drugi brojeći ih između prvog i drugog separatora, i tako dalje. Zadnji član x_k je broj kružića 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 kružića $(\sum \vec{x})$ te se podudaraju na pozicijama svih separatora (malo pomaknute parcijalne sume od $\vec{x}$) — što je dovoljno za zaključak 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 (4.64) ima točno x_i kružića. □

Za same funkcije, koristit ćemo istu ideju kao u (4.14): reprezentaciju ulaza preslikamo u reprezentaciju izlaza (ako je izlaz definiran), dok ne-reprezentacije ne preslikamo nikamo. Ovdje treba biti oprezan zbog propozicije 4.50, 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 $\bullet^y = \beta(y) \in \beta[\mathbb{N}] = \mathcal{I}_{\beta^1} = \Sigma_\bullet^*$.

Definicija 4.51: Neka je $k \in \mathbb{N}_+$ te f^k brojevna funkcija. Jezičnu funkciju βf nad Σ_β, zadanu s

$$\beta f(u) :\simeq \beta\big(f(\vec{x})\big), \text{ za } u = \beta(\vec{x}^k), \tag{4.65}$$

zovemo *binarnom reprezentacijom* funkcije f. ◁

Vjerojatno ste nekad napisali Turingov stroj za $\beta\mathsf{add}^2$, koristeći $\beta\mathsf{add}^2(u/v) = uv$, ako su $u, v \in \Sigma_\bullet^*$. Ipak, već tada ste zasigurno primijetili koliko bi bilo teško napisati $\beta\mathsf{mul}^2$, a pogotovo $\beta\mathsf{pow}$ — ostale divote iz poglavlja 3 ($\beta\mathsf{part}$?!) 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 $\beta\mathsf{univ}$ — čineći tako jedan od modela univerzalne izračunljivosti. Prvo ćemo dokazati obrat: one brojevne funkcije čije binarne reprezentacije su Turing-izračunljive, parcijalno su rekurzivne. To nije toliko impresivan rezultat, ali lakši je za dokazati, a osnovnu ideju dokaza moći ćemo upotrijebiti i u dokazu obrata.

Dakle, neka je $k \in \mathbb{N}_+$ te f^k brojevna funkcija takva da je βf Turing-izračunljiva. Kodirajmo $\mathbb{N}\Sigma_\beta$: otprije imamo $\langle\bullet\rangle = 1$, dodefinirajmo još $\langle/\rangle := 2$. Po teoremu 4.24 $\mathbb{N}\beta f$ je parcijalno rekurzivna — ali nas zanima f. Brojevne funkcije $(\mathbb{N}\beta f)^1$ i f^k su povezane preko jezične funkcije βf. Možemo li naći brojevnu vezu između njih? Precizno, možemo li naći brojevne 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.

U traženju tih funkcija može pomoći dijagram.

$$\begin{tikzcd}
\mathbb{N}^k \arrow[r, "f"] \arrow[d, "\beta^k"] & \mathbb{N} \arrow[d, "\beta^1"] \\
\Sigma_\beta^* \arrow[r, "\beta f"] \arrow[d, "\mathbb{N}\Sigma_\beta^*"] & \Sigma_\bullet^* \subset \Sigma_\beta^* \arrow[d, "\mathbb{N}\Sigma_\beta^*"] \\
\mathbb{N} \arrow[r, "\mathbb{N}\beta f"] & \mathbb{N}
\end{tikzcd}
\qquad
\begin{tikzcd}
(3,1,2) \arrow[r, mapsto, "\mathsf{mul}^3"] \arrow[d, mapsto, "\beta^3"] & 6 \arrow[d, mapsto, "\beta^1"] \\
\bullet\bullet\bullet/\bullet/\bullet\bullet \arrow[r, mapsto, "\beta\mathsf{mul}^3"] \arrow[d, mapsto, "\mathbb{N}\Sigma_\beta^*"] & \bullet\bullet\bullet\bullet\bullet\bullet \arrow[d, mapsto, "\mathbb{N}\Sigma_\beta^*"] \\
275 \arrow[r, mapsto, "\mathbb{N}\beta\mathsf{mul}^3"] & 63
\end{tikzcd} \tag{4.66}$$

Lijevo je općenita slika, desno je jedan konkretan primjer (računanje mul^3 na $(3,1,2)$). Brojevi u donjem retku razumljiviji su u pomaknutoj bazi 2: $275 = (11121211)_2$, a $63 = (111111)_2$.

Gornji pravokutnik predstavlja definiciju (4.65), dok donji predstavlja (4.14) za posebni slučaj binarne abecede i jezične funkcije βf nad njom.

Lema 4.52: Dijagram prikazan lijevo u (4.66) (samo pune strelice) komutira.

Drugim riječima, za svaki od ta dva pravokutnika, pa onda i za veliki pravokutnik sastavljen od ta dva, svejedno je kojim putom dođemo od njegova gornjeg lijevog do donjeg desnog vrha.

Dokaz. Prvo 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 vrijednost lijeve funkcije $\beta f\big(\beta^k(\vec{x})\big)$, što je po definiciji (4.65) jednako $\beta^1\big(f(\vec{x})\big) = (\beta^1 \circ f)(\vec{x})$. 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 jednostavniji jer je $\sigma := \mathbb{N}\Sigma_\beta^*$ bijekcija: uz oznaku $\varphi := \beta f$,

$$\mathbb{N}\beta f \circ \sigma = \mathbb{N}\varphi \circ \sigma = \sigma \circ \varphi \circ \sigma^{-1} \circ \sigma = \sigma \circ \varphi = \sigma \circ \beta f. \tag{4.67}$$

Sada iz te dvije jednakosti slijedi

$$\sigma \circ \beta^1 \circ f = \sigma \circ \beta f \circ \beta^k = \mathbb{N}\beta f \circ \sigma \circ \beta^k, \tag{4.68}$$

odnosno čitav „vanjski okvir” komutira. □

Upravo dokazana jednakost je korisna, jer pruža način da se u potpunosti izbjegnu jezične funkcije: vrhovi vanjskog pravokutnika su brojevni pa se njegove stranice mogu opisati brojevnim funkcijama. Gornju i donju stranicu već imamo, još je preostalo precizirati lijevu i desnu.

4.4.1. Funkcije bin^k — binarno kodiranje

Definicija 4.53: Za svaki $k \in \mathbb{N}_+$, definiramo $\mathsf{bin}^k := \mathbb{N}\Sigma_\beta^* \circ \beta^k$. ◁

Te brojevne funkcije su na dijagramu prikazane točkanim strelicama: lijeva stranica vanjskog pravokutnika je bin^k, a desna bin^1. Svaka bin^k je injekcija kao kompozicija dvije injekcije. Jesu li izračunljive bez pozivanja na jezični model?

Kako bismo izračunali $\mathsf{bin}(3,1,2) = (11121211)_2 = 275$ ili $\mathsf{bin}(6) = (111111)_2 = 63$ koristeći samo brojevne funkcije? Ovo drugo se čini bitno lakšim.

Lema 4.54: Funkcija bin^1 je primitivno rekurzivna.

Dokaz. Možemo odmah napisati točkovnu definiciju $\mathsf{bin}(x) = \sum_{i<x} 2^i$, pa će primitivna rekurzivnost slijediti iz leme 2.53 — ali svaki računarac zna da to ima i zatvoreni oblik: $\mathsf{bin}(x) = 2^x - 1 = \mathsf{pd}\big(\mathsf{pow}(2,x)\big)$ (jer je uvijek $2^x \geq 1$). □

Kako sada pomoću funkcije bin^1 možemo dobiti bin^2? 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 $\mathsf{bin}(x)$, 2 i $\mathsf{bin}(y)$. Dakle, da dobijemo njen kod, trebamo ih konkatenirati u pomaknutoj bazi 2. Operacija je definirana s (4.11).

Propozicija 4.55: Neka je Σ abeceda, $\mathbb{N}\Sigma$ njeno kodiranje te b broj znakova u njoj.
Tada za sve riječi $u, v \in \Sigma^*$ vrijedi $\langle uv \rangle = \langle u \rangle \mathbin{\widehat{b}} \langle v \rangle$.

Dokaz. Uz oznake $u = \alpha_1 \alpha_2 \cdots \alpha_{|u|}$ i $v = \beta_1 \beta_2 \cdots \beta_{|v|}$, vrijedi

$$\begin{aligned}\langle uv \rangle &= \langle \alpha_1 \alpha_2 \cdots \alpha_{|u|} \beta_1 \beta_2 \cdots \beta_{|v|} \rangle = \big(\mathbb{N}\Sigma(\alpha_1) \cdots \mathbb{N}\Sigma(\alpha_{|u|}) \mathbb{N}\Sigma(\beta_1) \cdots \mathbb{N}\Sigma(\beta_{|v|})\big)_b = \\ &= \big(\mathbb{N}\Sigma(\alpha_1) \cdots \mathbb{N}\Sigma(\alpha_{|u|})\big)_b \mathbin{\widehat{b}} \big(\mathbb{N}\Sigma(\beta_1) \cdots \mathbb{N}\Sigma(\beta_{|v|})\big)_b = \langle \alpha_1 \cdots \alpha_{|u|} \rangle \mathbin{\widehat{b}} \langle \beta_1 \cdots \beta_{|v|} \rangle = \langle u \rangle \mathbin{\widehat{b}} \langle v \rangle.\end{aligned}$$

Za $u = \varepsilon$ ili $v = \varepsilon$ tvrdnja također vrijedi jer je ε neutralni element za konkatenaciju, a $\langle \varepsilon \rangle = 0$ je neutralni element za $\mathbin{\widehat{b}}$: doista,

$$0 \mathbin{\widehat{b}} x = \mathsf{sconcat}(0, x, b) = 0 \cdot b^{\mathsf{slh}(x,b)} + x = 0 + x = x, \tag{4.69}$$

$$x \mathbin{\widehat{b}} 0 = \mathsf{sconcat}(x, 0, b) = x \cdot b^{\mathsf{slh}(0,b)} + 0 = x \cdot b^0 = x, \tag{4.70}$$

pri čemu je $\mathsf{slh}(0, b) = (\mu t \leq 0)\big(\sum_{i \leq t} b^i > 0\big) = 0$ jer je već $\sum_{i \leq 0} b^i = b^0 = 1 > 0$. □

Korolar 4.56: Za sve $x, y \in \mathbb{N}$, za sve $b \in \mathbb{N}_+$, vrijedi $\mathsf{slh}(x \mathbin{\widehat{b}} y, b) = \mathsf{slh}(x, b) + \mathsf{slh}(y, b)$.

Dokaz. Neka su x, y, b proizvoljni (b pozitivan). Uzmimo neku b-članu abecedu, primjerice $[1 \cdot\cdot\, b]$. Po propoziciji 4.14, postoje $u, v \in [1 \cdot\cdot\, b]^*$ takve da je $x = \langle u \rangle$ i $y = \langle v \rangle$ (to su upravo zapisi brojeva x i y u pomaknutoj bazi b). Po definiciji slh, vrijedi $\mathsf{slh}(\langle w \rangle, b) = |w|$ za sve $w \in [1 \cdot\cdot\, b]^*$, pa imamo

$$\begin{aligned}\mathsf{slh}(x, b) + \mathsf{slh}(y, b) &= \mathsf{slh}(\langle u \rangle, b) + \mathsf{slh}(\langle v \rangle, b) = |u| + |v| = |uv| = \mathsf{slh}(\langle uv \rangle, b) = \\ &[\text{propozicija 4.55}] = \mathsf{slh}(\langle u \rangle \mathbin{\widehat{b}} \langle v \rangle, b) = \mathsf{slh}(x \mathbin{\widehat{b}} y, b). \quad \square\end{aligned} \tag{4.71}$$

Propozicija 4.57: Za svaki $b \in \mathbb{N}_+$, $\mathbin{\widehat{b}}$ je primitivno rekurzivna asocijativna operacija.

Dokaz. Primitivna rekurzivnost slijedi iz leme 4.13. Za asocijativnost,

$$\begin{aligned}(x \mathbin{\widehat{b}} y) \mathbin{\widehat{b}} z &= \big(x \cdot b^{\mathsf{slh}(y,b)} + y\big) \cdot b^{\mathsf{slh}(z,b)} + z = \\ &= \big(x \cdot b^{\mathsf{slh}(y,b)} \cdot b^{\mathsf{slh}(z,b)} + y \cdot b^{\mathsf{slh}(z,b)}\big) + z = x \cdot b^{\mathsf{slh}(y,b) + \mathsf{slh}(z,b)} + \big(y \cdot b^{\mathsf{slh}(z,b)} + z\big) = \\ &[\text{korolar 4.56}] = x \cdot b^{\mathsf{slh}(y \mathbin{\widehat{b}} z, b)} + (y \mathbin{\widehat{b}} z) = x \mathbin{\widehat{b}} (y \mathbin{\widehat{b}} z) \quad \square\,.\end{aligned} \tag{4.72}$$

Zbog asocijativnosti ćemo ubuduće pisati izraze poput $x \mathbin{\widehat{b}} y \mathbin{\widehat{b}} z$ bez zagrada. Uočimo da je za asocijativnost ključno da se radi o **pomaknutoj** bazi: u običnoj bazi 10 je $(5 \mathbin{\widehat{10}} 0) \mathbin{\widehat{10}} 2 = 50 \mathbin{\widehat{10}} 2 = 502 \neq 5 \mathbin{\widehat{10}} (0 \mathbin{\widehat{10}} 2) = 5 \mathbin{\widehat{10}} 2 = 52$ pa ne vrijedi asocijativnost.

Propozicija 4.58: 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 4.54.
Pretpostavimo da je bin^l primitivno rekurzivna za neki $l \in \mathbb{N}_+$. Tada je

$$\begin{aligned}\mathsf{bin}^{l+1}(\vec{x}, y) &= \langle \beta(\vec{x}, y) \rangle = \langle \bullet^{x_1} / \cdots / \bullet^{x_l} / \bullet^{y} \rangle = \langle \beta(\vec{x}) / \beta(y) \rangle = \\ &[\text{propozicija 4.55}] = \langle \beta(\vec{x}) \rangle \mathbin{\widehat{2}} \langle / \rangle \mathbin{\widehat{2}} \langle \beta(y) \rangle = \mathsf{bin}^k(\vec{x}) \mathbin{\widehat{2}} 2 \mathbin{\widehat{2}} \mathsf{bin}^1(y),\end{aligned} \tag{4.73}$$

što je primitivno rekurzivno po pretpostavci indukcije, propoziciji 4.57 i lemi 4.54. □

Promotrimo sada dijagram lijevo u (4.66), gledajući i točkane strelice. Definicija 4.53 zapravo kaže da „kružni odsječak" sa svake strane tog dijagrama komutira, pa iz toga i leme 4.52 slijedi da čitav „vanjski oval" također komutira.

Korolar 4.59: Neka je $k \in \mathbb{N}_+$ te f^k funkcija. Tada vrijedi $\mathsf{bin}^1 \circ f = \mathbb{N}\beta f \circ \mathsf{bin}^k$.

Dokaz. To je jednakost (4.68) u koju je uvrštena (s obje strane) definicija 4.53. □

Da bismo izrazili f pomoću $\mathbb{N}\beta f$, potreban nam je (izračunljivi) lijevi inverz za bin^1. Kako možemo iz $63 = (111111)_2 = \mathsf{bin}(6)$ natrag dobiti 6? Samo prebrojimo znamenke.

Lema 4.60: Definiramo funkciju blh s $\mathsf{blh}(n) := \mathsf{slh}(n, 2)$.
Funkcija blh je primitivno rekurzivna te vrijedi $\mathsf{blh} \circ \mathsf{bin}^1 = I_1^1$.

Dokaz. Primitivna rekurzivnost slijedi iz simboličke definicije $\mathsf{blh} = \mathsf{slh} \circ (I_1^1, C_2^1)$, po lemi 4.13 i propoziciji 2.21. Za kompoziciju, uzmimo proizvoljni $n \in \mathbb{N}$ i računamo $\mathsf{blh}\big(\mathsf{bin}(n)\big) = \mathsf{blh}(2^n - 1)$. Relacija koju minimiziramo (po $t \leq 2^n - 1$) je

$$\textstyle\sum_{i \leq t} 2^i > 2^n - 1 \iff 2^{t+1} - 1 > 2^n - 1 \iff t + 1 > n \iff t \geq n \tag{4.74}$$

pa je najmanji takav t upravo $n = I_1^1(n) \leq 2^n - 1$. □

Teorem 4.61: Neka je $k \in \mathbb{N}_+$ te 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, $\mathbb{N}\beta f$ je parcijalno rekurzivna. Iz leme 4.60 i korolara 4.59 sada imamo

$$f = I_1^1 \circ f = \mathsf{blh} \circ \mathsf{bin}^1 \circ f = \mathsf{blh} \circ \mathbb{N}\beta f \circ \mathsf{bin}^k, \tag{4.75}$$

odnosno f je dobivena kompozicijom iz tri funkcije, od kojih je srednja parcijalno rekurzivna, a prva i zadnja su primitivno rekurzivne (lema 4.60 i propozicija 4.58), pa su rekurzivne (korolar 2.35), a time i parcijalno rekurzivne. Kako je skup parcijalno rekurzivnih funkcija zatvoren na kompoziciju, f je parcijalno rekurzivna. □

Istim dijagramom možemo dokazati i obrat teorema 4.61. Jednadžba

$$\mathsf{bin}^1 \circ f = \mathbb{N}\beta f \circ \mathsf{bin}^k \tag{4.76}$$

iz korolara 4.59 može poslužiti i za dobivanje $\mathbb{N}\beta f$ iz f — umjesto desne trebamo obrnuti lijevu stranicu pravokutnika, odnosno naći desni inverz za bin^k. Nažalost, to je iz tri razloga zapetljanije nego ovo što smo napravili u dokazu teorema 4.61.

Prvi razlog tiče se napomene 1.1. Zbog $\mathcal{D}_{\mathsf{bin}^k} = \mathbb{N}^k$, traženi desni inverz ima k izlaznih podataka, pa ga reprezentiramo s k brojevnih funkcija $\mathsf{arg}_1, \mathsf{arg}_2, \ldots, \mathsf{arg}_k$ — tako nazvanih jer ekstrahiraju pojedine argumente funkcije f iz jedinog argumenta a funkcije $\mathbb{N}\beta f$. Te funkcije su neovisne o k: recimo, $\mathsf{arg}_2\big(\mathsf{bin}^2(x, y)\big) = \mathsf{arg}_2\big(\mathsf{bin}^4(x, y, z, t)\big) = y$, pa ih ne moramo označavati s dva broja, poput I_n^k.

Drugo, implementacija je bitno kompliciranija nego u lemi 4.60. Dok su riječi nad jednočlanom abecedom trivijalno izomorfne s $\mathbb{N}$ ($w \mapsto |w|$ je izomorfizam), riječi nad dvočlanom

abecedom imaju bogatu strukturu, za čije raščlanjivanje trebamo kompliciranije funkcije — svojevrsnu biblioteku <string.h> za rad s riječima nad binarnom abecedom. U sljedećoj točki napravit ćemo minimum potreban za implementaciju funkcija arg_i.

Treće, funkcija bin^k nije surjekcija, pa *nema* totalni desni inverz. Precizno, $\mathbb{N}\beta f$ nije *jednaka* $\mathsf{bin}^1 \circ f \circ (\mathsf{arg}_1, \ldots, \mathsf{arg}_k)$, već je restrikcija te funkcije na $\mathcal{I}_{\mathsf{bin}^k}$. Recimo, u slučaju kad a ima 7 znamenaka 2 u pomaknutoj bazi 2, izraz $\mathbb{N}\beta\mathsf{mul}^3(a)$ nema vrijednost, dok je $2^{\mathsf{arg}_1(a)\cdot\mathsf{arg}_2(a)\cdot\mathsf{arg}_3(a)} - 1$ ima. Srećom, imamo tehniku (restrikcija na rekurzivan skup) koja će uskladiti domene.

4.4.2. Standardna biblioteka za Σ_β^*

Propozicija 4.62: 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 stavimo $u := \varepsilon$. 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 $w = w(uu')$ pa je $|w| = |w(uu')| = |w| + |uu'|$, iz čega $|uu'| = 0$. Kako je ε jedina riječ duljine 0, imamo $uu' = \varepsilon$, i analogno $u = u' = \varepsilon$, dakle $w = vu = v\varepsilon = v$. □

Korolar 4.63: Dvomjesna brojevna relacija $\sqsubseteq_\beta$, zadana s

$$x \sqsubseteq_\beta y :\Longleftrightarrow \text{„}x \text{ je kod nekog prefiksa riječi čiji kod je } y\text{”} \tag{4.77}$$

(kodovi su po $\mathbb{N}\Sigma_\beta^*$), primitivno je rekurzivan refleksivni parcijalni uređaj na $\mathbb{N}$.

Dokaz. Da se radi o parcijalnom uređaju slijedi iz propozicija 4.62 i 4.14, a primitivna rekurzivnost iz lema 4.13 i 4.60: $x \sqsubseteq_\beta y \Longleftrightarrow x = \mathsf{sprefix}\big(y, \mathsf{blh}(x), 2\big)$. □

Primjer 4.64: Recimo, vrijedi $4 = (12)_2 = \langle \bullet/ \rangle \sqsubseteq_\beta \langle \bullet/\bullet\bullet\bullet \rangle = (12111)_2 = 39$. ◁

Za sljedeću lemu, trebat će nam *uokvireni* brojevi: oni čiji zapisi u pomaknutoj bazi 2 počinju i završavaju znamenkom 2. Želimo „isprogramirati” dokaz propozicije 4.50, koji ima nekoliko slučajeva ovisno o tome nalazimo li se prije prvog separatora, nakon zadnjeg, ili između njih. Ako riječ ima separatore na oba kraja, svaki x_i možemo odrediti brojeći jedinice između susjednih separatora.

Lema 4.65: Postoje primitivno rekurzivne funkcije $\mathsf{pos2}$ i $\mathsf{streak1}$, takve da za svaki uokvireni broj x, za svaki $i \in \mathbb{N}$, vrijedi:

$\mathsf{pos2}(x, i)$ je pozicija i-te dvojke (ili $\mathsf{blh}(x)$ ako takva ne postoji) te

$\mathsf{streak1}(x, i)$ 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 2.54) funkciju koja broji dvojke (separatore) u zapisu broja u pomaknutoj bazi 2.

$$\mathsf{count2}(x) := \big(\#\, i < \mathsf{blh}(x)\big)\big(\mathsf{sdigit}(x, i, 2) = 2\big) \tag{4.78}$$

Tvrdimo da primitivno rekurzivne funkcije zadane točkovno s

$$\mathsf{pos2}(x,i) := \mathsf{blh}\big((\mu z < x)(z \mathbin{\widehat{2}} 2 \sqsubseteq_\beta x \wedge \mathsf{count2}(z) = i)\big), \tag{4.79}$$

$$\mathsf{streak1}(x,i) := \mathsf{pos2}\big(x, \mathsf{Sc}(i)\big) \dot{-} \mathsf{Sc}\big(\mathsf{pos2}(x,i)\big), \tag{4.80}$$

zadovoljavaju tražene specifikacije.

Za $i < \mathsf{count2}(x)$, 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 ($\mathsf{sprefix}$ 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

$$\begin{aligned} i = \mathsf{count2}(z') = \mathsf{count2}(\langle v' \rangle) \geq \mathsf{count2}(\langle v/ \rangle) = \mathsf{count2}(\langle v \rangle \mathbin{\widehat{2}} \langle / \rangle) = \\ = \mathsf{count2}(\langle v \rangle \mathbin{\widehat{2}} 2) = \mathsf{count2}(\langle v \rangle) + 1 = \mathsf{count2}(z) + 1 = i + 1, \end{aligned} \tag{4.81}$$

kontradikcija. No kako je z jedinstven, dovoljno je naći *neki* takav i taj će biti minimalan, a $\mathsf{sprefix}$ 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 $i \geq \mathsf{count2}(x)$, takav z ne postoji (jer $z \mathbin{\widehat{2}} 2$ ima više dvojki nego x, pa ne može biti $z \mathbin{\widehat{2}} 2 \sqsubseteq_\beta x$), što znači da minimizacija $(\mu z < x)$ dade x, odnosno $\mathsf{pos2}(x,i) = \mathsf{blh}(x)$.

Sada je za $\mathsf{streak1}$ dovoljno oduzeti odgovarajuće pozicije: poziciju prvog znaka nakon i-te dvojke, od pozicije sljedeće dvojke. Za $i < \mathsf{count2}(x) - 1$, to očito funkcionira: jer su jedine znamenke 1 i 2, između susjednih dvojki nalaze se samo jedinice.

Za $i = \mathsf{count2}(x) - 1$, i-ta dvojka je upravo zadnja, pa je $\mathsf{pos2}(x,i) = \mathsf{pd}\big(\mathsf{blh}(x)\big)$. Također po specifikaciji $\mathsf{pos2}$ vrijedi $\mathsf{pos2}(x,i+1) = \mathsf{blh}(x)$, pa je $\mathsf{streak1}(x,i) = \mathsf{blh}(x) \dot{-} \mathsf{Sc}\big(\mathsf{pd}(\mathsf{blh}(x))\big) = 0$ (jer je $\mathsf{Sc}\big(\mathsf{pd}(t)\big) = t^{\cdot} \geq t$). A za $i \geq \mathsf{count2}(x)$ imamo $\mathsf{pos2}(x,i+1) = \mathsf{pos2}(x,i) = \mathsf{blh}(x)$, pa je opet $\mathsf{streak1}(x,i) = \mathsf{blh}(x) \dot{-} \mathsf{Sc}\big(\mathsf{blh}(x)\big) = 0$. □

Za ekstrakciju pojedinog x_n sada trebamo uokviriti $\mathsf{bin}^k(\vec{x})$ dvojkama te pozvati „metodu" $\mathsf{streak1}$ s argumentom $n - 1$.

Propozicija 4.66: Za svaki $n \in \mathbb{N}_+$ postoji primitivno rekurzivna funkcija arg_n, takva da za sve $\vec{x} \in \mathbb{N}^+$ vrijedi $\mathsf{arg}_n\big(\mathsf{bin}(\vec{x})\big) = \langle \vec{x} \rangle[n-1]$.

Dokaz. Za svaki $n \in \mathbb{N}_+$ vrijedi $n - 1 \in \mathbb{N}$, pa definiramo

$$\mathsf{arg}_n(x) := \mathsf{streak1}(2 \mathbin{\widehat{2}} x \mathbin{\widehat{2}} 2, n-1). \tag{4.82}$$

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 $\mathsf{count2}\big(\mathsf{bin}(\vec{x})\big) = k - 1$, odnosno za $y := 2 \mathbin{\widehat{2}} \mathsf{bin}(\vec{x}) \mathbin{\widehat{2}} 2$ vrijedi $\mathsf{count2}(y) = k - 1 + 2 = k + 1$. Sada za svaki $i \in [1 \cdot\cdot k]$ vrijedi $\mathsf{arg}_i\big(\mathsf{bin}(\vec{x})\big) = \mathsf{streak1}(y, i-1)$, što je upravo x_i (recimo, za $i = 2 < k$ imat ćemo $\mathsf{streak1}(\langle /\bullet^{x_1}/\bullet^{x_2}/\cdots/\bullet^{x_k}/\rangle, 1) = \mathsf{pos2}(y,2) - \mathsf{Sc}\big(\mathsf{pos2}(y,1)\big) = (1 + x_1 + 1 + x_2) - \mathsf{Sc}(1 + x_1) = x_2$). Za $i > k$ imat ćemo $i - 1 \geq k = \mathsf{count2}(y) - 1$, pa će po lemi 4.65 biti $\mathsf{streak1}(y, i-1) = 0$, kao što i treba. □

Teorem 4.67: Za svaku parcijalno rekurzivnu funkciju f, βf je Turing-izračunljiva.

Dokaz. Označimo s $k \in \mathbb{N}_+$ mjesnost od f. Funkciju $\mathbb{N}\beta f$ možemo dobiti (iz dijagrama) kao $\mathsf{bin}^1 \circ f \circ (\mathsf{arg}_1, \ldots, \mathsf{arg}_k)$, restringiranu na $\mathcal{I}_{\mathsf{bin}^k}$. Ta slika je primitivno rekurzivna: samo treba provjeriti broj dvojki ($k - 1 \in \mathbb{N}$ je konstanta). Dakle,

$$\mathbb{N}\beta f(x) \simeq \mathsf{bin}^1\big(f\big(\mathsf{arg}_1(x), \ldots, \mathsf{arg}_k(x)\big)\big), \text{ ako je } \mathsf{count2}(x) = k - 1 \tag{4.83}$$

— iz čega slijedi, po korolaru 3.62, da je $\mathbb{N}\beta f$ parcijalno rekurzivna. Ona je RAM-izračunljiva po teoremu 2.38, pa je βf Turing-izračunljiva po teoremu 4.39. □

Kao što smo već rekli, dinamizacijom funkcijā arg_i možemo simulirati i funkcije koje primaju proizvoljan broj argumenata (*varargs*). Definiramo pomoćne primitivno rekurzivne funkcije

$$\mathsf{argc} := \mathsf{Sc} \circ \mathsf{count2}, \tag{4.84}$$

$$\mathsf{arg}(a, i) := \mathsf{streak1}(2 \mathbin{\widehat{2}} a \mathbin{\widehat{2}} 2, i). \tag{4.85}$$

Primjer 4.68: Recimo, $\mathsf{add}^{\cdots}$, koja zbraja sve svoje argumente (koliko god da ih ima), je primitivno rekurzivna, jer je funkcija zadana s

$$\mathbb{N}\beta\mathsf{add}^{\cdots}(a) = \sum_{i < \mathsf{argc}(a)} \mathsf{arg}(a, i) \tag{4.86}$$

primitivno rekurzivna po lemi 2.53 i prethodnim rezultatima. ◁

To nam neće bitno trebati u nastavku, ali zgodno je znati da imamo i tu mogućnost.

5. Neodlučivost

Dokazali smo brojne teoreme koji pokazuju ekvivalentnost različitih modela izračunljivosti. Za početak ih sve objedinimo na jednom mjestu. Najvažniji nam je onaj koji govori o brojevnim funkcijama, karakterizirajući skup $\mathcal{C}\mathrm{omp}$.

Teorem 5.1 (Teorem ekvivalencije za brojevne funkcije): Neka je $k \in \mathbb{N}_+$ te $f : \mathbb{N}^k \rightharpoonup \mathbb{N}$ brojevna funkcija mjesnosti k. Tada su sljedeće tvrdnje ekvivalentne:

(r) f je RAM-izračunljiva ($f \in \mathcal{C}\mathrm{omp}_k$);

(m) f je makro-izračunljiva (postoji makro-algoritam koji računa f);

(p) f je parcijalno rekurzivna (f je dobivena iz inicijalnih funkcija 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 već smo dokazali. Ponovimo samo glavne ideje.

(r⇒m) Ovo je trivijalno: $\mathcal{P}\mathrm{rog} \subset \mathcal{MP}\mathrm{rog}$ (svaki RAM-program je ujedno i makro-program) — napomena 1.21.

(m⇒r) Svaki makro-program $Q \in \mathcal{MP}\mathrm{rog}$ ekvivalentan je svojem spljoštenju $Q^\flat \in \mathcal{P}\mathrm{rog}$ — teorem 1.27.

(p⇒r) *Postorder*-obilaskom njene simboličke definicije, kompiliramo funkciju f u RAM-program — teorem 2.38.

(r⇒i) Ako RAM-program $P \in \mathcal{P}\mathrm{rog}$ računa f, tada je njegov kod $e := \ulcorner P \urcorner \in \mathrm{Prog}$ jedan indeks za f — propozicija 3.54.

(i⇒p) Funkcija f^k s indeksom e dobivena je kompozicijom comp_k, konstante C_e^k i koordinatnih projekcija — korolar 3.53.

(t⇒p) Funkciju f možemo dobiti kompozicijom iz parcijalno rekurzivnih funkcija blh, $\mathbb{N}\beta f$ i bin^k — teorem 4.61.

(p⇒t) Funkciju $\mathbb{N}\beta f$ možemo dobiti restrikcijom na $\mathcal{I}_{\mathrm{bin}^k}$ kompozicije funkcija bin^1, f i $\mathrm{arg}_1, \ldots, \mathrm{arg}_k$ — teorem 4.67.

Dijagramatski, usmjereni graf
$$\begin{tikzcd} \mathtt{r} \arrow[d, leftrightarrow] & \mathtt{p} \arrow[l] \arrow[r, leftrightarrow] \arrow[dl] & \mathtt{t} \\ \mathtt{m} & \mathtt{i} \arrow[u] & \end{tikzcd}$$
je jako povezan. □

Za jednomjesne funkcije imamo korolar 4.46, zgodniji zbog jednočlane abecede i bijektivnog 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 $\Sigma \neq \varnothing$ abeceda te $\varphi : \Sigma^* \rightharpoonup \Sigma^*$ 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 $(\operatorname{card}\Sigma)!$ tvrdnji — po jednu za svako kodiranje od Σ — ali sve su one ekvivalentne po korolaru 4.41.

Opet, sve bitno je već dokazano; slijedi rekapitulacija glavnih ideja.

(T⇒P) Kodirajući sve dijelove Turing-izračunavanja s riječju zadanog koda, dobijemo parcijalnu rekurzivnost funkcije $\mathbb{N}\varphi$ — teorem 4.24.

(P⇒R) Kompiliramo funkciju $\mathbb{N}\varphi$ u RAM-program — teorem 2.38.

(R⇒T) Transpiliranjem RAM-programa koji računa $\mathbb{N}\varphi$, u pet faza dobijemo Turingov stroj koji računa φ — teorem 4.39. □

Karakteristična funkcija je totalna, dakle za izračunljive jezike mora biti rekurzivna.

Teorem 5.3 (Teorem ekvivalencije za jezike): Neka je $\Sigma \neq \varnothing$ abeceda te $L \subseteq \Sigma^*$ jezik nad njom. Tada su sljedeće tvrdnje ekvivalentne:

(ρ) relacija $\langle L \rangle := \{\langle w \rangle \mid w \in L\}$ je rekurzivna;

(ω) postoji Turingov odlučitelj $(Q, \Sigma, \Gamma, \sqcup, \delta, q_0, q_a, q_r)$ koji prepoznaje L.

Dokaz. Kao i u teoremu 5.2, stavka (ρ) je zapravo $(\operatorname{card}\Sigma)!$ tvrdnji. One su ekvivalentne po tranzitivnosti, jer je svaka od njih ekvivalentna sa stavkom (ω).

($\rho \Rightarrow \omega$) Malom modifikacijom kraja konstrukcije transpiliranog Turingova stroja koji računa $\mathbb{N}^{-1}\chi_{\langle L \rangle}$ dobijemo odlučitelj — teorem 4.48.

($\omega \Rightarrow \rho$) Malom modifikacijom kodiranja stanja, funkcije prijelaza i Turing-izračunavanja odlučitelja dobijemo rekurzivnost od $\langle L \rangle$ — teorem 4.47. □

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 Turingova 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 $v\sqcup\ldots$ za $v \in \Sigma^*$, ali može se vidjeti, tehnikama iz teorije 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 pri normalnom radu 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 (tu / sigurno nije mogao prije zapisati). $\triangleleft$

Već smo rekli da domene izračunljivih brojevnih funkcija ne moraju biti izračunljive — recimo, $\mathrm{HALT} = \mathcal{D}_{\mathsf{univ}}$ nije izračunljiva, što ćemo ubrzo dokazati — 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", možemo uočiti određene obrasce u definicijama i dokazima.

Algoritam $\mathcal{A}$ je obično 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 $\mathit{start} : A \to C$, koja svakom ulazu $x \in A$ pridružuje početnu konfiguraciju s ulazom x. Imamo još zadan podskup $E \subseteq C$ završnih konfiguracija, i surjekciju $\mathit{result} : E \to B$ (često zadanu na čitavom skupu C) koja svakoj završnoj konfiguraciji pridružuje rezultat izračunavanja.

Također imamo dvomjesnu relaciju $\rightsquigarrow$ na skupu C, koja za determinističke algoritme (kakve jedino promatramo) ima funkcijsko svojstvo — pa je možemo promatrati i kao funkciju $\mathit{nextconf} : C \to C$, koja svakoj konfiguraciji stroja pridružuje sljedeću konfiguraciju po trenutnom koraku algoritma $\mathcal{A}$. Završne konfiguracije pritom ostaju nepromijenjene: restrikcija $\mathit{nextconf}$ na skup E mora biti identiteta.

Iteriranjem funkcije $\mathit{nextconf}$ na vrijednosti funkcije start dobivamo izračunavanje. Precizno, za $x \in A$ definiramo

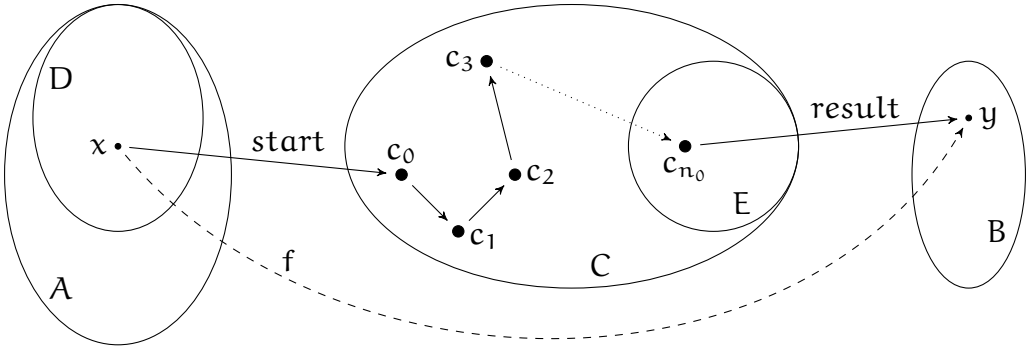
$$c_0 := \mathit{start}(x), \tag{5.1}$$

$$c_{n+1} := \mathit{nextconf}(c_n), \tag{5.2}$$

i tako smo dobili niz $(c_n)_{n\in\mathbb{N}} : \mathbb{N} \to C$ koji zovemo $\mathcal{A}$-izračunavanje s x.

Ako postoji n_0 takav da je $c_{n_0} \in E$, izračunavanje $(c_n)_{n\in\mathbb{N}}$ stane, i $y := \mathit{result}(c_{n_0})$ je njegov rezultat (to ne ovisi o izboru n_0, jer je $\mathit{nextconf}|_E = \mathrm{id}_E$, pa je završna konfiguracija

jedinstvena — kao u propoziciji 1.13). Ako s $D \subseteq A$ označimo skup svih ulaznih podataka s kojima $\mathcal{A}$-izračunavanje stane, time smo dobili funkciju $f : D \to B$ koju $\mathcal{A}$ računa.



(5.3)

Ako sad između istih skupova A i B stavimo još neki drugi skup konfiguracija C', sa svojom funkcijom $\mathit{nextconf}'$ i podskupom završnih konfiguracija E', možemo zamisliti funkciju start kao zadatak f' za taj novi stroj, uz $A' := A$ i $B' := C$. No kako taj stroj zna računati samo funkcije iz A u B, moramo prethodno kodirati C nekom funkcijom $\mathrm{Code}_1 : C \to B$. To je zapravo najzanimljiviji dio, jer o njemu ovisi koliko će laki biti kasniji koraci. Dakle $f' = \mathrm{Code}_1 \circ \mathit{start}$, 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. Lijepi uvod u teoriju ASM, pisan specifično za računarce, možete naći u [HW03]. ◁

Analogni postupak možemo napraviti za result, samo s druge strane; i za $\mathit{nextconf}$, 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 $\mathit{nextconf}$ (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 po čemu 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 metaprogramiranje (koje je nužno za univerzalnu funkciju) model u kojem radimo, pored reprezentacije ulaznih i izlaznih podataka, mora moći reprezentirati i prirodne brojeve. Tada je najjednostavnije, po principu *Occamove britve*, 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 $(C, start, nextconf, E, result)$ — 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 (u više dimenzijā!) „prostor kompromisa" (*tradeoffs*) prilikom dizajniranja takvih sustava. Na jednom kraju su konkretni sustavi za koje je odmah jasno što su konfiguracije i kako je funkcija $nextconf$ zadana — recimo, RAM-strojevi. Na drugom kraju su 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	konkretni (imperativni)	apstraktni (funkcijski)
jezični (praktični)	Turingov stroj	λ-račun
brojevni (teorijski)	RAM-stroj	parcijalna rekurzivnost

(5.4)

Tablica (5.4) je vrlo pojednostavljena — em postoji mnogo 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, primjerice geometrijski (Wangove domine) ili algebarski (diofantski skupovi), pa čak i topološki (homotopska teorija tipova); dok je „podjela" na konkretno i apstraktno 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, još apstraktniji od parcijalne rekurzivnosti, kao i Minskijevi brojači (*counter machines*), još konkretniji od RAM-strojeva. Model ASM iz

napomene 5.5 proteže se 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 „putovati" 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 precizirati kodiranje, no to je najčešće jednostavno i neovisno o tehničkim detaljima (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 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 [Lov18].)

Alonzo Church je prvi primijetio da je teza potrebna kako bi se dokazali rezultati o nepostojanju algoritma. Ipak, njegova formalizacija (λ-račun) bila je preapstraktna da bi zadovoljila vodeće eksperte za izračunljivost toga doba. To je među ostalim potaklo Turinga na detaljnu analizu 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 (najčešće programski jezici — ali i sustavi poput društvenih igrara [CBH19]) 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.*"
>
> Trahtenbrot: "*This is the way the miracle occurred: the essence of a process that can be carried out by purely mechanical means was understood and incarnated in precise mathematical definitions.*"

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 [Soa96] da se ona uobliči kao *definicija*, fiksirajući jedan koncept izračuljivosti.

Definicija 5.6 (Soareova definicija izračunljivosti): Za jezičnu funkciju φ kažemo da je *izračunljiva* ako postoji Turingov stroj koji je računa. Za brojevnu 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 ekvivalentnost 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 znanje o prirodi svjetlosti, inercijalnim sustavima i teoriji relativnosti. Bez tog znanja definicija se također može prihvatiti, ali djeluje proizvoljno. Autorovo (didaktičko, ne matematičko) uvjerenje je da bi tako djelovala i definicija 5.6 prije dokaza teorema ekvivalencije.

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 βg nije Turing-izračunljiva, po definiciji 5.6 g nije izračunljiva, pa možemo reći da za nju ne postoji algoritam.

Najčešće će g biti karakteristična funkcija, jer obično se postojanje algoritma ispituje za *probleme odluke*: za dani skup A i njegov podskup D, postoji li algoritam koji za proizvoljni $x \in A$ odlučuje 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 funkcije $\chi_{\mathbb{N}A[D]}$. Sve te ideje smo već vidjeli, za $A := \Sigma^*$, kad smo promatrali odlučive jezike u točki 4.3.

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 $\mathbb{N}A : A \to \mathbb{N}$, kažemo da je *odlučiv* ako je brojevni problem $\mathbb{N}A[D]$ odlučiv. ◁

„Brojevni problemi" nisu ništa drugo nego relacije, ali postavljeni u obliku pitanja, odnosno ispitivanja nekog svojstva. Recimo, $\mathrm{Halt}_2 = \mathcal{D}^3_{\mathsf{comp}_2}$ 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 $\{e\}(x, y)$ vrijednost?". Već smo nagovijestili da taj problem nije odlučiv. Kako bismo to dokazali? Uzmimo za početak nešto lakše. U napomeni 1.3 smo već spomenuli paradoks sličan Russellovu, koji nastaje ako pretpostavimo da su svi algoritmi totalni: poredali smo sve jednomjesne algoritme u niz $(\mathcal{A}_n)_{n\in\mathbb{N}}$, pa smo za ulaz n primijenili $\mathcal{A}_n$ na n, i vratili sljedbenik izlaznog podatka. Pomoću indeksā možemo to doslovno napraviti (zovemo ih indeksima jer je niz $(\{e\}^1)_{e\in\mathbb{N}}$ popis svih jednomjesnih parcijalno rekurzivnih funkcija, i „indeks funkcije f" je upravo njen indeks u tom nizu).

Definicija 5.8: *Russellova funkcija* je jednomjesna funkcija $\mathsf{Russell}^1$ zadana s

$$\mathsf{Russell}(n) :\simeq \mathsf{comp}_1(n, n) + 1. \tag{5.5}$$

Kleenejev skup je domena Russellove funkcije: $K^1 := \mathcal{D}_{\mathsf{Russell}}$. ◁

Korolar 5.9: Russellova funkcija je parcijalno rekurzivna.

Dokaz. Iz propozicije 3.48, jer je $\mathsf{Russell}$ kompozicija Sc, comp_1 i I^1_1. □

Paradoks iz uvoda sada se može iskazati ovako: imamo algoritam za $\mathsf{Russell}$, ali nijedan algoritam koji računa vrijednosti od $\mathsf{Russell}$ 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 $\mathsf{r}|_K = \mathsf{Russell}$.

Dokaz. Pretpostavimo da takva funkcija postoji. Tada je ona parcijalno rekurzivna, pa po teoremu ekvivalencije ima indeks, označimo jedan njen indeks s e (pogledajte napomenu 3.57); i totalna je, pa postoji $q := r(e) \in \mathbb{N}$. Sada je sljedbenik tog broja jednak

$$1 + q = 1 + r(e) = 1 + \{e\}(e) = 1 + \mathrm{comp}(e, e), \tag{5.6}$$

iz čega slijedi $e \in K$ i $\mathrm{Russell}(e) = 1 + q$. No $r(e) = q \neq 1 + q$, pa r ne može biti proširenje od Russell, kontradikcija. □

Teorem 5.11: Kleenejev skup K nije rekurzivan.

Dokaz. Pretpostavimo suprotno, i označimo s $\widetilde{\mathrm{Russell}} := \mathrm{if}\{K : \mathrm{Russell}, Z\}$ proširenje Russellove funkcije nulom. Po korolaru 5.9 Russell je parcijalno rekurzivna, po pretpostavci suprotnog je K rekurzivna relacija, a Z je inicijalna. Prema teoremu 3.60, iz toga bi slijedilo da je $\widetilde{\mathrm{Russell}}$ parcijalno rekurzivna funkcija.

S druge strane, po definiciji 2.45 je $R_0 := K^c$, pa je

$$\mathcal{D}_{\widetilde{\mathrm{Russell}}} = (\mathcal{D}_Z \cap R_0) \cup (\mathcal{D}_{\mathrm{Russell}} \cap K) = (\mathbb{N} \cap K^c) \cup (K \cap K) = K^c \cup K = \mathbb{N}, \tag{5.7}$$

dakle $\widetilde{\mathrm{Russell}}$ je totalna funkcija. Sveukupno bismo dobili da je $\widetilde{\mathrm{Russell}}$ rekurzivna funkcija, što je u kontradikciji s lemom 5.10. □

Time smo napokon dokazali da postoji izračunljiva funkcija s neizračunljivom domenom, i vidimo u čemu je problem s našom dosadašnjom 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 s ulazom x nije završna.

To je temeljna nesimetrija u definiciji algoritma: dok je zaustavljanje algoritma moguće karakterizirati neograničenom egzistencijalnom kvantifikacijom (projekcijom), što jest domena izračunljive funkcije (dobivene minimizacijom), 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 pojedinačno. Čak i da ih ne provjeravamo redom, do svakog trenutka smo provjerili samo konačno mnogo njih, među njima 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 funcijama, tada znamo da ako je g izračunljiva, tada je i f takva. To smo koristili mnogo puta u funkcijskom programiranju — recimo, pogledajte primjer 2.29. 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 gubi na korisnosti, 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, u literaturi postoje razne vrste svođenja problema, pa i općenitih funkcija. Odabiremo najjednostavniju prikladnu našim potrebama.

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 $R \leq P$, ako postoje rekurzivne funkcije $G_1^k, G_2^k, \ldots, G_l^k$ takve da vrijedi $\chi_R = \chi_P \circ (G_1, G_2, \ldots, G_l)$. $\triangleleft$

Jednakost iz definicije se može točkovno zapisati kao $R(\vec{x}) \Leftrightarrow P\big(G_1(\vec{x}), \ldots, G_l(\vec{x})\big)$. To je zapravo obična kompozicija $\chi_R = \chi_P \circ G$, samo uzevši u obzir da G ima kodomenu $\mathbb{N}^l$, pa je prikazujemo kroz l koordinatnih funkcija u skladu s napomenom 1.1.

Također naglasimo da nismo ništa dokomponirali na χ_P slijeva: ne dozvoljavamo *post-processing* povratne vrijednosti od χ_P, već samo *pre-processing* njenih ulaznih podataka. Kako se radi o karakterističnim funkcijama kodomene $\mathtt{bool} = \{0, 1\}$, jedini netrivijalni *post-processing* je negiranje, ali upravo smo vidjeli da komplement nije baš jednostavna operacija na problemima (npr. na problemima zaustavljanja), pa ga s razlogom ne želimo u ovom kontekstu.

Propozicija 5.13: Relacija $\leq$ (na skupu svih brojevnih relacija) je refleksivna i tranzitivna.

Dokaz. Za refleksivnost, stavimo $G_1 := I_1^l$, $G_2 := I_2^l$, $\ldots$, $G_l := I_l^l$ (koordinatne funkcije identitete $\mathrm{id}_{\mathbb{N}^l}$). Očito je $\chi_P \circ (I_1^l, \ldots, I_l^l) = \chi_P \circ \mathrm{id}_{\mathbb{N}^l} = \chi_P$, pa je $P \leq P$.

Tranzitivnost je kompliciranija, ali samo zbog napomene 1.1. Neka su $k, l, m \in \mathbb{N}_+$ te neka su R^k, Q^m i P^l relacije takve da je $R \leq Q \leq P$. Dakle, postoje rekurzivne funkcije $G_1^k, \ldots, G_m^k$ takve da je $\chi_R = \chi_Q \circ (G_1, \ldots, G_m)$, i rekurzivne funkcije $H_1^m, \ldots, H_l^m$ takve da je $\chi_Q = \chi_P \circ (H_1, \ldots, H_l)$. Sada je (možemo svuda pisati jednakosti jer su sve funkcije ili karakteristične ili rekurzivne, dakle totalne, a kompozicija totalnih je totalna po propoziciji 2.8)

$$\begin{aligned}
\chi_R(\vec{x}) = \chi_Q\big(G_1(\vec{x}), \ldots, G_m(\vec{x})\big) &= \big(\chi_P \circ (H_1, \ldots, H_l)\big)\big(G_1(\vec{x}), \ldots, G_m(\vec{x})\big) = \\
&= \chi_P\big(H_1\big(G_1(\vec{x}), \ldots, G_m(\vec{x})\big), \ldots, H_l\big(G_1(\vec{x}), \ldots, G_m(\vec{x})\big)\big) = \\
&= \chi_P\big(\big(H_1 \circ (G_1, \ldots, G_m)\big)(\vec{x}), \ldots, \big(H_l \circ (G_1, \ldots, G_m)\big)(\vec{x})\big) = \\
&= \big(\chi_P \circ \big(H_1 \circ (G_1, \ldots, G_m), \ldots, H_l \circ (G_1, \ldots, G_m)\big)\big)(\vec{x}), \qquad (5.8)
\end{aligned}$$

pa ako definiramo $F_i := H_i \circ (G_1, \ldots, G_m)$ za sve $i \in [1 \cdot\cdot\, l]$, sve te funkcije su rekurzivne po lemi 2.33. Sada (5.8) postaje $\chi_R = \chi_P \circ (F_1, \ldots, F_l)$, odakle slijedi $R \leq P$. $\square$

$R \leq P$ 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 $R \leq P$. To znači $\chi_R = \chi_P \circ (G_1, \ldots, G_l)$, pa je R rekurzivna po lemi 2.33. □

Korolar 5.15: Ako su R i P problemi takvi da R nije odlučiv i $R \leq P$, tada ni P nije odlučiv.

Dokaz. Ovo je samo kontrapozicija propozicije 5.14, iskazana jezikom problema. □

Korolar 5.15, kao što smo već nagovijestili, omogućuje nam da pomoću jednog neodlučivog problema dođemo do ostalih.

Propozicija 5.16: Problemi Halt_1 i HALT nisu odlučivi.

Dokaz. Tvrdimo da vrijedi $K(n) \Leftrightarrow \mathrm{Halt}_1(n, n)$. Doista,

$$K = \mathcal{D}_{\mathsf{Russell}} = \mathcal{D}_{\mathsf{Sc} \circ \mathsf{comp}_1 \circ (I_1^1, I_1^1)} = \mathcal{D}_{\mathsf{comp}_1 \circ (I_1^1, I_1^1)} = \left\{ n \,\middle|\, \left(I_1^1(n), I_1^1(n)\right) \in \mathcal{D}_{\mathsf{comp}_1} \right\} = \{ n \mid (n, n) \in \mathrm{Halt}_1 \},$$

gdje smo koristili korolar 2.7 da se riješimo sljedbenika.
Iz toga slijedi $K \leq \mathrm{Halt}_1$, pa je Halt_1 neodlučiv po korolaru 5.15 i teoremu 5.11.

Slično, po definiciji iz korolara 3.47, vrijedi $\mathrm{Halt}_1(x, e) \Leftrightarrow \mathrm{HALT}(\langle x \rangle, e)$, a funkcija Code^1 je primitivno rekurzivna po propoziciji 3.4, pa je $\mathrm{Halt}_1 \leq \mathrm{HALT}$, iz čega je i HALT neodlučiv. □

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 $\mathrm{Halt}_1(x, e) \Leftrightarrow \mathrm{Halt}_k(x, 0, 0, \ldots, 0, e)$), ali to će slijediti jednom kad dokažemo teorem o parametru.

Po Church–Turingovoj tezi, ne postoji algoritam koji bi za svaki par (P, u) RAM-programa P i njegovog ulaza u odlučivao stane li P-izračunavanje s u — odnosno, za svaki algoritam $\mathcal{A}$ postoje $P \in \mathcal{P}rog$ i $u \in \mathbb{N}$ takvi da $\mathcal{A}$ ne daje ispravan odgovor na pitanje stane li P-izračunavanje s u. Ali vrijedi 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 2.38, ona je i RAM-izračunljiva. Označimo s P_0 jedan RAM-program koji je računa. Po definiciji 1.11, 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. □

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. Jezična funkcija $\rho := \bullet\mathsf{Russell}$ je Turing-izračunljiva po korolaru 4.46, a po definiciji unarne reprezentacije vrijedi

$$\rho(\bullet^t) \simeq \bullet^{\mathsf{Russell}(t)}. \tag{5.9}$$

To znači da za $\mathcal{T}_0$ možemo uzeti Turingov stroj koji računa ρ — koji jest nad unarnom abecedom, i stane točno za riječi oblika $\bullet^t, t \in K$.

Sad, kada bi postojao algoritam koji bi za svaku riječ nad $\Sigma_\bullet$ 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. □

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 primijeniti lemu 5.18 na Turingov stroj $\mathcal{T}_0$ nad 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 posebno 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 $\Sigma \neq \varnothing$, pa α postoji), i *zamijenimo* ga s $\bullet$: gledamo Turingov stroj $\mathcal{T}_0'$ koji izgleda 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 posebno vrijedilo i za sve riječi oblika $\alpha^u, u \in \mathbb{N}$, pa bismo opet imali kontradikciju s neodlučivošću od K kao u prethodnom odlomku.

Postoji još samo jedan slučaj koji nismo uzeli u obzir: što ako je praznina stroja $\mathcal{T}_0$ upravo $\bullet$? Tada ona po definiciji nije u Σ, ali je ne možemo samo 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. □

5.3. Neodlučivost logike 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, zanimljivije je pitanje koje vodi na slavni milijun dolara vrijedan problem $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ — ali barem znamo da je valjanost odlučiva. Štoviše, dobra je vježba, koristeći kodiranje opisano u primjeru 3.23, formalno dokazati da je skup $\mathsf{PValid} \subset \mathsf{PF}$, 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.

Za logiku 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 dade 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", jer zapravo nije bitno da se radi o problemu valjanosti. Ekvivalentno možemo promatrati probleme ispunjivosti, oborivosti, proturječnosti, ili zaključivanja (logičke posljedice konačnog skupa formula), i svi su oni međusobno svedivi. Ovdje ćemo se baviti problemom koji prirodno ima oblik problema zaključivanja, pa se prvo usredotočimo na njegovu vezu s problemom valjanosti.

Definicija 5.20: Označimo s $\mathsf{F1}$ skup svih formula logike prvog reda,
a s $\mathit{Valid} \subset \mathsf{F1}$ skup svih valjanih formula među njima.
Problem valjanosti pita: za zadanu formulu $\phi \in \mathsf{F1}$, je li $\phi \in \mathit{Valid}$?
Problem zaključivanja pita: za zadane $\phi_1, \phi_2, \ldots, \phi_k, \phi \in \mathsf{F1}$,
je li ϕ logička posljedica skupa $\{\phi_1, \ldots, \phi_k\}$? $\triangleleft$

Propozicija 5.21: Problemi valjanosti i zaključivanja svedivi su jedan na drugi.

Skica dokaza. Možemo samo neformalno govoriti o svođenju, jer još nismo precizirali abecedu i jezik logike prvog reda. Taj jezik jest detaljno prikazan u [VukML09], ali nad beskonačnom abecedom, što nije dovoljno dobro za naše potrebe (pokušajte otkriti na kojim smo sve mjestima dosad koristili konačnost abecede). Uz nekoliko 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. 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 zanimljivije. Pretpostavimo da imamo algoritam za utvrđivanje valjanosti, a zadane su nam formule ϕ_1, ϕ_2, ..., ϕ_k i ϕ, s pitanjem 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 $\vDash$ 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*):

$$\phi_1, \ldots, \phi_l, \phi_{l+1} \vdash \phi \quad \text{ako i samo ako} \quad \phi_1, \ldots, \phi_l \vdash (\phi_{l+1} \to \phi), \tag{5.10}$$

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}. Iz logike znamo da $\mathfrak{N} \vDash \phi_{l+1}$ ako i samo ako $\mathfrak{N} \vDash \overline{\phi_{l+1}}$, dakle odgovor na problem zaključivanja neće se promijeniti, a $\overline{\phi_{l+1}}$ jest rečenica, pa možemo primijeniti teorem dedukcije kao u prethodnom odlomku. □

Za dovršetak odnosno formalizaciju dokaza još bi trebalo vidjeti da je funkcija koja preslikava $\big(\phi_1, \ldots, \phi_l, \phi_{l+1}, \phi\big)$ u $\big(\phi_1, \ldots, \phi_l, (\overline{\phi_{l+1}} \to \phi)\big)$ rekurzivna (zadana s $l+1$ koordinatnih funkcija na kodovima, prvih l od kojih su inicijalne), što zasad ne možemo jer nemamo kodiranje skupa 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 Churchova 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 $k \in \mathbb{N}_+$, $P \in \mathcal{P}rog$ 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 $\Gamma \vDash \zeta$. 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?

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 završnu konfiguraciju — ne zanima nas rezultat izračunavanja, nego samo zaustavljanje, pa ζ može biti egzistencijalno kvantificirana po stanju registara, i samo fiksirati vrijednost programskog brojača, a onda ne mora ovisiti o $\vec{u}$.

Instrukcije programa P mogli bismo shvatiti kao pravila zaključivanja — recimo, 3. INC $\mathcal{R}_1$ kaže da iz formule koja predstavlja konfiguraciju $(x, y, \vec{z}, 0, \ldots, 3)$ možemo izvesti formulu koja predstavlja konfiguraciju $(x, y+1, \vec{z}, 0, \ldots, 4)$, za sve x, y i $\vec{z}$. Nažalost, standardni račun logike prvog reda (*račun predikata*) ne dopušta nam imati vlastita (nelogička) pravila zaključivanja; „osuđeni" smo na *modus ponens* i generalizaciju. Srećom, *modus ponens* je vrlo općenito pravilo, koje može simulirati ostala pravila pomoću odgovarajućih nelogičkih aksioma — primjerice, ako bismo htjeli imati pravilo kojim iz ϕ i η izvodimo θ, dovoljno je u nelogičke aksiome dodati formulu $\big(\phi \to (\eta \to \theta)\big)$. Tada možemo iz tog aksioma i ϕ izvesti $(\eta \to \theta)$, pa onda iz te formule i η izvesti θ, 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 formula može sadržavati samo konačno mnogo terma, morat ćemo se ograničiti 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 („nulu") i jedan jednomjesni funkcijski simbol („sljedbenik"). Zatvoreni termi na očit način predstavljaju prirodne brojeve: broj u predstavljamo termom dobivenim u-strukom primjenom tog funkcijskog simbola na taj konstantski simbol.

Naglasimo da *ne* promatramo logiku prvog reda s jednakošću — nećemo imati potrebu promatrati jednakost kao relacijski simbol s nekim posebnim svojstvima. Usporedba s nulom za instrukcije tipa DEC bit će sintaksna.

Formalnije: krenimo od RAM-algoritma P^k, duljine $n := n_P$ i širine $m := m_{P^k}$. Signatura σ, nad kojom ćemo graditi naše formule, imat će:

- jedan konstantski simbol o;
- jedan jednomjesni funkcijski simbol s;
- $n+1$ m-mjesnih relacijskih simbola R_0^m, R_1^m, …, R_n^m.

Za svaki $u \in \mathbb{N}$, $\overline{u}$ označava zatvoreni term $s(s(\cdots(s(o))\cdots))$ s u pojava funkcijskog simbola s: $\overline{0} = o$, $\overline{1} = s(o)$, $\overline{2} = s\big(s(o)\big)$ i tako dalje. Za sve $\vec{u} = (u_1, u_2, \ldots, u_k) \in \mathbb{N}^k$ definiramo formulu

$$\pi_{\vec{u}} := R_0(o, \overline{u_1}, \overline{u_2}, \ldots, \overline{u_k}, o, o, \ldots, o), \tag{5.11}$$

gdje nakon $\overline{u_k}$ ima još $m-k-1$ simbola o (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 $i \in [0 \cdot\cdot\, n\rangle$, definiramo formulu ι_i ovisno o tipu instrukcije I_i programa P:

▷ Ako je to i. INC $\mathcal{R}_j$, definiramo

$$\iota_i := \big(R_i(x_0, \ldots, x_{m-1}) \to R_{i+1}(x_0, \ldots, x_{j-1}, s(x_j), x_{j+1}, \ldots, x_{m-1})\big). \tag{5.12}$$

▷ Ako je to i. DEC $\mathcal{R}_j, l$, definiramo

$$\begin{aligned}\iota_i := \big(&\big(R_i(x_0, \ldots, x_{j-1}, o, x_{j+1}, \ldots, x_{m-1}) \to R_l(x_0, \ldots, x_{j-1}, o, x_{j+1}, \ldots, x_{m-1})\big) \wedge \\ &\wedge \big(R_i(x_0, \ldots, x_{j-1}, s(x_j), x_{j+1}, \ldots, x_{m-1}) \to R_{i+1}(x_0, \ldots, x_{m-1})\big)\big).\end{aligned} \tag{5.13}$$

▷ Ako je to i. GO TO l, definiramo

$$\iota_i := \big(R_i(x_0, \ldots, x_{m-1}) \to R_l(x_0, \ldots, x_{m-1})\big). \tag{5.14}$$

Definiramo skup i formulu

$$\Gamma_{\vec{u}} := \{\pi_{\vec{u}}\} \cup \{\iota_i \mid i \in [0 \cdot\cdot\, n\rangle\}, \tag{5.15}$$

$$\zeta := \exists x_0 \cdots \exists x_{m-1} R_n(x_0, \ldots, x_{m-1}). \tag{5.16}$$

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 $\mathfrak{N}$ s nosačem $\mathbb{N}$ i interpretacijom nelogičkih simbola zadanom s:

- $o^{\mathfrak{N}} := 0$,
- $s^{\mathfrak{N}} := \mathsf{Sc}$ te
- za svaki $i \in [0 \cdot\cdot n]$, $R_i^{\mathfrak{N}} := \big\{\big(c_t(\mathcal{R}_0), \ldots, c_t(\mathcal{R}_{m-1})\big) \mid t \in \mathbb{N} \wedge c_t(\mathrm{PC}) = i\big\}$.

Indukcijom po r se dokaže $\overline{r}^{\mathfrak{N}} = r$ za sve $r \in \mathbb{N}$, pa $\mathfrak{N} \vDash R_{pc}(\overline{r_0}, \overline{r_1}, \ldots, \overline{r_{m-1}})$ — odnosno $\mathfrak{N} \vDash_\nu R_{pc}(x_0, x_1, \ldots, x_{m-1})$, gdje je ν valuacija takva da je $\nu(x_j) = r_j$ za sve $j \in [0 \cdot\cdot m\rangle$ — neformalno znači da RAM-stroj s programom P i ulazom $\vec{u}$ može doći u konfiguraciju $(r_0, r_1, \ldots, r_{m-1}, 0, 0, \ldots, pc)$ (nakon t koraka izračunavanja).

Lema 5.22: Vrijedi $\mathfrak{N} \vDash \pi_{\vec{u}}$.

Dokaz. Gledajući (5.11), tražimo $t \in \mathbb{N}$ takav da je $c_t(\mathrm{PC}) = 0$ i $c_t(\mathcal{R}_j) = \begin{cases} u_j, & j\in[1\cdot\cdot k] \\ 0, & \text{inače} \end{cases}$; drugim riječima, c_t treba biti početna konfiguracija s ulazom $\vec{u}$, a ona se sigurno postiže za $t = 0$. □

Lema 5.23: Za svaki $i \in [0 \cdot\cdot n\rangle$ vrijedi $\mathfrak{N} \vDash \iota_i$.

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 (5.12). Ta formula je kondicional, pa uzmimo proizvoljnu valuaciju ν i pretpostavimo da u $\mathfrak{N}$ uz valuaciju ν vrijedi njen antecedens: $\mathfrak{N} \vDash_\nu R_i(x_0, \ldots, x_{m-1})$. To znači da postoji t takav da je $c_t = \big(\nu(x_0), \ldots, \nu(x_{m-1}), 0, \ldots, i\big)$, no takva konfiguracija po definiciji 1.9(2) prelazi u

$$\big(\nu(x_0), \ldots, \nu(x_{j-1}), \nu(x_j) + 1, \nu(x_{j+1}), \ldots, \nu(x_{m-1}), 0, \ldots, i + 1\big). \tag{5.17}$$

S druge strane, $c_t \rightsquigarrow c_{t+1}$ po definiciji 1.11, pa po lemi 1.10 imamo da je (5.17) upravo c_{t+1} (to ćemo koristiti i kasnije). Drugim riječima, stroj može postići konfiguraciju (5.17), iz čega slijedi (jer je $\nu(x_j) + 1 = \mathsf{Sc}\big(\nu(x_j)\big) = s^{\mathfrak{N}}\big(\nu(x_j)\big) = \big(s(x_j)\big)^{\mathfrak{N}}[\nu]$)

$$\mathfrak{N} \vDash_\nu R_{i+1}\big(x_0, \ldots, x_{j-1}, s(x_j), x_{j+1}, \ldots, x_{m-1}\big), \tag{5.18}$$

odnosno u $\mathfrak{N}$ uz valuaciju ν vrijedi i konzekvens formule (5.12), 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 $\mathfrak{N}$ (uz proizvoljnu valuaciju ν) vrijedi formula (5.13). Ta formula je konjunkcija dva kondicionala, pa trebamo dokazati da vrijede oba, sličnom metodom kao u prethodnom odlomku. Za prvi, pretpostavimo da vrijedi $\mathfrak{N} \vDash_\nu R_i(x_0, \ldots, x_{j-1}, o, x_{j+1}, \ldots, x_{m-1})$. To znači da postoji t takav da je $c_t = \big(\nu(x_0), \ldots, \nu(x_{j-1}), 0, \nu(x_{j+1}), \ldots, \nu(x_{m-1}), 0, \ldots, i\big)$ (jer je $o^{\mathfrak{N}}[\nu] = o^{\mathfrak{N}} = 0$), no takva konfiguracija po definiciji 1.9(4) prelazi u konfiguraciju

$$c_{t+1} = \big(\nu(x_0), \ldots, \nu(x_{j-1}), 0, \nu(x_{j+1}), \ldots, \nu(x_{m-1}), 0, \ldots, l\big), \tag{5.19}$$

dostižnu u $t+1$ koraka, pa vrijedi $\mathfrak{N} \vDash_\nu R_l(x_0, \ldots, x_{j-1}, o, x_{j+1}, \ldots, x_{m-1})$.

Za drugi, pretpostavimo $\mathfrak{N} \vDash_\nu R_i(x_0, \ldots, x_{j-1}, s(x_j), x_{j+1}, \ldots, x_{m-1})$. Kao u INC-slučaju, $\big(s(x_j)\big)^{\mathfrak{N}}[\nu] = \nu(x_j) + 1$, što je pozitivno zbog $\nu(x_j) \in |\mathfrak{N}| = \mathbb{N}$, pa to znači da postoji t takav da je $c_t = \big(\nu(x_0), \ldots, \nu(x_{j-1}), \nu(x_j) + 1, \nu(x_{j+1}), \ldots, \nu(x_{m-1}), 0, \ldots, i\big)$. Po definiciji 1.9(3) ta konfiguracija prelazi u $c_{t+1} = \big(\nu(x_0), \ldots, \nu(x_{m-1}), 0, \ldots, i+1\big)$ te vrijedi $\mathfrak{N} \vDash_\nu R_{i+1}(x_0, \ldots, x_{m-1})$, čime smo dokazali i drugi kondicional.

Ako je instrukcija i. GO TO l za neki $l \leq n$, tada opet uzmemo proizvoljnu valuaciju ν, i trebamo dokazati da uz nju u $\mathfrak{N}$ vrijedi formula (5.14). To slijedi iz definicije 1.9(5), jer se stanje registara u ovom slučaju uopće ne mijenja, samo se vrijednost programskog brojača promijeni iz i u l. □

Propozicija 5.24: Za svaki $\vec{u} \in \mathbb{N}^k$, ako $\Gamma_{\vec{u}} \vDash \zeta$, tada P-izračunavanje s $\vec{u}$ stane.

Dokaz. Pretpostavka $\Gamma_{\vec{u}} \vDash \zeta$ 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 $\mathfrak{N}$ takva struktura, pa vrijedi $\mathfrak{N} \vDash \zeta$. Čitajući (5.16), vidimo da to znači da za svaku valuaciju ν (pa posebno, recimo, za onu koja sve varijable preslika u 0) postoji valuacija ν', koja se podudara s ν na svim individualnim varijablama $x_i, i \geq m$, takva da vrijedi

$$\big(\nu'(x_0), \ldots, \nu'(x_{m-1})\big) \in R_n^{\mathfrak{N}}. \tag{5.20}$$

Valuacije nam zapravo ovdje uopće nisu bitne — jedino što je bitno je da iz toga slijedi $R_n^{\mathfrak{N}} \neq \varnothing$, pa postoji $t \in \mathbb{N}$ takav da je $c_t(\text{PC}) = n$. No to znači da je c_t završna konfiguracija, pa P-izračunavanje s $\vec{u}$ stane nakon najviše t koraka. □

Sada nam je cilj dokazati obrat propozicije 5.24. Bitna razlika je u tome 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 (zapravo, ne možemo ih ni *iskazati* jer nemamo simbol za jednakost u teoriji), ali svejedno ćemo moći pokazati da u njoj vrijedi ζ.

Pa neka je $\mathfrak{M} = (M, \Theta, g, S_0, S_1, \ldots, S_n)$ proizvoljna σ-struktura (redom imamo $o^{\mathfrak{M}} =: \Theta \in M$, $s^{\mathfrak{M}} =: g : M \to M$, a za svaki $i \in [0 \cdot\cdot n]$, $R_i^{\mathfrak{M}} =: S_i \subseteq M^m$), i neka vrijedi $\mathfrak{M} \vDash \Gamma_{\vec{u}}$.

Neka je $(c_t)_{t \in \mathbb{N}}$ P-izračunavanje s $\vec{u}$. Za proizvoljne $t \in \mathbb{N}$ i $j \in [0 \cdot\cdot m\rangle$, označimo $a_j^t := \overline{c_t(\mathcal{R}_j)}^{\mathfrak{M}} = g\big(g(\cdots g(\Theta)\cdots)\big)$, gdje ima $c_t(\mathcal{R}_j)$ poziva funkcije g.

Lema 5.25: Za svaki $t \in \mathbb{N}$ vrijedi $\mathfrak{M} \vDash R_{c_t(\text{PC})}\big(\overline{c_t(\mathcal{R}_0)}, \ldots, \overline{c_t(\mathcal{R}_{m-1})}\big)$, odnosno $(a_0^t, a_1^t, \ldots, a_{m-1}^t) \in S_{c_t(\text{PC})}$.

Dokaz. Indukcijom po t. Za $t = 0$, tražena formula je upravo $\pi_{\vec{u}}$, koja se nalazi 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, $i := c_b(\text{PC}) < n$, pa postoji instrukcija programa P s rednim brojem i, i također je $\iota_i \in \Gamma_{\vec{u}}$, dakle $\mathfrak{M} \vDash \iota_i$.

Ako je ta instrukcija tipa INC, recimo i. INC $\mathcal{R}_j$ za neki $j < m$, tada u $\mathfrak{M}$ vrijedi (5.12), odnosno uz valuaciju $\nu(x_j) := a_j^b$, vrijedi

$$\left(a_0^b, \ldots, a_{j-1}^b, g(a_j^b), a_{j+1}^b, \ldots, a_{m-1}^b\right) \in S_{i+1}, \tag{5.21}$$

uz pretpostavku $(a_0^b, \ldots, a_{m-1}^b) \in S_i = S_{c_b(\text{PC})}$. No ta pretpostavka je upravo pretpostavka indukcije, pa onda vrijedi i (5.21). A formula (5.21) je upravo formula koja odgovara konfiguraciji c_{b+1}, jer je $i + 1 = c_b(\text{PC}) + 1 = c_{b+1}(\text{PC})$, i

$$a_j^{b+1} = \overline{c_{b+1}(\mathcal{R}_j)}^{\mathfrak{M}} = \overline{1 + c_b(\mathcal{R}_j)}^{\mathfrak{M}} = \left(s\left(\overline{c_b(\mathcal{R}_j)}\right)\right)^{\mathfrak{M}} = s^{\mathfrak{M}}\left(\overline{c_b(\mathcal{R}_j)}^{\mathfrak{M}}\right) = g(a_j^b). \tag{5.22}$$

Ako je ta instrukcija tipa GO TO, s odredištem $l \leq n$, uz istu valuaciju imamo da $\vec{a} \in S_i$ povlači $\vec{a} \in S_l$. Tvrdnja $\vec{a} \in S_i$ je pretpostavka indukcije, dok je $\vec{a} \in S_l$ upravo tvrdnja koja nam je potrebna za korak (stanje registara ostaje isto, odnosno $a_j^b = a_j^{b+1}$ za svaki $j \in [0 \cdots m\rangle$).

Ako je ta instrukcija tipa DEC, recimo DEC $\mathcal{R}_j, l$, promotrimo slučajeve s obzirom na to je li $c_b(\mathcal{R}_j) = 0$ (pogrešno je reći „s obzirom na to je li $a_j^b = \Theta$", jer nitko ne brani da bude npr. $g(\Theta) = \Theta$, odnosno $\overline{1}^{\mathfrak{M}} = \overline{0}^{\mathfrak{M}}$). 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 konjuktu u (5.13). Ako nije, tada je sigurno $c_b(\mathcal{R}_j) \geq 1$, pa promotrimo (5.13) (odnosno njen drugi konjunkt) uz valuaciju $\nu'(x_h) := \begin{cases} a_h^b = \overline{c_b(\mathcal{R}_h)}^{\mathfrak{M}}, & h \neq j \\ \overline{c_b(\mathcal{R}_j) - 1}^{\mathfrak{M}}, & h = j \end{cases}$. Uz tu valuaciju vrijedi antecedens, jer je $\left(s(x_j)\right)^{\mathfrak{M}}[\nu'] = a_j^b$ — pa onda vrijedi i konzekvens, 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. □

Propozicija 5.26: Ako P-izračunavanje s $\vec{u}$ stane, tada $\Gamma_{\vec{u}} \vDash \zeta$.

Dokaz. Pretpostavimo da izračunavanje $(c_t)_{t \in \mathbb{N}}$ stane, odnosno postoji t_0 takav da je $c_{t_0}(\text{PC}) = n$. Da bismo dokazali $\Gamma_{\vec{u}} \vDash \zeta$, uzmimo proizvoljnu σ-strukturu $\mathfrak{M}$ u kojoj vrijedi $\Gamma_{\vec{u}}$. Za tu strukturu vrijedi lema 5.25, pa posebno za $t = t_0$ vrijedi $(a_0^{t_0}, \ldots, a_{m-1}^{t_0}) \in S_n$. To znači da za svaku valuaciju ν možemo naći ν' koja se podudara s ν u svim varijablama x_j za $j \geq m$, a $\nu'(x_j) = a_j^{t_0}$ za $j < m$, takvu da vrijedi $\mathfrak{M} \vDash_{\nu'} R_n(x_0, \ldots, x_{m-1})$. No to upravo znači $\mathfrak{M} \vDash \zeta$. □

Teorem 5.27 (Churchov teorem o neodlučivosti logike prvog reda):
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 Kleenejeva skupa:

> Kao u dokazu korolara 5.17 (kompiliranjem), 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 $\Gamma_u \vDash^? \zeta$.

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 Kleenejev skup, jer za svaki $u \in \mathbb{N}$ vrijedi:

$$u \in K \iff P_0\text{-izračunavanje s } u \text{ stane} \iff (\Gamma_u \vDash \zeta) \iff (\vDash \varphi_u). \tag{5.23}$$

Prva ekvivalencija je definicija 1.11, jer je $K = \mathcal{D}_{\mathsf{Russell}}$, 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 Kleenejeva skupa a koji nisu. Po Church–Turingovoj tezi, to bi značilo da je relacija K rekurzivna, što je u kontradikciji s teoremom 5.11. □

Time je dokazan Churchov teorem, ali na neformalnoj razini — „dokazali" smo da ne postoji neformalni algoritam, pozivanjem na Church–Turingovu tezu. Možemo li bez toga?

5.3.3. Formule prvog reda kao formalni jezik

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 potvrdan. U ovom slučaju, promotrili bismo jezik $\mathtt{Valid}$ (nad nekom abecedom $\Sigma_{\log}$, dovoljno bogatom da može izraziti svaku formulu logike prvog reda) u kojem bi se nalazile samo valjane formule. Ako uspijemo dokazati da $\mathtt{Valid}$ nije rekurzivan, „odgurat" ćemo primjenu Church–Turingove teze na sam kraj dokaza, i pseudokod u dokazu teorema 5.27 moći ćemo zamijeniti pravom rekurzivnom funkcijom koja svodi K na $\mathtt{Valid}$ (preciznije $\langle \mathtt{Valid} \rangle$).

Hoćemo li time dobiti išta vrednije nego u prethodnoj točki? Da, jer imamo garanciju da smo barem svođenje napisali egzaktno — a ako prihvatimo definiciju 5.6, argument postaje sasvim matematički bez pozivanja na intuiciju pojma algoritma.

Za početak uvedimo abecedu $\Sigma_{\log}$. Koristimo pristup blizak standardnom, samo s konačnom abecedom. (To je važan uvjet: bez njega, na primjer, funkcija prijelaza ne bi bila konačna, pa bi dokaz leme 4.20 bio bitno kompliciraniji.) Recimo, uobičajeni alfabet logike prvog reda ima beskonačno mnogo individualnih varijabli x_0, x_1, x_2, ... — ali kad pogledamo kako ih doista pišemo, posebno kasnije (x_{38}, x_{904}, ...), vidimo da su sve one zapravo riječi nad 11-članom abecedom $\{x, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Drugim riječima, tradicionalni alfabet našeg jezika je beskonačan jer nije atomaran: sām se može shvatiti kao (regularni) jezik nad elementarnijom abecedom, koja jest konačna.

Taj fenomen je uobičajen u modernim programskim jezicima: imaju hijerarhiju jezičnih „slojeva", koji 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 sve su one jednostavnim pravilima sastavljene od konačno mnogo mogućih znakova. Konkretno, u programskom jeziku C imena

varijabli čine jezik nad 63-članom abecedom $\{\texttt{\_}, \texttt{a}, \texttt{b}, \texttt{c}, \ldots, \texttt{z}, \texttt{A}, \texttt{B}, \texttt{C}, \ldots, \texttt{Z}, \texttt{0}, \texttt{1}, \texttt{2}, \ldots, \texttt{9}\}$, zadan pravilom da prvi znak ne smije biti znamenka, a čitavo ime 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* pronađenih 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 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 — 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)→R(x,x))` je formula u kojoj postoje dva relacijska simbola: R_0^1 i R_0^2.

Mogli bismo dakle tako raditi, ali je dekadski zapis supskripta nespretan. Već smo prikazivali prirodne brojeve u unarnom zapisu, zašto ne i ovdje? Tradicionalna matematika poznaje zapise x, x', x'', ... — što je upravo naš prebrojiv skup individualnih varijabli, samo nad manjom abecedom $\{x, '\}$. Isti pristup upotrijebit ćemo za konstantske (c, c', c'', ...), funkcijske (f, f', ...) i relacijske (R, R', ...) simbole.

Ostatak alfabeta logike prvog reda je konačan i možemo ga ubaciti u našu abecedu, ali radi jednostavnosti uzet ćemo minimalni skup veznika i kvantifikatora $\{\neg, \rightarrow, \forall\}$ (tzv. *osnovni jezik* — pogledajte [VukML09, str. 124] za obrazloženje). Još moramo imati zarez i zagrade za konstrukciju složenih terma i atomarnih formula, a (vanjske) zagrade ćemo koristiti i pri pisanju kondicionala. Negacija i univerzalna kvantifikacija su prefiksni operatori, višeg prioriteta od kondicionala, pa im 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 barem odlučivog) skupa aksioma.

Definicija 5.28: *Logička abeceda* je zadana prvim retkom tablice

Σ_{log}	x	c	f	R	,	'	(	)	¬	→	∀	#
$\mathbb{N}\Sigma_{\text{log}}$	1	2	3	4	5	6	7	8	9	10	11	12

. (5.24)

Njeno kodiranje zadano je drugim retkom te tablice. Njena pomaknuta baza je 12. ◁

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 govori o uređaju $<$ — obično se simbol R_0^2 rezervira za nju), a ona se može u osnovnom jeziku napisati kao `∀x¬∀x'¬R'(x,x')`, pa je njen kod $\langle\varphi\rangle = (\text{B19 B169 4671 5168})_{12} = 14\,318\,611\,220\,424\,608$. ◁

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 $\Sigma_{\log}$
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')`

(5.25)

Time formula u osnovnom jeziku postaje `R(f(f(c),f(c)),f(f(c)))`, s kodom $\langle 1 + 1 = 2\rangle = (473\,7372\,8537\,2885\,3737\,2888)_{12} = 2\,544\,115\,135\,095\,560\,147\,164\,520$. ◁

Propozicija 5.31: Jezik F1 nad abecedom $\Sigma_{\log}$ 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 iz točke 4.3 vrsta su takvih automata, no postoje i jednostavniji automati, koji prepoznaju jednostavnije jezike.

Specijalno, u sintaksnoj analizi vrlo je bitna klasa *beskontekstnih* jezika, koje generiraju beskontekstne gramatike, a prepoznaju ih potisni automati. Potisni automat je specijalni slučaj (nedeterminističnog višetračnog) Turingova 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

$$\mathsf{Form} \to \mathsf{Rel}\,\mathtt{(}\,\mathsf{Terms}\,\mathtt{)} \mid \neg\mathsf{Form} \mid \mathtt{(}\,\mathsf{Form}\to\mathsf{Form}\,\mathtt{)} \mid \forall\,\mathsf{Var}\,\mathsf{Form} \tag{5.26}$$

$$\mathsf{Terms} \to \mathsf{Term} \mid \mathsf{Terms}\,\mathtt{,}\,\mathsf{Term} \tag{5.27}$$

$$\mathsf{Term} \to \mathsf{Var} \mid \mathsf{Const} \mid \mathsf{Func}\,\mathtt{(}\,\mathsf{Terms}\,\mathtt{)} \tag{5.28}$$

uz tipove tokena

$$\mathsf{Var} \to \mathtt{x} \mid \mathsf{Var}\,\mathtt{'} \tag{5.29}$$

$$\mathsf{Const} \to \mathtt{c} \mid \mathsf{Const}\,\mathtt{'} \tag{5.30}$$

$$\mathsf{Func} \to \mathtt{f} \mid \mathsf{Func}\,\mathtt{'} \tag{5.31}$$

$$\mathsf{Rel} \to \mathtt{R} \mid \mathsf{Rel}\,\mathtt{'} \tag{5.32}$$

(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 [Sip13]. □

Iz prethodne propozicije slijedi (po teoremu 4.47) da je $\langle \mathrm{F1}\rangle$ 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. U nastavku često koristimo konkatenaciju u pomaknutoj bazi 12 koju zovemo jednostavno *konkatenacijom* i u skladu s tim umjesto $\frown_{12}$ pišemo samo $\frown$. Još nam treba jedan jednostavni rezultat.

Korolar 5.32: Ako je riječ v prava podriječ od w, tada je $\langle v\rangle < \langle w\rangle$.

Dokaz. Nejednakost (4.8) kaže da manji broj ne može imati dulji zapis u pomaknutoj bazi. Dakle, kraća riječ mora imati manji kod, a očito je $|v| < |w|$. □

5.3.4. Formalna izreka Churchova teorema

Lema 5.33: 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 $\langle R_i\rangle$, primitivno su rekurzivne.

Dokaz. Zapravo je jedini zanimljivi dio vidjeti da je „potenciranje" (uzastopna konkatenacija) znaka odnosno riječi, $\mathrm{Repeat}(w, n) := w^n$, primitivno rekurzivno. Prateća funkcija te operacije može se napisati primitivnom rekurzijom i konkatenacijom:

$$\mathsf{Repeat}(x, 0) = \langle\varepsilon\rangle = 0, \tag{5.33}$$

$$\mathsf{Repeat}(x, n+1) = \mathsf{Repeat}(x, n) \frown x. \tag{5.34}$$

Sada, uz pokratu $t := \mathsf{Repeat}(6, i) = \langle \texttt{'''}\cdots\texttt{'}\rangle$, imamo $\mathsf{Var}(i) := \langle \texttt{x''}\cdots\texttt{'}\rangle = 1 \frown t$ — i analogno $\mathsf{Const}(i) := 2 \frown t$, $\mathsf{Func}(i) := 3 \frown t$ te $\mathsf{Rel}(i) := 4 \frown t$. □

Introdukciju logičkih veznika mogli bismo obavljati na sličan način, samo ih treba izraziti u osnovnom jeziku. Recimo, primitivno rekurzivna funkcija

$$\mathsf{Conjunction}(x, y) := \langle \neg\texttt{(}\rangle \frown x \frown \langle \rightarrow\neg\rangle \frown y \frown \langle \texttt{)}\rangle = 115 \frown x \frown 129 \frown y \frown 8 \tag{5.35}$$

ima svojstvo da je $\mathsf{Conjunction}(\langle\varphi\rangle, \langle\psi\rangle) = \langle\varphi \wedge \psi\rangle$ (preciznije $\langle \neg\texttt{(}\varphi \rightarrow \neg\psi\texttt{)}\rangle$ u osnovnom jeziku) za sve formule φ i ψ, pa se može shvatiti kao introdukcija konjunkcije na kodovima. Za eliminaciju možemo iskoristiti pristup grubom silom (*brute force*):

$$\mathsf{IsConjunction}(x, y, z) :\Longleftrightarrow y \in \langle \mathsf{F1}\rangle \wedge z \in \langle \mathsf{F1}\rangle \wedge x = \mathsf{Conjunction}(y, z), \tag{5.36}$$

$$\mathsf{LeftConjunct}(x) := (\mu y < x)(\exists z < x)\mathsf{IsConjunction}(x, y, z), \tag{5.37}$$

$$\mathsf{RightConjunct}(x) := (\mu z < x)(\exists y < x)\mathsf{IsConjunction}(x, y, z). \tag{5.38}$$

Koristili smo propoziciju 5.31, teorem 4.47 te korolar 5.32 za ograničavanje kvantifikacije i minimizacije. Brojni drugi trikovi takvog tipa, uključujući i sasvim netrivijalne stvari poput supstitucije terma za varijablu u formuli, mogu se vidjeti u [Smu92].

Za univerzalno zatvorenje zapravo 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 konačni *nadskup* skupa slobodnih varijabli. A kako za svaku varijablu x_n koja se pojavljuje (slobodno ili vezano) u φ vrijedi da je odgovarajuća riječ $\texttt{x''}\cdots\texttt{'}$ podriječ duljine $n+1$ od φ, slijedi da mora biti $n < n+1 \le |\varphi| = \mathsf{slh}(\langle\varphi\rangle, 12)$. Dakle, ako nam je zadan kod c formule φ, dovoljno je ispred „nalijepiti" samo one kvantifikatore $\forall x_i$ za $i < \mathsf{slh}(c, 12)$, i možemo biti sigurni da smo dobili univerzalno zatvorenje.

$$\mathsf{UnivQuant}(i) := \langle\forall x_i\rangle = \langle\forall\rangle \frown \langle x_i\rangle = 11 \frown \mathsf{Var}(i) \tag{5.39}$$

$$\mathsf{UnivQuants}(0) := \langle\varepsilon\rangle = 0 \tag{5.40}$$

$$\mathsf{UnivQuants}(i+1) := \langle\forall x_0 \forall x_1 \forall x_2 \cdots \forall x_i\rangle = \mathsf{UnivQuants}(i) \frown \mathsf{UnivQuant}(i) \tag{5.41}$$

$$\mathsf{UnivClosure}(c) := \mathsf{UnivQuants}\big(\mathsf{slh}(c, 12)\big) \frown c \tag{5.42}$$

Sada je funkcija UnivQuant primitivno rekurzivna po lemi 5.33 i propoziciji 4.57, zatim UnivQuants po propoziciji 2.22, a onda i UnivClosure po propoziciji 4.57 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 $c \mapsto \mathsf{UnivQuants}(m) \frown c$.

Pogledajmo kako u osnovnom jeziku izgleda formula φ_u iz dokaza teorema 5.27(4). To ovisi o instrukcijama programa P_0 koji računa Russellovu funkciju, ali dio koji nam je jedini bitan za svođenje je prilično malen i odnosi se samo na formulu π_u.

Preciznije, neka je $P_0 = \left[\ t.\ I_t\ \right]_{t<n}$. U Γ_u imamo formule $\iota_t, t < n$, čija se univerzalna zatvorenja obrnutim redom prebacuju na desnu stranu (5.10), gdje na početku stoji samo formula ζ. Na kraju nam na lijevoj strani ostane samo π_u (formulu na desnoj stani tada označimo s ψ), koju ne trebamo univerzalno zatvarati jer je rečenica. Formula ψ, iako ogromna, ne ovisi o u — ako konstruiramo funkciju argumenta u, njen kod će biti (vrlo velika) konstanta.

Dakle, $\varphi_u := (\pi_u \to \psi)$, gdje je ψ oblika $\big(\overline{\iota_0} \to \big(\overline{\iota_1} \to (\overline{\iota_2} \to \cdots(\overline{\iota_{n-1}} \to \zeta)\cdots)\big)\big)$. Uz standardne reprezentacije (o kao c_0 odnosno $\mathtt{c}$, s kao f_0^1 odnosno $\mathtt{f}$, a R_i kao R_i^m odnosno riječ koda $\mathsf{Rel}(i)$), možemo svaku pojedinu atomarnu formulu koja se pojavljuje u ψ zapisati u osnovnom jeziku. Od veznika su nam potrebni kondicionali (koje imamo u osnovnom jeziku, samo moramo pisati vanjske zagrade), i konjunkcije za formule oblika (5.13) — koje se mogu reprezentirati kako smo to objasnili u (5.35), ili 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. Obično je lakše koristiti ovaj drugi pristup.

Primjer 5.34: Promotrimo problem zaustavljanja P-izračunavanja s $\vec{u}$, gdje je

$$P := \begin{bmatrix} 0.\ \textsc{dec}\ \mathcal{R}_1, 1 \\ 1.\ \textsc{dec}\ \mathcal{R}_1, 1 \end{bmatrix}, \tag{5.43}$$

a $\vec{u} = (1, 2)$ (dakle, $m = 3$, a $n = 2$). Vidimo da P-izračunavanje s $\vec{u}$ ne stane:

$$(0, 1, 2, 0, \ldots, 0) \rightsquigarrow (0, 0, 2, 0, \ldots, 1) \rightsquigarrow (0, 0, 2, 0, \ldots, 1) \rightsquigarrow \cdots. \tag{5.44}$$

Jezikom logike prvog reda:

$$\iota_0^0 = \big(R_0(x_0, o, x_2) \to R_1(x_0, o, x_2)\big) \qquad \iota_0^+ = \big(R_0\big(x_0, s(x_1), x_2\big) \to R_1(x_0, x_1, x_2)\big) \tag{5.45}$$

$$\iota_1^0 = \big(R_1(x_0, o, x_2) \to R_1(x_0, o, x_2)\big) \qquad \iota_1^+ = \big(R_1\big(x_0, s(x_1), x_2\big) \to R_2(x_0, x_1, x_2)\big) \tag{5.46}$$

$$\zeta = \exists x_0 \exists x_1 \exists x_2 R_2(x_0, x_1, x_2) \qquad \psi = \big(\iota_0^0 \to \big(\iota_0^+ \to (\iota_1^0 \to (\iota_1^+ \to \zeta))\big)\big) \tag{5.47}$$

$$\pi_{(1,2)} = R_0\big(o, s(o), s\big(s(o)\big)\big) \qquad \varphi_{(1,2)} = (\pi_{(1,2)} \to \psi) \tag{5.48}$$

odnosno znamo da nije valjana formula (zapisana u osnovnom jeziku)

$$\begin{aligned}
\varphi_{(1,2)} = {} & \mathtt{(R(c,f(c),f(f(c)))\to} \\
& \mathtt{(\forall x \forall x' \forall x''(R(x,c,x'')\to R'(x,c,x''))\to} \\
& \mathtt{(\forall x \forall x' \forall x''(R(x,f(x'),x'')\to R'(x,x',x''))\to} \\
& \mathtt{(\forall x \forall x' \forall x''(R'(x,c,x'')\to R'(x,c,x''))\to} \\
& \mathtt{(\forall x \forall x' \forall x''(R'(x,f(x'),x'')\to R''(x,x',x''))\to} \\
& \mathtt{\neg \forall x \forall x' \forall x'' \neg R''(x,x',x''))))))},
\end{aligned} \tag{5.49}$$

čiji kod ima dvjestotinjak znamenaka — ali o $\vec{u}$ ovisi samo prvi red; sve ostalo je konstanta za fiksni RAM-algoritam P^k. ◁

Lema 5.35: Funkcija Phi, koja svakom prirodnom broju u pridružuje kod formule φ_u iz dokaza teorema 5.27 (4), primitivno je rekurzivna.

Dokaz. Prvo, označimo s $\mathtt{psi} := \langle\psi\rangle$ kod formule ψ koja nastane prebacivanjem svih $\overline{\iota_i}$ na desnu stranu relacije $\vDash$. 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 $u \mapsto \langle\overline{u}\rangle$, primitivno rekurzivna.

$$\mathsf{Numeral}(u) := \langle \mathtt{f(f(\cdots f(c)\cdots))}\rangle = \mathsf{Repeat}(43, u) \frown 2 \frown \mathsf{Repeat}(8, u) \tag{5.50}$$

$$\begin{aligned}\mathsf{Pi}(u) := \langle\pi_u\rangle &= \langle R_0(o, \overline{u}, o, \ldots, o)\rangle = \langle \mathtt{R(c,}\rangle \frown \langle\overline{u}\rangle \frown \langle(\mathtt{,c})^{m-2}\rangle \frown \langle\mathtt{)}\rangle = \\ &= 7949 \frown \mathsf{Numeral}(u) \frown \mathsf{Repeat}(62, m-2) \frown 8\end{aligned} \tag{5.51}$$

$$\mathsf{Phi}(u) := \langle\varphi_u\rangle = \langle(\pi_u \to \psi)\rangle = 7 \frown \mathsf{Pi}(u) \frown 10 \frown \mathtt{psi} \frown 8 \tag{5.52}$$

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. Treba nam i za Numeral iz istog razloga: trotočke prije i nakon `f(c)`. □

Propozicija 5.36 (Churchov teorem formalno): Jezik $\mathcal{V}alid$ nije odlučiv.

Dokaz. Lanac ekvivalencija (5.23) sada možemo proširiti još trima karikama:

$$\begin{aligned}u \in K \iff P_0\text{-izračunavanje s } u \text{ stane} \iff (\Gamma_u \vDash \zeta) \iff (\vDash \varphi_u) \iff \\ \iff \varphi_u \in \mathcal{V}alid \iff \langle\varphi_u\rangle \in \langle\mathcal{V}alid\rangle \iff \mathsf{Phi}(u) \in \langle\mathcal{V}alid\rangle.\end{aligned} \tag{5.53}$$

Predpredzadnja ekvivalencija je definicija skupa $\mathcal{V}alid$ (u donjem redu je φ_u zapisana u osnovnom jeziku). Predzadnja ekvivalencija je u jednom smjeru definicija (4.58), a u drugom injektivnost kodiranja (propozicija 4.14): ako je $\langle\varphi_u\rangle = \langle\varphi'\rangle$ za neku $\varphi' \in \mathcal{V}alid$, tada je $\varphi_u = \varphi' \in \mathcal{V}alid$. Zadnja ekvivalencija je definicija funkcije Phi.

Čitav se lanac (5.53) može sažeti u $\chi_K = \chi_{\langle\mathcal{V}alid\rangle} \circ \mathsf{Phi}$, gdje je Phi (čak primitivno, po lemi 5.35) rekurzivna, pa vrijedi $K \leq \langle\mathcal{V}alid\rangle$. Kada bi $\mathcal{V}alid$ bio odlučiv, po teoremu 4.47 bi jednomjesna relacija $\langle\mathcal{V}alid\rangle$ bila rekurzivna. Prema propoziciji 5.14, tada bi i Kleenejev skup bio rekurzivan, što je kontradikcija s teoremom 5.11. □

5.4. Prema Gödelovu prvom teoremu nepotpunosti

Argumentima potrebnim za dokaz Churchova teorema već smo se poprilično približili dokazu Gödelova prvog teorema, u formulaciji Tarskog: **Postoji istinita rečenica na prirodnim brojevima, nedokaziva u PA**. Za potpuni dokaz trebalo bi razraditi još mnogo tehničkih detalja, ali pogledajmo samo osnovnu ideju.

U prethodnoj točki uveli smo, za svaki prirodni broj u, rečenicu φ_u koja je valjana ako i samo ako je u element Kleenejeva skupa. Također smo uveli strukturu $\mathfrak{N} = (\mathbb{N}, 0, \mathsf{Sc}, \ldots)$, čiji konstantsko-funkcijski fragment se podudara sa standardnim modelom od PA (prirodni brojevi, nula i sljedbenik).

Definicija 5.37: *Gödelov skup* je skup rečenicā $\mathcal{G}\ddot{o} := \{\neg\varphi_u \mid u \in K^c\}$. ◁

Propozicija 5.38: Vrijedi $\mathfrak{N} \vDash \mathcal{G}\ddot{o}$.

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 $\mathfrak{N} \nvDash \varphi_u$ povlači $\mathfrak{N} \vDash \neg\varphi_u$, jer je φ_u rečenica. Drugim riječima, za svaki $u \in K^c$ vrijedi $\mathfrak{N} \vDash \neg\varphi_u$, što smo trebali dokazati. □

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 $\Sigma_{\log}$,

$$\mathrm{Proof} := \{\, \psi_0 \# \psi_1 \# \cdots \# \psi_n \mid (\psi_0, \psi_1, \ldots, \psi_n) \text{ je dokaz u logici prvog reda} \,\}, \tag{5.54}$$

odlučiv. To sigurno nećemo napraviti u potpunosti, ali sve osnovne ideje već imamo. Prvo, slično kao u točki 4.4.2, možemo napraviti funkcije arg'_i, koje iz kodiranog #-separiranog niza formula ekstrahiraju pojedinu formulu (zapravo njen kod). Dinamizacijom možemo napraviti i $(\mathsf{arg}')^2$, pomoću koje onda možemo govoriti 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 Hilbertova računa predikata), ili pak mora biti dobivena modus ponensom ili generalizacijom iz neke prethodne formule. Uz pokrate φ_i za $\mathsf{arg}'(p, i)$ te analogno φ_j i φ_k imamo

$$\begin{aligned} p \in \langle \mathrm{Proof} \rangle \iff & \big(\forall i < 2 + \big(\#j < \mathsf{slh}(p, 12)\big)\big(\mathsf{sdigit}(p, j, 12) = 12\big)\big)\big(\varphi_i \in \langle \mathrm{F1} \rangle \wedge \\ & \wedge \big(\varphi_i \in \textstyle\bigcup_{j=1}^{5} \langle \mathrm{Axiom}_j \rangle \vee (\exists j < i)(\exists k < i)\mathsf{ModPon}(\varphi_i, \varphi_j, \varphi_k) \vee (\exists j < i)\mathsf{Gen}(\varphi_i, \varphi_j)\big)\big) \end{aligned}$$

Sheme (A1), (A2) i (A3) možemo opisati konkatenacijom, uz ograničavanje kvantifikacije pomoću korolara 5.32. Opisat ćemo (A1) — (A2) i (A3) su dulje ali ne bitno kompliciranije.

$$f \in \langle \mathrm{Axiom}_1 \rangle :\iff (\exists a < f)\,(\exists b < f)\,\big(a \in \langle \mathrm{F1} \rangle \wedge b \in \langle \mathrm{F1} \rangle \wedge f = \langle (\rangle \frown a \frown \langle \to (\rangle \frown b \frown \langle \to \rangle \frown a \frown \langle)) \rangle\big)$$

Za sheme (A4) i (A5) moramo još voditi računa o reprezentaciji terma, i uvjetima za supstituciju — što uvodi određene tehničke poteškoće, ali sve su one rješive našim alatima razvijenima u točki 2.4.2. Za alternativnu aksiomatizaciju logike prvog reda koja izbjegava brojne tehničke poteškoće pogledajte [Smu92].

Modus ponens i generalizaciju je jednostavno zapisati.

$$\mathsf{ModPon}(a, b, c) :\iff b = \langle (\rangle \frown c \frown \langle \to \rangle \frown a \frown \langle) \rangle \tag{5.55}$$

$$\mathsf{Gen}(a, b) :\iff (\exists k < a)(a = \mathsf{UnivQuant}(k) \frown b) \tag{5.56}$$

Odnosno, α je dobivena modus ponensom iz β i γ ako i samo ako je $\beta = (\gamma \to \alpha)$. Također, α je dobivena generalizacijom iz β ako postoji k takav da je $\alpha = \forall x_k \beta$. Pri tome je $k < k + 1 = |x_k| < |\forall x_k| < |\forall x_k \beta| = |\alpha| \leq \langle \alpha \rangle = a$ po lemi 4.35, pa možemo ograničiti kvantifikaciju po k.

Kad smo se koliko-toliko uvjerili da je Proof odlučiv, trebamo isto provesti za $\mathrm{Proof}_{\mathrm{PA}}$, jezik koji sadrži dokaze u Peanovoj aritmetici. Nije tako strašno: među aksiome treba dodati i nelogičke, Peanove aksiome, i još 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, i to je jedna od spomenutih tehničkih poteškoća elegantno izbjegnutih u [Smu92].

Promotrimo 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 $\neg\varphi'_u$). Zatim pokrenemo dvije dretve.
>
> U prvoj 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 $p \in \mathrm{Proof}_{PA}$, nađemo zadnju pojavu znaka # u p, i dio p od te pojave (isključivo) do kraja usporedimo s formulom $\neg\varphi'_u$ dobivenom na početku. Ako su jednake, prekidamo program i vraćamo *false* (0).

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 $p \in \mathrm{Proof}_{PA}$ 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 $\neg\varphi'_u$ (koja jest istinita po propoziciji 5.38). To bi značilo da za sve $u \notin K$, naš algoritam također stane i vraća 0.

Sve u svemu, vidimo da neformalni algoritam koji smo napisali računa χ_K, što je po Church–Turingovoj tezi u kontradikciji s teoremom 5.11.

Jedina pretpostavka koja može biti pogrešna je da PA dokazuje sve formule oblika $\neg\varphi'_u, u \in K^c$ — drugim riječima, postoji formula iz $\mathcal{G}$ö koja nije dokaziva u PA. □

5.4.1. Problemi u skici dokaza Gödelova teorema

U dokazu teorema 5.39 mnogo je toga napravljeno neformalno. Idejno smo veoma blizu, ali tehnička zahtjevnost takvog rezultata je ogromna. Samo navedimo neke najočitije rupe, i ukratko opišimo kako ih možemo „zakrpati".

- Možemo li doista relacije R_n izraziti pomoću nule, sljedbenika, zbrajanja i množenja? Koristeći funkciju Reg svaku od njih možemo izraziti kao projekciju primitivno rekurzivne relacije, ali kodiranje potrebno za Reg bitno koristi potenciranje. Pokazuje se [Smu92] da je Gödelov teorem mnogo lakše 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 (funkcija β i kineski teorem o ostacima) je zapetljan, ali do danas nismo otkrili ništa bolje.

- Kako iz činjenice da je skup $\langle \mathrm{Proof}_{\mathrm{PA}} \rangle$ rekurzivan možemo dobiti njegovu izračunljivu enumeraciju? Ovo možete 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). $\mathrm{Proof}_{\mathrm{PA}}$ je očito 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, 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 barem možemo zamijeniti „Hilbertovom tezom" da je intuitivni pojam dokazivanja u nekoj teoriji točno 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 potvrdan, iako konstrukcija uopće nije jednostavna. 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. Mi ćemo sličan fenomen samoreferiranja vidjeti 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.

6. Metaprogramiranje

U ovom poglavlju dokazat ćemo nekoliko rezultata sa zajedničkom idejom: algoritmi mogu preko kodiranja raditi na raznim tipovima podataka, pa tako mogu raditi i na algoritmima. Recimo, u lemi 2.10, nismo mi samo dokazali da ako su $G_1, G_2, \ldots, G_l, H \in \mathcal{C}omp$ (odgovarajućih mjesnosti), tada je i $H \circ (G_1, \ldots, G_l) \in \mathcal{C}omp$ — već smo doista konstruirali funkciju $\mathrm{compose} : \mathcal{P}rog^+ \times \mathcal{P}rog \times \mathbb{N}_+ \to \mathcal{P}rog$ koja prima RAM-programe P_{G_1}, P_{G_2}, ..., P_{G_l} i P_H što računaju odgovarajuće funkcije (i mjesnost k) te od njih sastavlja RAM-program $Q_F^\flat$ što računa njihovu kompoziciju. Funkcija $\mathrm{compose}$ (odnosno $\mathrm{compose}_k^{l+1}$, jer je parametrizirana mjesnostima) je izračunljiva, i ovdje ćemo to dokazati. Nećemo baš programirati dokaz leme 2.10, jer on ide preko spljoštenja i funkcijskog makroa, što je komplicirano za izvesti formalno — ići ćemo „zaobilaznim putom", za koji će se poslije ispostaviti da je zapravo prilično koristan.

6.1. Specijalizacija

Kao što smo već nagovijestili, počet ćemo sa *specijalizacijom*: postupkom kojim u određenom algoritmu fiksiramo jedan (zadnji) ulazni podatak na neki konkretni broj. Dosad smo vidjeli mnoge primjere specijalizacije:

- Funkcija blh iz leme 4.60 dobivena je specijalizacijom funkcije slh, fiksirajući njen zadnji (drugi) argument na 2. Kažemo da je blh 2-specijalizacija slh, i pišemo $\mathsf{blh} = \$(2, \mathsf{slh})$.
- Operacija $\frown$ je 12-specijalizacija funkcije sconcat (propozicija 4.57).
- Bazu primitivne rekurzije (2.11) možemo zapisati kao $\$(0, G \mathbin{\mathrm{PR}} H) = G$.
- Prema (3.49), $\mathsf{result} = \$(0, \mathsf{ex})$.
- 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.

$$G_3 = \$(0, \mathsf{Tape}) = \$(b', b, \mathsf{Recode}) = \$\big(b', \$(b, \mathsf{Recode})\big) \tag{6.1}$$

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 ako imamo fiksnu vrijednost nekog argumenta, određeni

dio izraza možemo izračunati unaprijed i tako pojednostaviti izraz. Sličnu ideju imamo u konkretnoj matematici, povezanu s pojmom *zatvorenog oblika*: npr. iako je teško izračunati općenite vrijednosti funkcije $f(n,k) := \sum_{j=1}^{n} j^k$, za $k = 2$ dobivamo formulu $f(n,2) = \frac{n(n+1)(2n+1)}{6}$. U logičkom kontekstu toj ideji odgovara *supstitucija*, kojom 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 na mnogim mjestima, što upućuje na njenu korisnost.

Sasvim je nesporno (i već smo to dokazali u svim specijalnim slučajevima navedenima na početku ove točke) da specijalizacija čuva izračunljivost. Zbog teorema ekvivalencije, svejedno je kako 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 $g^k := \$(y, f)$, zadana s $g(\vec{x}) :\simeq f(\vec{x}, y)$, parcijalno rekurzivna.

Dokaz. Definiciju od g možemo zapisati u simboličkom obliku

$$g = f \circ (I_1^k, I_2^k, \ldots, I_k^k, C_y^k), \tag{6.2}$$

što vrijedi jer imamo $\mathcal{D}_g = \{\vec{x} \in \mathbb{N}^k \mid (\vec{x}, y) \in \mathcal{D}_f\}$ — a to 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. □

Korolar 6.2: Ako je $k \in \mathbb{N}_+$, $y \in \mathbb{N}$ te funkcija f^{k+1} (primitivno) rekurzivna,
tada je i $g^k := \$(y, f)$ (primitivno) rekurzivna.

Dokaz. Isti kao prethodni — zapravo lakši, jer ne moramo određivati domenu. □

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 eksplicitno, kako *izračunati* indeks od g pomoću indeksa od f. Mogli bismo izračunati (neke) indekse od $I_i^k, i \in [1 \cdot\cdot k]$ i C_y^k — te kad bismo imali compose kao izračunljivu funkciju na indeksima, kao posebni 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, $S = \mathbb{N}\$$, gdje funkcije kodiramo njihovim indeksima kako je detaljno objašnjeno u primjeru 3.58. Ukratko, S prima y i *bilo koji* indeks za f te vraća *neki* indeks za $\$(y, f)$.

Bilo bi sjajno kada bismo mogli dobiti S kao jednu funkciju, neovisnu o mjesnosti k funkcije f — no 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* ($\mathcal{R}_{k+1}$) u kojem će pisati ulazni podatak y u odgovarajućem RAM-stroju ovisi o k, a k ne možemo zaključiti iz samog RAM-programa odnosno njegova koda e.

Napomena 6.3: Zapravo bismo mogli 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 može poslužiti kao osnova za malo ozbiljniji studentski rad, ali je kompliciranije — a nama će ova verzija biti dovoljna. ◁

Propozicija 6.4 (Teorem o parametru): Za svaki $k \in \mathbb{N}_+$ postoji primitivno rekurzivna funkcija $S_k^2 : \mathbb{N}^2 \to \mathsf{Prog}$, takva da za sve $y, e \in \mathbb{N}$ vrijedi $\{S_k(y,e)\}^k = \$(y, \{e\}^{k+1})$.

Spomenutu funkcijsku jednakost možemo zapisati točkovno:

$$\mathsf{comp}_k\big(\vec{x}, S_k(y,e)\big) \simeq \mathsf{comp}_{k+1}(\vec{x}, y, e), \text{ za sve } \vec{x} \in \mathbb{N}^k \tag{6.3}$$

(i to je razlog za „obrnuti" redoslijed pisanja argumenata od S_k^2).

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, jedinstven zbog injektivnosti kodiranja. (Poslije ćemo vidjeti što ako nema takvog programa.) Trebamo spljoštiti makro-program

$$Q := \left[\begin{array}{c} 0.\ \textsc{inc}\ \mathcal{R}_{k+1} \\ 1.\ \textsc{inc}\ \mathcal{R}_{k+1} \\ \vdots \\ (y-1).\ \textsc{inc}\ \mathcal{R}_{k+1} \\ \hline y.\ P^* \end{array}\right] \tag{6.4}$$

od dva dijela (razdvojena crtom) — dakle kōd spljoštenja bit će dobiven konkatenacijom dva koda. Za gornji dio smo tehniku već vidjeli u primjeru 3.28 — samo umjesto konstante $\ulcorner \textsc{inc}\ \mathcal{R}_0 \urcorner = 6$ imamo konstantu $\ulcorner \textsc{inc}\ \mathcal{R}_{k+1} \urcorner = \mathsf{codeINC}(k+1) = 2 \cdot 3^{k+2}$ (k je fiksan). Dakle, prvi kod je $\overline{G}(y)$, za $G := C_{\mathsf{codeINC}(k+1)}$.

Za donji dio, primijenimo algoritam iz definicije 1.23. Kako je jedini makro 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 instrukcijā samog programa P — s rednim brojevima i odredištima pomaknutima za y. Za redne brojeve pobrinut će se konkatenacija jer je $\mathsf{lh}\big(\overline{G}(y)\big) = y$; trebamo još pomaknuti odredišta.

Kod dokazivanja primitivne rekurzivnosti konkatenacije (lema 3.18) vidjeli smo ideju „točkovne definicije konačnog niza": drugi kod će biti $\overline{H}\big(y, e, \mathsf{lh}(e)\big)$, gdje je $H(y,e,t) := \mathsf{Shift}(y, e[t])$, tako da još trebamo definirati funkciju Shift koja pomiče odredište (ako postoji) instrukcije zadane kodom (drugi argument) za y (prvi argument).

Koristeći komponente dest i regn te tipove instrukcija (leme 3.26 i 3.25), definiramo:

$$\mathsf{Shift}(y,i) := \begin{cases} \mathsf{codeDEC}\big(\mathsf{regn}(i), \mathsf{dest}(i) + y\big), & \mathsf{InsDEC}(i) \\ \mathsf{codeGOTO}\big(\mathsf{dest}(i) + y\big), & \mathsf{InsGOTO}(i)\ . \\ i, & \text{inače} \end{cases} \tag{6.5}$$

Radi totalnosti, trebamo nekako definirati Shift i kad drugi argument nije u Ins, ali jer ćemo je pozivati samo na $e[t]$ za $e \in \mathsf{Prog}$ i $t < \mathsf{lh}(e)$, zapravo je nebitno kako tamo djeluje — stavit ćemo taj slučaj zajedno sa slučajem instrukcije tipa INC, koju Shift mora ostaviti na miru. Tada je Shift primitivno rekurzivna po teoremu 2.46, pa je takva i $H^3 = \mathsf{Shift} \circ \big(I_1^3, \mathsf{part} \circ (I_2^3, I_3^3)\big)$. G je primitivno rekurzivna po propoziciji 2.21, pa su njihove povijesti $\overline{G}$ i $\overline{H}$ primitivno rekurzivne po lemi 3.15. Sada prema lemi 3.18 možemo utvrditi primitivnu rekurzivnost funkcije

$$F(y,e) := \ulcorner Q^\flat \urcorner = \overline{G}(y) * \overline{H}\big(y, e, \mathsf{lh}(e)\big) \in \mathsf{Prog} \text{ (zbog } Q^\flat \in \mathcal{P}\mathrm{rog}). \tag{6.6}$$

Još treba vidjeti da k-mjesna funkcija s tim indeksom zadovoljava jednadžbu (6.3), što možemo praćenjem po koracima: gornja razina makro programa (6.4) je sasvim sekvencijalna. Dakle, za sve $\vec{x} \in \mathbb{N}^k$ imamo

	$\mathcal{R}_0$	$\mathcal{R}_1, \ldots, \mathcal{R}_k$	$\mathcal{R}_{k+1}$	$\mathcal{R}_{k+2}..$	PC	AC
	0	$x_1, \ldots, x_k$	0	0	0	0
$0.$ INC $\mathcal{R}_{k+1}$	0	$x_1, \ldots, x_k$	1	0	1	0
$1.$ INC $\mathcal{R}_{k+1}$	0	$x_1, \ldots, x_k$	2	0	2	0
$\vdots$	0	$x_1, \ldots, x_k$	$\vdots$	0	$\vdots$	0
$(y-1).$ INC $\mathcal{R}_{k+1}$	0	$x_1, \ldots, x_k$	y	0	y	0
$y.$ P*	$\{e\}(\vec{x}, y)$	$?, \ldots, ?$	$?$	$?$	$y + 1 = n_Q$	0

, (6.7)

pa je izlazni podatak doista $\{e\}^{k+1}(\vec{x}, y) \simeq \mathsf{comp}_{k+1}(\vec{x}, y, e)$ — 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.

Još smo ostali dužni riješiti slučaj kad ne postoji program P, jer $e \notin \mathsf{Prog}$. Prema propoziciji 3.54(2) je tada $f := \{e\}^{k+1} = \otimes^{k+1}$, pa je i $\$(y, f)$ prazna — formula (2.4) kaže da funkcija (6.2) ima praznu domenu. Za povratnu vrijednost S_k^2 nam treba kod nekog RAM-programa koji računa $\otimes^k$ — uzmimo $\ulcorner[\ 0.\ \text{GO TO } 0\ ]\urcorner = \langle \mathsf{codeGOTO}(0) \rangle = 2^{25} = 33\,554\,432$. Dakle,

$$S_k(y, e) := \begin{cases} F(y, e), & \mathsf{Prog}(e) \\ 33\,554\,432, & \text{inače} \end{cases}, \tag{6.8}$$

što je primitivno rekurzivno po teoremu 2.46. □

6.1.1. Teorem o parametrima

Jednom kad imamo teorem o parametru, njegovom ponovljenom primjenom možemo specijalizirati i više zadnjih argumenata izračunljive funkcije odjednom. Primjer, ujedno i ideju dokaza, vidjeli smo u (6.1).

Korolar 6.5: Za sve $k, l \in \mathbb{N}_+$ postoji primitivno rekurzivna funkcija S_k^{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

$$\mathsf{comp}_k\big(\vec{x}, S_k(\vec{y}, e)\big) \simeq \mathsf{comp}_{k+l}(\vec{x}, \vec{y}, e). \tag{6.9}$$

Dokaz. Fiksirajmo k, i dokažimo (za taj k) tvrdnju indukcijom po l.

Kako je $l \in \mathbb{N}_+$, baza je $l = 1$, i $S_k^{l+1} = S_k^2$ je primitivno rekurzivna po propoziciji 6.4.

Pretpostavimo sad da za neki l postoji primitivno rekurzivna funkcija S_k^{l+1} sa svojstvom (6.9). Kako definirati S_k^{l+2}? Uzmemo proizvoljni $(\vec{y}, z) \in \mathbb{N}^{l+1}$ i računamo:

$$\mathsf{comp}_{k+l+1}(\vec{x}, \vec{y}, z, e) \simeq \mathsf{comp}_{k+l}\big(\vec{x}, \vec{y}, S_{k+l}(z, e)\big) \simeq \mathsf{comp}_k\big(\vec{x}, S_k\big(\vec{y}, S_{k+l}(z, e)\big)\big) \tag{6.10}$$

— iz čega slijedi da je najprirodnije definirati

$$S_k^{l+2}(\vec{y}, z, e) := S_k^{l+1}\big(\vec{y}, S_{k+l}^2(z, e)\big). \tag{6.11}$$

Dakle, S_k^{l+2} je definirana kompozicijom iz S_k^{l+1} (primitivno rekurzivne po pretpostavci indukcije), S_{k+l}^2 (primitivno rekurzivne po propoziciji 6.4) i koordinatnih projekcija (primitivno rekurzivnih 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_k^{l+1} i dokazali da je primitivno rekurzivna. □

Primjer 6.6: Supskript funkcije S govori koliko argumenata naše funkcije $\{e\}$ će „preživjeti" specijalizaciju, a prethodnik mjesnosti od S kaže koliko ih želimo fiksirati (i moramo navesti njihove vrijednosti prije e u pozivu S). Recimo, u slučaju da funkciji $\{e_1\}^7$ želimo fiksirati zadnja tri argumenta redom na brojeve 2, 8 i 5, imali bismo $e_2 := \mathsf{S}_4^4(2,8,5,e_1) = \mathsf{S}_4^2\big(2,\mathsf{S}_5^2\big(8,\mathsf{S}_6^2(5,e_1)\big)\big)$, i za sve $x,y,z,t \in \mathbb{N}$ bi vrijedilo $\{e_2\}(x,y,z,t) \simeq \{e_1\}(x,y,z,t,2,8,5)$ — odnosno $\{e_2\}^4 = \$(2,8,5,\{e_1\}^7)$. ◁

Sada možemo i vidjeti da je $\mathbb{N}\mathsf{compose}$, barem za fiksne mjesnosti k i l, primitivno rekurzivna. Tehnika koju ćemo pritom koristiti može se općenito primijeniti na razne funkcije višeg reda, definirane na funkcijama koje primaju preko indeksa.

Naglasimo 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 $\mathsf{F} := \mathsf{G} \circ \mathsf{H}$) primitivno rekurzivne. Zanima nas da su *parcijalno* rekurzivne, odnosno da imaju indekse (redom ih označimo s g, h i f). Govorimo o primitivnoj rekurzivnosti funkcije $\mathsf{compose}_1^2$ koja prima g i h te vraća f — pogledajte primjer 3.58 za detalje.

Ugrubo, *kompilirati* kompoziciju dvije funkcije koje već imamo kompilirane, je daleko „pitomije" nego *izvršiti* (evaluirati) tu kompoziciju. Izvršavanje ne mora stati ako te funkcije nisu totalne — dok sāmo kompiliranje kompozicije mora stati.

Propozicija 6.7: Za sve $k, l \in \mathbb{N}_+$ postoji primitivno rekurzivna funkcija $\mathsf{compose}_k^{l+1}$, takva da za sve $g_1, g_2, \ldots, g_l, h \in \mathbb{N}$ vrijedi

$$\{\mathsf{compose}_k(g_1,\ldots,g_l,h)\}^k = \{h\}^l \circ (\{g_1\}^k,\ldots,\{g_l\}^k). \tag{6.12}$$

Dokaz. Programirat ćemo postupak računanja kompozicije u proizvoljnom $\vec{x} \in \mathbb{N}^k$, ali ga nećemo pokrenuti — nego ćemo iz njegova indeksa specijalizacijom dobiti indeks same kompozicije.

Dakle, fiksirajmo pozitivne k i l te definirajmo funkciju F^{k+l+1} točkovno s

$$\mathsf{F}(\vec{x}^k,\vec{g}^l,h) :\simeq \mathsf{comp}_l\big(\mathsf{comp}_k(\vec{x},g_1),\ldots,\mathsf{comp}_k(\vec{x},g_l),h\big). \tag{6.13}$$

Funkcija F je definirana kompozicijom iz parcijalno rekurzivnih funkcija comp_k i comp_l te koordinatnih projekcija — pa je parcijalno rekurzivna. Po korolaru 3.56, F ima indeks; označimo jedan njen indeks s e (sjetite se napomene 3.57).

Funkcija $\mathsf{compose}_k^{l+1}$ prima g_1, …, g_l i h te na njih fiksira zadnjih $l+1$ argumenata od F:

$$\mathsf{compose}_k^{l+1}(\vec{g},h) := \mathsf{S}_k^{l+2}(\vec{g},h,e). \tag{6.14}$$

Dakle, $\mathsf{compose}_k^{l+1} = \$(e,\mathsf{S}_k^{l+2})$, 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

$$\begin{aligned}
\{\mathsf{compose}_k^{l+1}(\vec{g},h)\}(\vec{x}) &\simeq \mathsf{comp}_k\big(\vec{x},\mathsf{compose}_k(\vec{g},h)\big) \simeq \mathsf{comp}_k\big(\vec{x},\mathsf{S}_k^{l+2}(\vec{g},h,e)\big) \simeq \\
&\simeq \mathsf{comp}_{k+l+1}(\vec{x},\vec{g},h,e) \simeq \{e\}^{k+l+1}(\vec{x},\vec{g},h) \simeq \mathsf{F}(\vec{x},\vec{g},h) \simeq \\
&\simeq \mathsf{comp}_l\big(\mathsf{comp}_k(\vec{x},g_1),\ldots,\mathsf{comp}_k(\vec{x},g_l),h\big) \simeq \mathsf{comp}_l\big(\{g_1\}(\vec{x}),\ldots,\{g_l\}(\vec{x}),h\big) \simeq \\
&\simeq \{h\}\big(\{g_1\}(\vec{x}),\ldots,\{g_l\}(\vec{x})\big) \simeq \big(\{h\}^l \circ (\{g_1\}^k,\ldots,\{g_l\}^k)\big)(\vec{x}),
\end{aligned} \tag{6.15}$$

što smo trebali dokazati. □

Često se tehnika koju smo upravo koristili ne mora ponovo provoditi, već se rezultat može dobiti primjenom funkcije $\mathsf{compose}_k^{l+1}$ (za neke k i l).

Korolar 6.8: Za svaki $k \in \mathbb{N}_+$ postoji primitivno rekurzivna funkcija plus_k takva da je za sve $e, f \in \mathbb{N}$, $\mathsf{plus}_k(e, f)$ indeks zbroja funkcija $\{e\}^k + \{f\}^k$.

Dokaz. Kako je $F + G$ samo skraćeni zapis za $\mathsf{add}^2 \circ (F, G)$, sve što nam treba je odgovarajuća funkcija $\mathsf{compose}$ i neki indeks za add^2. Nije problem, istom tehnikom kao za (1.13), napisati RAM-program za add^2,

$$P_{\mathsf{add}^2} := \begin{bmatrix} 0.\ \textsc{dec}\ \mathcal{R}_1, 3 \\ 1.\ \textsc{inc}\ \mathcal{R}_0 \\ 2.\ \textsc{go to}\ 0 \\ 3.\ \textsc{dec}\ \mathcal{R}_2, 6 \\ 4.\ \textsc{inc}\ \mathcal{R}_0 \\ 5.\ \textsc{go to}\ 3 \end{bmatrix}, \tag{6.16}$$

kao niti izračunati njegov kod

$$a := \langle\langle 1, 1, 3\rangle, \langle 0, 0\rangle, \langle 2, 0\rangle, \langle 1, 2, 6\rangle, \langle 0, 0\rangle, \langle 2, 3\rangle\rangle = 2^{22\,501} \cdot 3^7 \cdot 5^{25} \cdot 7^{8\,437\,501} \cdot 11^7 \cdot 13^{649}.$$

Sada je $\mathsf{plus}_k(e, f) := \mathsf{compose}_k(e, f, a)$, odnosno $\mathsf{plus}_k^2 = \$(a, \mathsf{compose}_k^3)$ je primitivna rekurzivna po korolaru 6.2. □

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 po zlu?

Pokušajmo što vjernije slijediti postupak koji nas je doveo do funkcije $\mathsf{compose}_k^{l+1}$, za primitivnu rekurziju. Zanemarimo 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 shvatiti kao parcijalnu specifikaciju — ako ulazni podaci nisu indeksi totalnih funkcija, rezultat može biti bilo što). Želimo primitivno rekurzivnu funkciju $\mathsf{primRecurse}_k$, takvu da za sve indekse totalnih funkcija $g, h \in \mathbb{N}$ vrijedi

$$\{\mathsf{primRecurse}_k(g, h)\}^{k+1} = \{g\}^k \ \mathrm{PR}\ \{h\}^{k+2}. \tag{6.17}$$

Za $\mathsf{compose}$ smo uzeli parcijalnu jednakost (2.5), i zapisali je pomoću univerzalnih funkcija kao (6.13). Ako to učinimo s definicijom 2.11, dobit ćemo nešto poput

$$F_k(\vec{x}, y, g, h) \simeq \begin{cases} \mathsf{comp}_k(\vec{x}, g), & y = 0 \\ \mathsf{comp}_{k+2}\big(\vec{x}, \mathsf{pd}(y), \mathsf{comp}_{k+1}\big(\vec{x}, \mathsf{pd}(y), f\big), h\big), & \text{inače} \end{cases}, \tag{6.18}$$

i sad bismo mogli upotrijebiti teorem 3.60, da nije jednog malog problema: jednadžba (2.12) ima traženu funkciju i s lijeve i s desne strane. Ovaj f koji se nalazi u grani „inače”, to je

upravo $\mathrm{primRecurse}_k(g, h)$ koji želimo odrediti. Dakle, nemamo ga još, pa ga ne možemo koristiti u definiciji od F_k.

Ili možemo? Sjetimo se napomene 1.2 — f je upravo „implicitna varijabla" kakve smo rekli da trebamo prenijeti kao argumente. Dakle, na lijevoj strani (6.18) zapravo imamo $F_k(\vec{x}, y, g, h, f)$ — i po teoremu 3.60 to je parcijalno rekurzivna funkcija pa 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. Sigurno f ne možemo odrediti jednoznačno, naprosto jer nije jedinstven: $G \mathbin{\text{PR}} H$, 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* $\{f\}^{k+1} = \{g\}^{k} \mathbin{\text{PR}} \{h\}^{k+2}$ 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 općeniti oblik 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 2.24),

```
unsigned factorial(unsigned n)
    { return n ? n*factorial(n-1) : 1; }
```
(6.19)

može se točkovno zapisati kao

$$\mathsf{factorial}(n) = \begin{cases} n \cdot \mathsf{factorial}\big(\mathsf{pd}(n)\big), & n > 0 \\ 1, & \text{inače} \end{cases} \tag{6.20}$$

ili simbolički, kao $\mathsf{factorial} = \mathsf{if}\big\{\mathbb{N}_+ : \mathsf{mul}^2 \circ (\mathsf{I}_1^1, \mathsf{factorial} \circ \mathsf{pd}), \mathsf{Sc} \circ \mathsf{Z}\big\}$.

Tu jednakost nema smisla zvati *definicijom* od factorial, jer je cirkularna — ali ona 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 ne mora imati rješenje, ili ih može imati više.

Još jedan detalj treba uzeti u obzir: u (6.20) 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;
}
```
(6.21)

— ovakva h je definirana samo na parnim brojevima, i svaki preslikava u njegovu polovinu (usporedite s jezičnom funkcijom φ_h 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 u slučaju $n = 1$, C bi svejedno računao istu parcijalnu funkciju. Ipak, odgovarajuća točkovna funkcijska jednadžba

$$h(n) \simeq \begin{cases} 0, & n = 0 \\ h(n), & n = 1 \\ h(n \dot{-} 2) + 1, & \text{inače} \end{cases} \tag{6.22}$$

bi pored tog rješenja (nazovimo ga h_0) imala i razna druga, čak totalna rješenja. Jedno od njih je $h_{42}(n) := \begin{cases} h_0(n), & 2 \mid n \\ 42 + h_0(n-1), & \text{inače} \end{cases}$ — provjerite!

To samo znači ono što već znamo, da trebamo biti mnogo 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, ... koristi i funkciju F. Kao i obično, konkretnu funkciju F koja zadovoljava tu funkcijsku jednadžbu zovemo *rješenjem* te jednadžbe. Za konkretnu opću rekurziju, 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 mnogo ako ono nije jedinstveno. Recimo, vidjeli smo jednadžbu (6.22) koja ima totalno rješenje h_{42}, ali svejedno nijedan programski jezik ne bi pronašao to rješenje, već bi računao netotalnu funkciju h_0.

Kao i drugdje gdje promatramo funkcijske jednadžbe (recimo, u teoriji običnih diferencijalnih jednadžbi), rijetko se bavimo sasvim općenitim funkcijama. Kod diferencijalnih jednadžbi najčešće nas zanimaju *glatke* funkcije — ovdje, kod općih rekurzija, zanimaju nas *izračunljive* funkcije. Zbog teorema ekvivalencije, svejedno je kako manifestiramo tu izračunljivost — uglavnom ćemo to činiti kroz parcijalnu rekurzivnost ili kroz postojanje indeksa.

Dakle, zadane su nam parcijalno rekurzivne funkcije G_1, G_2, ... preko svojih indeksa g_1, g_2, ... — i tražimo indeks f funkcije F koja zadovoljava opću rekurziju $F = \mathcal{M}(\vec{G}, F)$. 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 $F(\vec{x}) \simeq \{f\}(\vec{x}) \simeq \mathrm{comp}(\vec{x}, f)$.

No ako već pišemo točkovnu definiciju, indeksi $\vec{g}$ nam ne trebaju — s obzirom na to da su G_i poznate funkcije, možemo ih uklopiti u definiciju funkcije G. Jedina funkcija s kojom to ne možemo učiniti (jer još nije 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 $\mathrm{comp}_k(\vec{y}, f)$.

(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)

$$\mathsf{comp}_k(\vec{x}, e) \simeq G(\vec{x}, e), \text{ za sve } \vec{x} \in \mathbb{N}^k. \tag{6.23}$$

Ako konkretni broj $e_0 \in \mathbb{N}$ zadovoljava tu jednadžbu, funkciju $\{e_0\}^k = \$(e_0, G)$ zovemo *rješenjem* opće rekurzije. ◁

Ponekad i indeks e_0 neformalno zovemo „rješenjem", ali jedinstvenošću uvijek smatramo jedinstvenost *funkcije*, ne indeksa. To je bitno jer inače u uobičajenim situacijama (kad G koristi argument e samo za pozivanje funkcije F) *nikad* ne bismo imali jedinstvenost: čim postoji neki e_0, svi ostali indeksi funkcije $\{e_0\}^k$ 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_0, po teoremu ekvivalencije je parcijalno rekurzivna, a takve totalne funkcije zovemo rekurzivnima.

U prethodna dva odlomka, čini se, zanemarili smo pitanje egzistencije, odnosno ponašali smo se kao da indeks e_0 koji zadovoljava (6.23) uvijek postoji. Važan i netrivijalan *teorem rekurzije* kaže da to doista vrijedi.

Naravno, netrivijalan je samo u formalnom smislu: intuitivno objašnjenje kako izvršavati opću rekurzivnu funkciju (za zadanu funkciju G) po Church–Turingovoj tezi računa parcijalno rekurzivnu funkciju. Ipak, takva intuicija nas može zavarati (prisjetite se funkcije h_{42}), 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_0 s traženim svojstvom.

6.2.1. Teorem rekurzije

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. Drugim riječima, želimo $\{e\}^k = \$(e, G)$.

Recimo, u slučaju funkcije `factorial`, kompajler se pobrine za to da poziv te funkcije u njenom kodu doista pozove nju samu (sa za 1 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 govorimo samo o specifikaciji.)

```
typedef unsigned funkcija(unsigned n);
unsigned G(unsigned n, funkcija f)
    { return n ? n*f(n-1) : 1; }
/* ... nekako definiramo factorial */
int main(void)
    { return G(5, factorial); } /* ovo vrati 120 */
```
(6.24)

Matematički, „pokazivač na funkciju" je 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 $\partial H := \$(h, H)$, gdje je h jedan (fiksni) indeks funkcije H. ◁

Kako funkcija može imati više indeksa, može imati i više dijagonalā — odnosno, formalno bismo dijagonalu trebali definirati za *indeks*, a ne za funkciju. No nama će prvenstveno biti bitna prateća funkcija $D = \mathbb{N}\partial$, koja je ionako 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 ne dopušta nikakvu aritmetiku s funkcijskim pokazivačima, ali je neki kompajleri ipak podržavaju. 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 INC ($\mathsf{InsINC}(h[0])$), 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 s kodomenom Prog, koja preslikava indeks h funkcije H^{k+1} u indeks njene dijagonale $\partial H = \$(h, H)$.

Dokaz. Teorem o parametru kaže da se indeks $D_k(h)$ za $\partial H = \$(h, \{h\}^{k+1})$ može dobiti kao $S_k(h, h) \in \mathsf{Prog}$. Drugim riječima, možemo definirati $D_k := S_k^2 \circ (I_1^1, I_1^1)$, što je primitivno rekurzivna funkcija kao kompozicija primitivno rekurzivnih. □

Dijagonaliziranje funkcije, kao što smo vidjeli, još uvijek nije dovoljno da bismo doista dobili rješenje opće rekurzije. U programu (6.24) svejedno smo trebali nekako drugačije dobiti `factorial` koju bismo poslali kao drugi argument funkciji `G`. To se čini prilično beskorisnim (jer jednom kad imamo `factorial`, možemo je odmah pozvati bez petljanja s dijagonalom), ali zapravo je vrlo blizu rješenju.

Luda ideja: funkcije `factorial` i `G`, konceptualno gledano, služe istoj svrsi — računanju faktorijele zadanog broja. Ako nemamo `factorial`, možemo li umjesto nje upotrijebiti `G` kao drugi argument od `G` (efektivno, računati ∂`G`)?

Doslovno, to neće ići — tipovi ne pašu. Drugi argument funkcije `G` trebao bi biti pokazivač na jednomjesnu funkciju, a `G` je dvomjesna. C-kompajler će nam samo dati *warning*, jer za njega su svi pokazivači ionako samo „ukrašeni prirodni brojevi" — baš kao i indeksi u svijetu parcijalno rekurzivnih funkcija — ali u strože tipiziranom jeziku kompajler bi nam odbio prevesti kod, uz poruku o grešci.

C je ovdje izuzetno zanimljiv, jer na mnogim kompajlerima, ako zanemarite *warning*, zapravo ćete dobiti ispravno rješenje (provjereno 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 kompiliranju 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).

Na procesorima s velikim brojem registara, ponekad se argumenti u funkciju male mjesnosti ne prenose preko stoga nego preko registara — ali slično razmišljanje funkcionira: poziv `f(n-1)` dekrementira registar u kojem se prenosi prvi argument, ali registar u kojem se prenosi drugi argument ostaje nepromijenjen, kao da smo pozvali `f(n-1,G)`.

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 */);
unsigned G(unsigned n, funkcija f)
    { return n ? n * f(n-1, f) : 1; }
int main(void)
    { return G(5, G); } /* vrati 120 bez warninga */
```
(6.25)

Ali ako već želimo tako zaobilaziti statičke tipove, jednostavnije je prebaciti se na dinamički tipizirani jezik. U Pythonu, stvari rade točno kako smo zamislili.

```
>>> def H(n, f): return n * f(n-1, f) if n else 1
>>> H(7, H)
5040
```
(6.26)

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): return n * d(f)(n-1) if n else 1
>>> factorial = d(H)
>>> factorial(7)
5040
```
(6.27)

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 `∂f`. Sada to možemo formalizirati, tako da definicija od `H` pozove `G` s `∂f` automatski.

```
>>> def G(n, f): return n * f(n-1) if n else 1
>>> def H(x, f): return G(x, d(f))
```
(6.28)

I to je dokaz teorema rekurzije (za $k = 1$)! Ako pažljivo pogledamo, vidjet ćemo da (6.28) nigdje ne koristi 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 $e \in \mathsf{Prog}$ 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} točkovno s $H(\vec{x}, f) :\simeq G\big(\vec{x}, D_k(f)\big)$. Ta funkcija je parcijalno rekurzivna (dobivena je kompozicijom iz G, D_k i koordinatnih projekcija) pa ima indeks: označimo jedan od njih s h (sjetite se napomene 3.57).

Tvrdimo da je ∂H rješenje opće rekurzije, odnosno njen indeks $e := D_k(h) \in \mathsf{Prog}$ je traženi broj. Doista, za sve $\vec{x} \in \mathbb{N}^k$ imamo

$$\{e\}(\vec{x}) \simeq \partial H(\vec{x}) \simeq H(\vec{x}, h) \simeq G\big(\vec{x}, D_k(h)\big) \simeq G(\vec{x}, e), \tag{6.29}$$

odnosno $\{e\}^k = \$(e, G)$, što smo trebali dokazati. □

6.2.2. Ackermannova funkcija

Sada ćemo vidjeti kako pomoću teorema rekurzije možemo definirati rekurzivne funkcije. 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 dade parcijalno rekurzivno rješenje — štoviše, dade 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 — jedinstvenost kaže da, kad god je e_1 „rješenje", za sve $\vec{x} \in \mathbb{N}^k$ vrijedi $\mathsf{comp}(\vec{x}, e_1) \simeq F_0(\vec{x})$; 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 dovoljno jednostavna, najčešće je programabilna, pa primjerice 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 $\mathsf{factorial}$).

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 ogradom. 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 *ugniježđena* 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 indukcijski 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: sljedbenik je jednomjesna funkcija, a sve ostale su dvomjesne; zato stavimo $A_0^2 := \mathsf{Sc} \circ I_2^2$. Ostale funkcije označimo na očiti način: $A_1^2 := \mathsf{add}^2$, $A_2^2 := \mathsf{mul}^2$, $A_3^2 := \mathsf{pow}$, i tako dalje. Osnovna ideja — svaka osim prve funkcije u tom nizu dobivena je iteracijom prethodne — se može iskazati kao

$$A_{n+1}(x, y+1) = A_n\big(x, A_{n+1}(x, y)\big), \text{ za sve } x, y, n \in \mathbb{N}, \tag{6.30}$$

što zapravo znači $A_{n+1} = G_{n+1} \mathbin{\mathsf{PR}} A_n \circ (I_1^3, I_3^3)$; razlikuju se samo početni uvjeti zadani s $G_n := \$(0, A_n)$. Prvih nekoliko je doista različito: $G_0(x) = \mathsf{Sc}(0) = 1$, $G_1(x) = x + 0 = x$, $G_2(x) = x \cdot 0 = 0$, $G_3(x) = x^0 = 1$ — ali svi kasniji se obično fiksiraju na 1.

Dakle, za sve $n > 1$ vrijedi $A_{n+1} := C_1^1 \mathbin{\mathsf{PR}} A_n \circ (I_1^3, I_3^3)$.

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 2.13). Za $n \geq 2$, ako je A_n primitivno rekurzivna, tada je $A_{n+1} = C_1^1 \mathbin{\mathsf{PR}} A_n \circ (I_1^3, I_3^3)$ simbolička definicija A_{n+1}, iz koje slijedi da je i ona primitivno rekurzivna. □

Primijetimo sličnost s dokazom propozicije 2.21. No dok smo dinamizirani (zadan argumentom) broj iteracija operatora $\circ$ mogli shvatiti kao primitivnu rekurziju (primjer 2.51), dinamizirani broj iteracija operatora PR više ne možemo tako shvatiti. Zato dinamizacija familije $A_n, n \in \mathbb{N}$ — tromjesna funkcija zadana s

$$A(x, y, z) := A_z(x, y) \tag{6.31}$$

— *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 PR 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 vrijednost — označimo je s t. No tada je i $A(x, y, z) = t$, dakle izraz $A(x, y, z)$ također ima vrijednost, pa je $(x, y, z) \in \mathcal{D}_{A^3}$. □

Kako je svaka A_n^2 izračunljiva, i funkcija A^3 je intuitivno izračunljiva: ako nam netko dade (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 konkretni z) znamo učiniti. Po Church–Turingovoj tezi A^3 bi trebala biti parcijalno rekurzivna, dakle rekurzivna po korolaru 6.14, ali možemo li to dokazati bez primjene Church–Turingove teze? Da — korištenjem teorema rekurzije.

Napomena 6.15: Često se pojednostavljena, dvomjesna funkcija A^2, zadana s

$$A(0, n) := n + 1, \tag{6.32}$$

$$A(m + 1, 0) := A(m, 1), \tag{6.33}$$

$$A(m + 1, n + 1) := A\big(m, A(m + 1, n)\big), \tag{6.34}$$

u literaturi navodi pod imenom „Ackermannova funkcija”, iako su je zapravo smislili Rósza 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 [Čač20] da za sve $m, n \in \mathbb{N}$ vrijedi

$$A(m, n) = A(2, n + 3, m) \dot{-} 3, \tag{6.35}$$

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 [VukIzr15, dodatak] — ukratko, svaka primitivno rekurzivna funkcija je dominirana jednim retkom tablice Ackermannove funkcije) pa kontrapozicijom slijedi da ni A^3 nije primitivno rekurzivna. ◁

6.2.3. Rekurzivnost Ackermannove funkcije

Propozicija 6.16: A^3 je rekurzivna funkcija.

Dokaz. Prvo skupimo na jedno mjesto sve jednadžbe kojima je A^3 definirana. Osnovna jednadžba je (6.30), sada u obliku (6.36), koja kazuje što činiti kad su i drugi i treći argument pozitivni. Ako je treći argument 0, imamo definiciju A_0 preko sljedbenika drugog argumenta (6.37), a u preostalim slučajevima imamo početni uvjet (6.38), jednak prvom argumentu za zbrajanje, nuli za množenje, a jedinici za ostale operacije.

$$\mathsf{A}(x, y+1, z+1) = \mathsf{A}\big(x, \mathsf{A}(x, y, z+1), z\big) \tag{6.36}$$

$$\mathsf{A}(x, y, 0) = y + 1 \tag{6.37}$$

$$\mathsf{A}(x, 0, z) = \mathsf{Astart}(x, z) := \begin{cases} x, & z = 1 \\ 0, & z = 2 \\ 1, & \text{inače} \end{cases} \tag{6.38}$$

Slučaj kad su i drugi i treći argument 0 tako je pokriven i sa (6.37) i sa (6.38), ali to nije problem, jer je po obje te jednadžbe $\mathsf{A}(x, 0, 0) = 1$.

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):

$$\mathsf{A}(x, y, z) = \begin{cases} \mathsf{A}\big(x, \mathsf{A}\big(x, \mathsf{pd}(y), z\big), \mathsf{pd}(z)\big), & y > 0 \wedge z > 0 \\ y + 1, & z = 0 \\ \mathsf{Astart}(x, z), & \text{inače} \end{cases}, \tag{6.39}$$

a zatim to napišimo u obliku opće rekurzije $\mathsf{comp}_k(x, y, z, e) = \mathsf{G}(x, y, z, e)$, gdje je $k = 3$, e je traženi indeks, a funkcija G^4 je zadana s

$$\mathsf{G}(x, y, z, e) :\simeq \begin{cases} \mathsf{comp}_3\big(x, \mathsf{comp}_3\big(x, \mathsf{pd}(y), z, e\big), \mathsf{pd}(z), e\big), & y \in \mathbb{N}_+ \wedge z \in \mathbb{N}_+ \\ \mathsf{Sc}(y), & z = 0 \\ \mathsf{Astart}(x, z), & \text{inače} \end{cases} \tag{6.40}$$

(sad smo morali upotrijebiti znak '$\simeq$', jer G nije totalna — npr. $(0, 1, 1, 0) \notin \mathcal{D}_{\mathsf{G}}$).

Po teoremu 3.60, G je parcijalno rekurzivna. Po teoremu 6.12, postoji prirodni broj a takav da za sve $x, y, z \in \mathbb{N}$ vrijedi $\{a\}(x, y, z) \simeq \mathsf{G}(x, y, z, a)$. Tvrdimo da je $\{a\}^3 = \mathsf{A}^3$, odnosno da je rješenje jedinstveno.

Točkovno, trebamo dokazati $\{a\}(x, y, z) \simeq \mathsf{A}(x, y, z)$ 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 $x, y \in \mathbb{N}$)

$$\{a\}(x, y, 0) \simeq \mathsf{G}(x, y, 0, a) = y + 1 = \mathsf{A}(x, y, 0). \tag{6.41}$$

Vanjska pretpostavka: pretpostavimo da za $z = t$, za sve $x, y \in \mathbb{N}$, vrijedi

$$\{a\}(x, y, t) \simeq \mathsf{A}(x, y, t). \tag{6.42}$$

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 $x \in \mathbb{N}$)

$$\{a\}(x, 0, t + 1) \simeq G(x, 0, t + 1, a) \simeq \mathsf{Astart}(x, t + 1), \tag{6.43}$$

što je u slučaju $t = 0$ jednako $\mathsf{Astart}(x, 1) = x = x + 0 = A_1(x, 0) = A(x, 0, 1)$,
u slučaju $t = 1$ je jednako $\mathsf{Astart}(x, 2) = 0 = x \cdot 0 = A_2(x, 0) = A(x, 0, 2)$, a
u svim ostalim slučajevima ($t \geq 2$) je jednako $\mathsf{Astart}(x, t + 1) = 1 = A_{t+1}(x, 0) = A(x, 0, t + 1)$.
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

$$\{a\}(x, s, t + 1) \simeq A(x, s, t + 1). \tag{6.44}$$

Unutarnji korak: neka je sada $y = s + 1$, 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:

$$\begin{aligned}\{a\}(x, s + 1, t + 1) &\simeq G(x, s + 1, t + 1, a) \simeq \mathsf{comp}_3\big(x, \mathsf{comp}_3\big(x, \mathsf{pd}(s + 1), t + 1, a\big), \mathsf{pd}(t + 1), a\big) \simeq \\ &\mathsf{comp}_3\big(x, \mathsf{comp}_3(x, s, t+1, a), t, a\big) \simeq \mathsf{comp}_3\big(x, \{a\}(x, s, t+1), t, a\big) \simeq \mathsf{comp}_3\big(x, A(x, s, t+1), t, a\big) \\ &\simeq \{a\}\big(x, A(x, s, t + 1), t\big) \simeq A\big(x, A(x, s, t + 1), t\big) \simeq A(x, s + 1, t + 1).\end{aligned} \tag{6.45}$$

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. □

6.3. Ekvivalentnost indeksa

Teorem rekurzije može se shvatiti na još jedan način, na koji je prvotno bio dokazan. Fenomen smo već vidjeli na početku ovog poglavlja: teorem o parametru (propozicija 6.4) se prirodno može dobiti iz propozicije 6.7, jer po (6.2) možemo staviti

$$S'_k(y, e) := \mathsf{compose}_k\big(i_1, \ldots, i_k, \overline{C_6}(y), e\big), \tag{6.46}$$

gdje je $i_n := \langle \mathsf{codeDEC}(n, 3), \mathsf{codeINC}(0), \mathsf{codeGOTO}(0)\rangle$ indeks funkcije I^k_n, a (primjer 3.28) $\overline{C_6}(y)$ indeks funkcije C^k_y — ali s obzirom na to da je lakše dokazati teorem o parametru, išli smo u suprotnom smjeru.

Slično je i ovdje: jednakost $\{e\} = \$(e, G)$ iz teorema rekurzije možemo shvatiti kao relaciju među indeksima $e \approx S_k(e, g)$, gdje je g neki indeks za G. Ta „približna jednakost" ne može doista biti jednakost ni u kojem zanimljivom slučaju, jer za $g \in \mathsf{Prog}$ 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 1.26: ono što bismo htjeli reći nije da su e i $S_k(e, g) =: F(e)$ jednaki *brojevi*, već da su $\{e\}^k$ i $\{F(e)\}^k$ jednake *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 $\{e\}^k = \{f\}^k$. ◁

Primjer 6.18: Za svaki $k \in \mathbb{N}_+$, za sve $e, f \in \mathsf{Prog}^c$ vrijedi $e \approx_k f$ (jer svi brojevi izvan Prog su indeksi prazne funkcije). Također, za svaki $e \in \mathsf{Prog}$, za sve $j, k \in \mathbb{N}_+$, vrijedi $e * \langle \mathsf{codeINC}(j) \rangle \approx_k e$ (dodavanje instrukcije INC $\mathcal{R}_j$ na kraj programa neće promijeniti uvjet zaustavljanja, a ni izlaznu vrijednost jer je $j > 0$). ◁

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 1.14 koji kaže da za fiksni k, svaki RAM-program računa jedinstvenu k-mjesnu funkciju.

Prebrojivost kvocijentnog skupa slijedi iz činjenice da postoji prebrojivo mnogo izračunljivih funkcija (teorem 1.16). Konkretno, za svaki $k \in \mathbb{N}_+$ su sve funkcije $\mathsf{C}^k_y, y \in \mathbb{N}$, različite, pa je (primjer 3.28) $\overline{\mathsf{C}_6}(y) \not\approx_k \overline{\mathsf{C}_6}(z)$ za sve $y \neq z$. Iz toga slijedi da različitih klasa ima beskonačno, a ne može ih biti neprebrojivo jer su neprazni disjunktni podskupovi od $\mathbb{N}$ (recimo, $\min : \mathbb{N}/_{\approx_k} \to \mathbb{N}$, koja svakoj klasi ekvivalencije pridružuje njen najmanji element, je injekcija, pa je $\operatorname{card}\left(\mathbb{N}/_{\approx_k}\right) \leq \operatorname{card} \mathbb{N} = \aleph_0$).

Prebrojivost svake klase slijedi iz primjera 6.18: neka je $e \in \mathbb{N}$ proizvoljan. Ako je $e \in \mathsf{Prog}^c$, tada je čitav Prog^c u njegovoj klasi. Na primjer, $n \mapsto 2n + 3$ je injekcija s $\mathbb{N}$ u $\mathsf{Seq}^c \subseteq \mathsf{Prog}^c \subseteq [e]_{\approx_k} \subseteq \mathbb{N}$, pa je $[e]_{\approx_k}$ prebrojiva. Ako je pak $e \in \mathsf{Prog}$, tada je $\mathsf{F} : \mathbb{N}_+ \to [e]_{\approx_k}$, zadana s $\mathsf{F}(j) := e * \langle \mathsf{codeINC}(j) \rangle$, injekcija — jer j možemo rekonstruirati iz $f := \mathsf{F}(j)$ kao $\mathsf{regn}\big(\mathsf{rpart}(f, 0)\big)$. □

Možda malo više iznenađujuća činjenica vezana uz familiju relacija $\approx_k, k \in \mathbb{N}_+$ je da je ona *padajuća*: vrijedi $(\approx_1) \supset (\approx_2) \supset (\approx_3) \supset \cdots$. Da bismo to dokazali, prvo trebamo jedno tehničko svojstvo funkcijā S_k.

Lema 6.20: Za svaki $e \in \mathsf{Prog}$ i za svaki $k \in \mathbb{N}_+$ vrijedi $\mathsf{S}_k(0, e) = e$.

Dokaz. Prvo, rastavom na slučajeve vidimo da je $\mathsf{Shift}(0, i) = i$ za sve $i \in \mathbb{N}$. Recimo, za $i \in \mathsf{InsDEC}$, postoje $j, l < i$ takvi da je $i = \mathsf{codeDEC}(j, l) = \langle 1, j, l \rangle$. No to znači da je $j = i[1] = \mathsf{regn}(i)$ i $l = i[2] = \mathsf{dest}(i)$, pa je $\mathsf{Shift}(0, i) = \mathsf{codeDEC}(j, l + 0) = i$. Ostali slučajevi su još lakši.

Drugo, sada je $\mathsf{H}(0, e, t) = \mathsf{Shift}(0, e[t]) = e[t] = \mathsf{part}(e, t)$, pa je desni operand operacije $*$ u (6.8) jednak $\overline{\mathsf{H}}\big(0, e, \mathsf{lh}(e)\big) = \overline{\mathsf{part}}\big(e, \mathsf{lh}(e)\big) = e$ jer iz $\mathsf{Prog}(e)$ slijedi $\mathsf{Seq}(e)$. Lijevi operand iste operacije je $\overline{\mathsf{G}}(0) = 1$ jer je svaka povijest u nuli jednaka $\langle\rangle = 1$.

I treće, to onda znači da je (za $e \in \mathsf{Prog}$) $\mathsf{S}_k(0, e) = 1 * e$, što je jednako e prema (3.21) i (3.22), jer vrijedi $\mathsf{lh}(1) = 0$ i $\mathsf{Seq}(e)$. □

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 $\{e\}^{k+l}(\vec{x}^k, \vec{0}^l) \simeq \{e\}^k(\vec{x})$.

Dokaz. Ako $e \notin \mathsf{Prog}$, tvrdnja vrijedi jer ni lijeva ni desna strana nemaju vrijednost po propoziciji 3.54(2). Za $e \in \mathsf{Prog}$, prvo indukcijom po l, prateći dokaz korolara 6.5 i koristeći lemu 6.20 u svakom koraku, dokažemo $\mathsf{S}_k^{l+1}(\vec{0}, e) = e$. Iz toga onda slijedi

$$\{e\}^{k+l}(\vec{x}, \vec{0}) \simeq \{\mathsf{S}_k^{l+1}(\vec{0}, e)\}(\vec{x}) \simeq \{e\}^k(\vec{x}), \tag{6.47}$$

što smo i trebali dokazati. □

Na neformalnoj razini tvrdnja korolara 6.21 je očita: za bilo koji RAM-program P, P-izračunavanje s $\vec{x}$ je *isto* (isti niz istih konfiguracija) kao i P-izračunavanje s $(\vec{x}, \vec{0})$, za bilo koji broj dodanih nula na kraj ulaza — jer je izračunavanje determinističnо, a početna konfiguracija $c_0 = (0, \vec{x}, 0, 0, \dots)$ je ista. Samo smo rekli da će još l registara nakon $\mathcal{R}_k$ biti postavljeno na nule, no oni bi ionako bili postavljeni na nule u izračunavanju s $\vec{x}^k$, jer u tom izračunavanju nisu ulazni.

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 $\vec{x} \in \mathbb{N}^l$,

$$\{e\}^l(\vec{x}) \simeq \{e\}^{l+m}(\vec{x}^l, \vec{0}^m) \simeq \{f\}^{l+m}(\vec{x}, \vec{0}) \simeq \{f\}^l(\vec{x}). \tag{6.48}$$

Dakle vrijedi i $\{e\}^l = \{f\}^l$, pa je $e \approx_l f$.

Za $(\neq)$, moramo naći dva RAM-programa koji jesu ekvivalentni za sve moguće l-torke ulaznih podataka, ali kad počnemo stavljati ulaze i u registre od $\mathcal{R}_{l+1}$ do $\mathcal{R}_k$, više nisu. Za to nam mogu poslužiti 1 kao indeks nulfunkcije, i i_k kao indeks k-te koordinatne projekcije — pogledajte tekst nakon (6.46) za preciznu definiciju.

Tada je $1 \approx_l i_k$, jer za svaki $\vec{x} \in \mathbb{N}^l$ vrijedi

$$\{i_k\}^l(\vec{x}) \simeq \{i_k\}^{l+m}(\vec{x}, \vec{0}) = \mathsf{I}_k^k(\vec{x}, \vec{0}) = 0 = \mathsf{C}_0^l(\vec{x}) = \{1\}^l(\vec{x}); \tag{6.49}$$

ali za $\vec{y} := (\vec{0}^{k-1}, 1) \in \mathbb{N}^k$ vrijedi

$$\{i_k\}^k(\vec{y}) = \mathsf{I}_k^k(\vec{0}, 1) = 1 \neq 0 = \mathsf{C}_0^k(\vec{y}) = \{1\}^k(\vec{y}), \tag{6.50}$$

iz čega slijedi $1 \not\approx_k i_k$. □

Kao što rekosmo na početku, indeks e iz teorema rekurzije možemo shvatiti kao neku vrstu „fiksne točke" primitivno rekurzivne funkcije F^1 zadane s $\mathsf{F}(e) := \mathsf{S}_k(e, g)$, 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. Za početak ga pokušajmo formulirati što općenitije.

Prvo, sasvim je jasno da ne možemo tražiti $e = \mathsf{F}(e)$ — već za inicijalnu $\mathsf{F} = \mathsf{Sc}$ takav e ne postoji. Ipak možemo tražiti $e \approx_k \mathsf{F}(e)$ za bilo koji fiksirani $k \in \mathbb{N}_+$ (takav e će ovisiti o k; „uniformna" verzija, koja bi imala jedan e za sve k, ne vrijedi).

Drugo, F ne mora biti primitivno rekurzivna — vidjet ćemo da je dovoljno da bude rekurzivna. Tada teorem vrijedi i za primitivno rekurzivne funkcije, zbog korolara 2.35. Napomenimo da ne možemo još oslabiti pretpostavku zahtijevajući da je F parcijalno rekurzivna: $\otimes^1 = \mu\varnothing^2$ je parcijalno rekurzivna prema primjeru 2.31, a ne može postojati $e \in \mathbb{N}$ takav da su e i $\otimes(e)$ k-ekvivalentni, jer izraz $\otimes(e)$ nema vrijednost ni za koji e.

Treće, smijemo tražiti da e bude u Prog: dobit ćemo ga iz teorema rekurzije, a on daje samo sintaksno ispravne programe (preko dijagonalne funkcije).

Četvrto, F mora biti jednomjesna: broj k je mjesnost funkcije $\{e\}^k$, jednake funkciji $\{\mathsf{F}(e)\}^k$. Mjesnost od F mora biti 1 jer želimo u nju uvrstiti $e \in \mathbb{N}^1$.

I peto, F mora biti izračunljiva: nije dovoljno zahtijevati samo totalnost. Recimo, promotrimo karakterističnu funkciju klase ekvivalencije $\mathsf{Emp} := [0]_{\approx_1}$. Kad bi postojao broj e takav da je $e \approx_1 \chi_{\mathsf{Emp}}(e)$, tada bi $e \in \mathsf{Emp}$ povlačilo s jedne strane $e \approx_1 0$, a s druge $e \approx_1 \chi_{\mathsf{Emp}}(e) = 1$, pa bi po tranzitivnosti bilo $0 \approx_1 1$, kontradikcija. (Izraz $\{0\}(0)$ nema vrijednost po propoziciji 3.54(2), a $\{1\}(0) = \mathsf{Z}(0) = 0$, pa $\{0\}(0) \not\simeq \{1\}(0)$, dakle $\{0\}^1 \neq \{1\}^1$.) No $e \notin \mathsf{Emp}$ je također kontradikcija, jer bi to značilo $e \approx_1 \chi_{\mathsf{Emp}}(e) = 0$, dakle $e \in [0]_{\approx_1} = \mathsf{Emp}$. Zaključujemo da χ_{Emp} nema fiksnu točku, iako je (kao i svaka karakteristična funkcija) totalna.

Primijetite sličnost upravo provedenog razmišljanja s Russellovim paradoksom. Iz toga će slijediti, jednom kad dokažemo teorem o fiksnoj točki, da skup Emp nije rekurzivan. No zapravo će to biti samo jedna posljedica Riceova 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 $e \in \mathsf{Prog}$ takav da je $e \approx_k \mathsf{F}(e)$.

Dokaz. Zapišimo traženi uvjet pomoću univerzalne funkcije:

$$\mathsf{comp}_k(\vec{x}, e) \simeq \mathsf{comp}_k\big(\vec{x}, \mathsf{F}(e)\big), \text{ za sve } \vec{x} \in \mathbb{N}^k. \tag{6.51}$$

To je opća rekurzija. Na desnoj strani je G^{k+1} zadana s $\mathsf{G}(\vec{x}, e) :\simeq \mathsf{comp}_k\big(\vec{x}, \mathsf{F}(e)\big)$, dakle dobivena kompozicijom iz parcijalno rekurzivne comp_k, rekurzivne F i inicijalnih koordinatnih projekcija, pa je parcijalno rekurzivna. Po teoremu 6.12, postoji $e \in \mathsf{Prog}$ koji zadovoljava tu funkcijsku jednadžbu. Jer je F rekurzivna, dakle totalna, postoji i $f := \mathsf{F}(e) \in \mathbb{N}$ (f naravno ne mora biti iz Prog). Sada se (6.51) može zapisati kao $\{e\}(\vec{x}) \simeq \{f\}(\vec{x})$ za sve $\vec{x} \in \mathbb{N}^k$, odnosno $\{e\}^k = \{f\}^k$, dakle $e \approx_k f = \mathsf{F}(e)$, što smo i trebali. □

6.4. Invarijantnost

Vidjeli smo u prethodnoj točki da iz teorema o fiksnoj točki zapravo možemo zaključiti da klasa ekvivalencije $\mathsf{Emp} = [0]_{\approx_1}$ nije rekurzivna: njena karakteristična funkcija ne može imati fiksnu točku, jer to vodi na paradoks vrlo sličan Russellovu. 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 klasā ekvivalencije nije rekurzivna, osim trivijalnih unijā $\bigcup\varnothing = \varnothing$ (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 $\{e\}^1 = \{0\}^1 = \otimes^1$. Drugim riječima, Emp je upravo skup svih indeksa prazne jednomjesne funkcije. Ako uzmemo neku drugu izračunljivu funkciju, dobit ćemo neku

drugu klasu (svih njenih indeksa), 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 $\mathcal{F} := \{\{e\}^k \mid e \in A\} \subseteq \mathcal{C}omp_k$, i to tako da će praznom skupu S odgovarati prazni skup $\mathcal{F}$, a skupu $S = \mathbb{N}$ odgovarat će $\mathcal{F} = \mathcal{C}omp_k$. Kako je to preslikavanje injekcija, ostalim skupovima brojeva ($\varnothing \subset S \subset \mathbb{N}$) odgovarat će ostali skupovi funkcija ($\varnothing \subset \mathcal{F} \subset \mathcal{C}omp_k$). Vrijeme je da to formaliziramo.

Definicija 6.24: Neka je $k \in \mathbb{N}_+$. Za svaki $\mathcal{F} \subseteq \mathcal{C}omp_k$ definiramo *skup indeksa* kao

$$\ulcorner \mathcal{F} \urcorner := \{e \in \mathbb{N} \mid \{e\}^k \in \mathcal{F}\}. \tag{6.52}$$

Oznaka ne spominje k jer se k može rekonstruirati iz nepraznog $\mathcal{F}$, a $\ulcorner \varnothing \urcorner = \varnothing^1$ za sve k. ◁

Primijetite sličnost s definicijom (4.58) — kao što smo tamo kodirali riječi zapisom u bazi, ovdje kodiramo funkcije njihovim indeksima. Već smo rekli da relacija $\mathit{index} \subseteq \mathbb{N} \times \mathbb{N}_+ \times \mathcal{C}omp$, zadana s $\mathit{index}(e, k, F) \iff \{e\}^k = F$, nema funkcijsko svojstvo po prvoj varijabli: 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 $\ulcorner \cdots \urcorner : \mathcal{P}(\mathcal{C}omp_k) \to \mathcal{P}(\mathbb{N})$ je injekcija.

Štoviše, iz dokaza će slijediti da $\ulcorner \cdots \urcorner$ strogo raste: $\mathcal{F} \subset \mathcal{G}$ povlači $\ulcorner \mathcal{F} \urcorner \subset \ulcorner \mathcal{G} \urcorner$.

Dokaz. Neka su $\mathcal{F}, \mathcal{G} \subseteq \mathcal{C}omp_k$ različiti. Tada postoji funkcija F takva da je (bez smanjenja općenitosti) $F \in \mathcal{G}$, ali $F \notin \mathcal{F}$. Tada $F \in \mathcal{G} \subseteq \mathcal{C}omp_k$ znači da je F RAM-izračunljiva, pa postoji RAM-program P koji je računa. Tada je po propoziciji 3.54(1) $e := \llcorner P \lrcorner$ indeks funkcije F, odnosno $\{e\}^k \in \mathcal{G}$, pa je $e \in \ulcorner \mathcal{G} \urcorner$.

No kad bi bilo i $e \in \ulcorner \mathcal{F} \urcorner$, to bi značilo $\{e\}^k \in \mathcal{F}$, što je nemoguće jer $F \notin \mathcal{F}$. Drugim riječima, $e \in \ulcorner \mathcal{G} \urcorner \setminus \ulcorner \mathcal{F} \urcorner$, odnosno $\ulcorner \mathcal{F} \urcorner \neq \ulcorner \mathcal{G} \urcorner$. □

Na neki način, do na neprebrojivost skupa $\mathcal{P}(\mathcal{C}omp_k)$, čini se da imamo „kodiranje" skupova izračunljivih funkcija. Onda bismo mogli reći, po uzoru na točku 4.3, da $\mathcal{F}$ ima neko svojstvo izračunljivosti ako karakteristična funkcija $\chi_{\ulcorner \mathcal{F} \urcorner}$ ima to svojstvo. Čak možemo reći, neko *svojstvo* $\wp$ izračunljivih k-mjesnih funkcija je *odlučivo* ako je skup $\ulcorner \{F \in \mathcal{C}omp_k \mid \wp(F)\} \urcorner$ rekurzivan. Ali zbog Riceova teorema, takva definicija bi bila sasvim beskorisna: *jedini* rekurzivni skupovi indeksa su $\varnothing$ i $\mathbb{N}$, odnosno jedina odlučiva svojstva izračunljivih funkcija su trivijalna svojstva $\bot$ i $\top$.

Kako je to moguće? Pa $\mathbb{N}$ ima hrpu rekurzivnih podskupova. Uzmimo za primjer jednočlani skup $\{1\}$. On je rekurzivan po lemi 2.47, ali nije skup indeksa. Recimo, za $k = 1$, u odgovarajućem skupu $\mathcal{F}$ bila bi funkcija Z, ali skup svih njenih indeksa je daleko veći od $\{1\}$ — iz primjera 6.18 vidimo da je beskonačan. Čim smo stavili broj 1 unutra, morali smo staviti i čitavu klasu $\ulcorner \{Z\} \urcorner = [1]_{\approx_1}$.

Dakle, skupovi indeksa nisu bilo kakvi podskupovi od $\mathbb{N}$. Oni moraju biti *invarijantni* na neku 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.

Primjer 6.26: Recimo, mogli bismo zamisliti funkciju $\mathsf{sort}^1 : \mathsf{Seq} \to \mathsf{Seq}$ koja sortira konačne nizove zadane kodovima. Na primjer (*unit test*), $\mathsf{sort}(\langle 2,9,0,2\rangle) = \langle 0,2,2,9\rangle$. Nije preteško pokazati da je sort parcijalno rekurzivna, ali pritom moramo odabrati koji algoritam za sortiranje ćemo koristiti. *Selection sort* bi izgledao ovako nekako:

$$\mathsf{tail}(s) :\simeq \mu t(\langle s[0]\rangle * t = s) \qquad \mathsf{min}(s) :\simeq \mu n\big(\exists i < \mathsf{lh}(s)\big)(s[i] = n) \tag{6.53}$$

$$\mathsf{up}(s,x) \simeq \begin{cases} \langle\rangle, & s = \langle\rangle \\ \mathsf{up}\big(\mathsf{tail}(s), x\big), & \mathsf{Seq}'(s) \wedge s[0] \le x \\ \langle s[0]\rangle * \mathsf{up}\big(\mathsf{tail}(s), x\big), & \mathsf{Seq}'(s) \wedge s[0] > x \end{cases} \tag{6.54}$$

$$\mathsf{sort}(s) \simeq \begin{cases} \langle\rangle, & s = \langle\rangle \\ \overline{\mathsf{l}}_1^2\big(\mathsf{min}(s), \big(\#\, i < \mathsf{lh}(s)\big)\big(s[i] = \mathsf{min}(s)\big)\big) * \mathsf{sort}\big(\mathsf{up}\big(s, \mathsf{min}(s)\big)\big), & \mathsf{Seq}'(s) \end{cases} \tag{6.55}$$

(funkcije up i sort definirane su općim rekurzijama), dok bi *quicksort* bio nešto poput

$$\mathsf{dn}(s,x) \simeq \begin{cases} \langle\rangle, & s = \langle\rangle \\ \mathsf{dn}\big(\mathsf{tail}(s), x\big), & \mathsf{Seq}'(s) \wedge s[0] > x \\ \langle s[0]\rangle * \mathsf{dn}\big(\mathsf{tail}(s), x\big), & \mathsf{Seq}'(s) \wedge s[0] \le x \end{cases} \tag{6.56}$$

$$\mathsf{sort}(s) \simeq \begin{cases} \langle\rangle, & s = \langle\rangle \\ \mathsf{sort}\big(\mathsf{dn}\big(\mathsf{tail}(s), s[0]\big)\big) * \langle s[0]\rangle * \mathsf{sort}\big(\mathsf{up}\big(\mathsf{tail}(s), s[0]\big)\big), & \mathsf{Seq}'(s) \end{cases} \tag{6.57}$$

— što će nakon kompiliranja (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 s qs označimo indeks za *quicksort*, tada 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 $\ulcorner\{\mathsf{sort}\}\urcorner$. Ako bismo prije navedeni *unit test* htjeli napisati u obliku jednadžbe po indeksu,

$$\{e\}(\langle 2,9,0,2\rangle) = \langle 0,2,2,9\rangle, \tag{6.58}$$

tada je jasno da i ss i qs, i svi drugi indeksi iz $\ulcorner\{\mathsf{sort}\}\urcorner$, 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: $\wp(\mathsf{F}) :\Longleftrightarrow \mathsf{F}(810\,152\,280) = 272\,386\,847\,250$. Važno je naglasiti da to svojstvo ne ovisi o implementaciji funkcije F, već samo ovisi o funkciji samoj. ◁

Definicija 6.27: Neka je $k \in \mathbb{N}_+$. Za skup $S \subseteq \mathbb{N}$ kažemo da je k-*invarijantan* ako za sve $e, f \in \mathbb{N}$, iz $e \in S$ i $e \approx_k f$ slijedi $f \in S$. ◁

Invarijantnost karakterizira skupove indeksa brojevno, bez pozivanja na skupove funkcija.

Lema 6.28: Neka je $k \in \mathbb{N}_+$ te $S \subseteq \mathbb{N}$.

Tada je S k-invarijantan ako i samo ako je $S = \ulcorner\mathcal{F}\urcorner$ za neki $\mathcal{F} \subseteq \mathcal{C}\mathrm{omp}_k$.

Dokaz. Za smjer ($\Leftarrow$), neka je $f \approx_k e \in S = \ulcorner \mathcal{F} \urcorner$. Tada je po definiciji $\{f\}^k = \{e\}^k \in \mathcal{F}$, pa je i $f \in \ulcorner \mathcal{F} \urcorner = S$. 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 $\ulcorner \mathcal{F} \urcorner = S$.

Za inkluziju ($\supseteq$), po definiciji $\mathcal{F}$ iz $e \in S$ slijedi $\{e\}^k \in \mathcal{F}$, dakle $e \in \ulcorner \mathcal{F} \urcorner$.

Za inkluziju ($\subseteq$), iz $f \in \ulcorner \mathcal{F} \urcorner$ 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. □

Još je jedna stvar tu na prvi pogled čudna: zašto svojstvo (6.58) nije parcijalno rekurzivno? Čini se da je njegova karakteristična funkcija dobivena kompozicijom iz primitivno rekurzivne funkcije $\chi_=$, parcijalno rekurzivne funkcije comp_1 te konstanti $C^1_{\langle 2,9,0,2\rangle}$ i $C^1_{\langle 0,2,2,9\rangle}$. 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.

Teorem 6.29 (Riceov teorem): Neka je $k \in \mathbb{N}_+$, i $S \subseteq \mathbb{N}$ rekurzivan k-invarijantan skup. Tada je $S = \varnothing$ 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 $\varnothing$ ni $\mathbb{N}$. $S \neq \varnothing$ 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

$$F(x) := \begin{cases} n, & x \in S \\ s, & \text{inače} \end{cases}, \tag{6.59}$$

rekurzivna po teoremu 2.46 (rekurzivna verzija) — simbolički, $F = \mathsf{if}\{S : C^1_n, C^1_s\}$.

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 $F(e) = s \in S$, a 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 $S = \varnothing$ ili $S = \mathbb{N}$. □

6.4.1. Sintaksna i semantička svojstva RAM-programa

Riceov teorem smo izrekli u „pozitivnom" obliku — no on se češće koristi „negativno".

Korolar 6.30: Neka je $S \subset \mathbb{N}$ neprazan i k-invarijantan za neki $k \in \mathbb{N}_+$. Tada S nije rekurzivan.

Dokaz. Ovo je samo obrat po kontrapoziciji teorema 6.29. □

Korolar 6.31: Neka je $k \in \mathbb{N}_+$ te $\varnothing \subset \mathcal{F} \subset \mathcal{C}\mathsf{omp}_k$. Tada $\mathcal{F}$ nije odlučiv.

Dokaz. Označimo $S := \ulcorner \mathcal{F} \urcorner$. Prema lemi 6.28, skup S je k-invarijantan. S druge strane, po lemi 6.25 je $S \neq \ulcorner \varnothing \urcorner = \varnothing$ i $S \neq \ulcorner \mathcal{C}\mathsf{omp}_k \urcorner = \mathbb{N}$ (svaki prirodni broj je indeks neke k-mjesne funkcije). Po korolaru 6.30 S nije rekurzivan, što znači da $\mathcal{F}$ nije odlučiv. □

Korolar 6.32: Neka je $k \in \mathbb{N}_+$ te neka je $\wp$ bilo koje netrivijalno svojstvo k-mjesnih izračunljivih funkcija. Tada nijedan algoritam ne može točno odrediti, za proizvoljnu $F \in \mathcal{C}omp_k$, ima li F svojstvo $\wp$.

Dokaz. Budući da je $\wp$ netrivijalno svojstvo, postoji neka k-mjesna funkcija koja ima to svojstvo, a postoji i neka druga k-mjesna funkcija koja ga nema. To znači da za skup $\mathcal{F} := \{F \in \mathcal{C}omp_k \mid \wp(F)\}$ vrijedi $\varnothing \subset \mathcal{F} \subset \mathcal{C}omp_k$. 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. Po Church–Turingovoj tezi, to znači da bi taj algoritam računao $\chi_{\ulcorner\mathcal{F}\urcorner}$, što je u kontradikciji s korolarom 6.31. □

Za zadani RAM-program možemo postaviti mnoga 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 kompiliranjem 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 muke. Recimo, $\mathcal{R}_{15}$ je relevantan za RAM-program P ako i samo ako vrijedi $\big(\exists i < \mathsf{lh}(e)\big)\big(\mathsf{regn}(e[i]) \geq 15\big)$, što je primitivno rekurzivno svojstvo od $e = \ulcorner P \urcorner$.

S druge strane, Riceov teorem pruža dobar uvid u drugi dio tog pravila. Među semantičkim svojstvima izdvajaju se ona koja govore *samo* o računanoj 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, za utvrđivanje čije odlučivosti je potrebna detaljna analiza — ali 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 općenito uputstvo za korištenje Riceova 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 $T := \ulcorner \mathcal{F} \urcorner$.
3. Nađemo neki element od T (najčešće tako da nađemo neku jednostavnu funkciju iz $\mathcal{F}$, napišemo RAM-program koji je računa, i nađemo njegov kōd).
4. Analogno, nađemo neki element od T^c. Skupa s korakom 3 sada imamo $\varnothing \subset T \subset \mathbb{N}$.
5. Dokažemo da je T k-invarijantan, gdje je k mjesnost koju smo fiksirali u koraku 2.
6. Formalno svedemo $T \leq S$, 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.

Primjer 6.33: Za svaki $k \in \mathbb{N}_+$, skup $\mathtt{Halt}_k$ nije rekurzivan. Naime, ako definiramo $T(e) :\Longleftrightarrow \mathtt{Halt}_k(\vec{0}, e)$, skup $T = \ulcorner \{f \in \mathcal{C}\mathrm{omp}_k \mid \vec{0} \in \mathcal{D}_f\} \urcorner$ je k-invarijantan po lemi 6.28 te vrijedi $0 \notin T$ i $1 \in T$. Dakle T nije rekurzivan, pa zbog $T \leq \mathtt{Halt}_k$ niti $\mathtt{Halt}_k$ nije rekurzivan. $\triangleleft$

7. Rekurzivna prebrojivost

Vidjeli smo već nekoliko puta da domena izračunljive funkcije ne mora biti izračunljiva, čak smo i stekli neki uvid u razlog toga: radi izbjegavanja Russellova 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 — 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 relacijski pandan parcijalnim funkcijama. Tražiti „parcijalnu karakterističnu funkciju" kao funkciju iz $\mathbb{N}$ u $\mathtt{bool}$ 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 zadanog skupa nije jednoznačno određena njime.

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 — bitno je da smo na taj način specificirali *poluodlučive* probleme: ako je odgovor na pitanje potvrdan, algoritam će nam ga dati (čim stane), ali ako je odgovor negativan, nećemo to nikada saznati izvršavajući algoritam, upravo jer neće nikada stati. Tu ideju formaliziramo koristeći činjenicu da smo uvijek stajanje algoritma koji računa funkciju reprezentirali kroz domenu te funkcije — pogledajte definicije 1.11 i 4.5 te dijagram (5.3).

Definicija 7.1: Neka je $k \in \mathbb{N}_+$. Za brojevnu relaciju R^k kažemo da je *rekurzivno prebrojiva* ako postoji $F \in \mathcal{C}\mathrm{omp}_k$ takva da je $R = \mathcal{D}_F$. ◁

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 (2.4) i rezultati 2.6–2.8.

Propozicija 7.2: Svaka rekurzivna relacija je rekurzivno prebrojiva.

Ako imamo rekurzivnu karakterističnu funkciju, trebamo „zaboraviti" vrijednosti 0 (ukloniti tu prasliku iz domene) i ostaviti samo vrijednosti 1 — drugim riječima, restringirati χ_R na R.

Dokaz. Neka je $k \in \mathbb{N}$ te R^k rekurzivna relacija. Tada je prema korolaru 3.62 restrikcija $\chi_R|_R$ parcijalno rekurzivna, s domenom $\mathcal{D}_{\chi_R|_R} = \mathcal{D}_{\chi_R} \cap R = \mathbb{N}^k \cap R = R$. □

Lema 7.3: Neka su $k, l \in \mathbb{N}_+$ te R^k i P^l relacije takve da je $R \leq P$.
Ako je P rekurzivno prebrojiva, tada je i R takva.

Dokaz. Po definiciji 7.1 je $P = \mathcal{D}_H$ za neku parcijalno rekurzivnu funkciju H^l, a po definiciji 5.12 postoje rekurzivne funkcije $G_1^k, \ldots, G_l^k$ takve da je $\chi_R = \chi_P \circ (G_1, \ldots, G_l)$. Označimo $F := H \circ (G_1, \ldots, G_l)$ — to je parcijalno rekurzivna funkcija kao kompozicija takvih. No kako su sve G_i rekurzivne, i zato totalne, formula (2.4) daje

$$\vec{x} \in \mathcal{D}_F \Leftrightarrow \vec{x} \in \bigcap_{i=1}^{l} \mathcal{D}_{G_i} \wedge \big(G_1(\vec{x}), \ldots, G_l(\vec{x})\big) \in \mathcal{D}_H = P \Leftrightarrow \vec{x} \in \bigcap_{i=1}^{l} \mathbb{N}^k \wedge \chi_P\big(G_1(\vec{x}), \ldots, G_l(\vec{x})\big) = 1$$
$$\Leftrightarrow \vec{x} \in \mathbb{N}^k \wedge \big(\chi_P \circ (G_1, \ldots, G_l)\big)(\vec{x}) = 1 \Leftrightarrow \chi_R(\vec{x}) = 1 \Leftrightarrow \vec{x} \in R, \quad (7.1)$$

dakle po aksiomu ekstenzionalnosti, $R = \mathcal{D}_F$ je domena parcijalno rekurzivne funkcije, odnosno rekurzivno prebrojiva relacija. □

Napomena 7.4: Obrat propozicije 7.2 ne vrijedi: kanonski kontraprimjer je Kleenejev 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 1.17. Svakoj 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 $\operatorname{card}\big(\bigcup_{k \in \mathbb{N}_+} \mathcal{P}(\mathbb{N}^k)\big) = \aleph_0 \cdot 2^{\aleph_0} = \mathfrak{c} > \aleph_0$.

Poopćimo sada napomenu 3.59 odnosno korolar 3.62 na rekurzivno prebrojive relacije.

Lema 7.5 (Lema o restrikciji): Neka je $k \in \mathbb{N}_+$, G^k parcijalno rekurzivna funkcija te R^k rekurzivno prebrojiva relacija. Tada je restrikcija $G|_R$ također parcijalno rekurzivna.

Dokaz. Po definiciji 7.1 postoji parcijalno rekurzivna funkcija F^k takva da je $\mathcal{D}_F = R$. Tvrdimo da je $G|_R = I_2^2 \circ (F, G)$. Doista, domene su im jednake $R \cap \mathcal{D}_G$ po lemi 2.6, a na toj zajedničkoj domeni vrijednosti su im jednake vrijednostima funkcije G. □

Funkcija I_2^2 u dokazu igra istu ulogu kao operator ',' u jeziku C: izračuna oba operanda i vraća drugi ako oba imaju vrijednost.

Propozicija 7.6: Neka je $k \in \mathbb{N}_+$ te P^k i R^k rekurzivno prebrojive relacije.
Tada je $P \cap R$ također rekurzivno prebrojiva relacija.

Dokaz. Po definiciji 7.1 postoji parcijalno rekurzivna funkcija F takva da je $\mathcal{D}_F = P$. Po lemi 7.5 je $F|_R$ također parcijalno rekurzivna, s domenom $\mathcal{D}_F \cap R = P \cap R$. □

Tehnički, zahtjev da su 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. Isto vrijedi za lemu o restrikciji ($G^k|_{R^l} = \otimes$ za $k \neq l$). Ipak, nastavit ćemo promatrati prazne relacije i funkcije odvojeno po različitim mjesnostima, kao što smo i dosada činili.

Dokazali smo zatvorenost skupa svih rekurzivno prebrojivih relacija (iste mjesnosti) na presjeke (dviju relacija, ali lako bi se indukcijom dokazalo i za proizvoljno konačno mnogo

relacija). Što je sa zatvorenošću na 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.

Naime, u još smo jednom kontekstu susreli poluodlučivost: projekcija izračunljive relacije $\exists_* R$ ne mora biti izračunljiva, ali ako je $\vec{x} \in \exists_* R$, tada to možemo ustanoviti ispitivanjem, za sve $y \in \mathbb{N}$ redom, vrijedi li $R(\vec{x}, y)$. Te dvije formalizacije poluodlučivosti (zapravo *tri* formalizacije, jer izračunljivost projicirane relacije možemo formalizirati kao primitivnu rekurzivnost ili kao rekurzivnost) su ekvivalentne.

Teorem 7.7: Neka je R brojevna 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, jer je svaka primitivno rekurzivna relacija rekurzivna (primijenimo korolar 2.35 na njenu karakterističnu funkciju).

Za (2) $\Rightarrow$ (1), $R = \exists_* P = \mathcal{D}_{\mu P}$ je domena funkcije μP (2.34), koja 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 3.56, F ima indeks; odaberimo jedan njen indeks i označimo ga s e (sjetite se napomene 3.57). 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) $\exists n\, \mathsf{Final}(\langle \vec{x} \rangle, e, n)$. Dakle, dobili smo projekciju, sad samo treba ovo pod kvantifikatorom zapisati u odgovarajućem obliku: relacija zadana s

$$P(\vec{x}, n) :\Longleftrightarrow \mathsf{Final}(\langle \vec{x} \rangle, e, n) \tag{7.2}$$

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 $P \preceq \mathsf{Final}$, uz dodatni uvjet kao u napomeni 7.13). □

Pomoću projekcijske karakterizacije možemo 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 $R_1^k, R_2^k, \ldots, R_l^k$ rekurzivno prebrojive relacije, sve iste mjesnosti. Tada je njihova unija $\bigcup_{i=1}^l R_i$ također rekurzivno prebrojiva relacija.

Dokaz. Po teoremu 7.7, za svaki $i \in [1 \cdot\cdot\, l]$ postoji rekurzivna relacija P_i^{k+1} takva da je $R_i = \exists_* P_i$. Sada je

$$\begin{aligned}\vec{x} \in \bigcup_{i=1}^l R_i = \bigcup_{i=1}^l \exists_* P_i &\Longleftrightarrow (\exists i \in [1 \cdot\cdot\, l])(x \in \exists_* P_i) \Longleftrightarrow (\exists i \in [1 \cdot\cdot\, l])(\exists y \in \mathbb{N}) P_i(\vec{x}, y) \Longleftrightarrow \\ (\exists y \in \mathbb{N})(\exists i \in [1 \cdot\cdot\, l])\big((\vec{x}, y) \in P_i\big) &\Longleftrightarrow (\exists y \in \mathbb{N})\Big((\vec{x}, y) \in \bigcup_{i=1}^l P_i\Big) \Longleftrightarrow \vec{x} \in \exists_*\Big(\bigcup_{i=1}^l P_i\Big),\end{aligned} \tag{7.3}$$

te je $P := \bigcup_{i=1}^l P_i$ rekurzivna po propoziciji 2.44 — a onda je $\bigcup_{i=1}^l R_i = \exists_* P$ rekurzivno prebrojiva po teoremu 7.7. □

Projekcijska karakterizacija 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 implementaciju paralelnog računanja u l dretvi. Recimo, za $l = 2$,

$$\exists n\, \mathsf{Final}(\langle\vec{x}\rangle, \ulcorner P \urcorner, n) \vee \exists n\, \mathsf{Final}(\langle\vec{x}\rangle, \ulcorner Q \urcorner, n) \iff \exists n \big(\mathsf{Final}(\langle\vec{x}\rangle, \ulcorner P \urcorner, n) \vee \mathsf{Final}(\langle\vec{x}\rangle, \ulcorner Q \urcorner, n)\big) \quad (7.4)$$

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 (7.4), 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. Kontrakcija

Još jedan način gledanja na (7.4) je: poredali smo sve testove $\mathsf{Final}(\langle\vec{x}\rangle, \ulcorner P \urcorner, 0)$, $\mathsf{Final}(\langle\vec{x}\rangle, \ulcorner Q \urcorner, 0)$, $\mathsf{Final}(\langle\vec{x}\rangle, \ulcorner P \urcorner, 1)$, $\mathsf{Final}(\langle\vec{x}\rangle, \ulcorner Q \urcorner, 1)$, $\mathsf{Final}(\langle\vec{x}\rangle, \ulcorner P \urcorner, 2)$, ... u niz tako da svaki dođe na red (svi početni komadi su konačni). Drugim riječima, imamo izračunljivu bijekciju između $\mathbb{N} + \mathbb{N} = \mathbb{N} \times \{0, 1\} \cong \mathbb{N} \times \mathtt{bool}$ i $\mathbb{N}$. Što bi se dogodilo da umjesto $\mathbb{N} \times \mathtt{bool}$ uzmemo $\mathbb{N} \times \mathbb{N} \cong \mathbb{N}^2$? Umjesto jedne ograničene i jedne neograničene kvantifikacije, kao što smo imali u (7.3), 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. *funkcija sparivanja*), 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 (poput `std::pair<unsigned,unsigned>` u jeziku C++).

Lema 7.9: Postoje primitivno rekurzivne funkcije pair^2, fst^1 i snd^1, takve da je $\mathsf{fst} \circ \mathsf{pair} = \mathsf{I}_1^2$ i $\mathsf{snd} \circ \mathsf{pair} = \mathsf{I}_2^2$.

Točkovno, za sve $x, y \in \mathbb{N}$ vrijedi $\mathsf{fst}\big(\mathsf{pair}(x, y)\big) = x \wedge \mathsf{snd}\big(\mathsf{pair}(x, y)\big) = y$.

Prva implementacija. $\mathsf{pair} := \mathsf{Code}^2$, $\mathsf{fst}(p) := p[0]$, $\mathsf{snd}(p) := p[1]$. Tada je pair primitivno rekurzivna po propoziciji 3.4, a fst i snd po korolaru 6.2, kao specijalizacije primitivno rekurzivne funkcije part: $\mathsf{fst} = \$(0, \mathsf{part})$, $\mathsf{snd} = \$(1, \mathsf{part})$. Također za sve $x, y \in \mathbb{N}$ vrijedi

$$(\mathsf{fst} \circ \mathsf{pair})(x, y) = \mathsf{fst}\big(\mathsf{pair}(x, y)\big) = \mathsf{part}\big(\mathsf{Code}^2(x, y), 0\big) = \langle x, y\rangle[0] = x = \mathsf{I}_1^2(x, y), \quad (7.5)$$

i analogno $\mathsf{snd} \circ \mathsf{pair} = \mathsf{I}_2^2$, po propoziciji 3.14(2). □

Druga implementacija. $\mathsf{pair} := \mathsf{bin}^2$, $\mathsf{fst} := \mathsf{arg}_1$, $\mathsf{snd} := \mathsf{arg}_2$. Tada je prva funkcija primitivno rekurzivna po propoziciji 4.58, a preostale dvije po propoziciji 4.66, iz koje slijedi i prikaz koordinatnih projekcija kao kompozicijā $\mathsf{fst} \circ \mathsf{pair}$ i $\mathsf{snd} \circ \mathsf{pair}$. □

Treća implementacija. $\mathsf{pair}(i,j) := i + \sum_{t \le i+j} t$. Ovo je standardna enumeracija „po sporednim dijagonalama”, primitivno rekurzivna po napomeni 2.52, lemi 2.53 i primjeru 2.13. Njenih je prvih devet vrijednosti prikazano u tablici

$$\begin{array}{c|ccccc} \mathsf{pair} & 0 & 1 & 2 & 3 & \cdots \\ \hline 0 & 0 & 1 & 3 & 6 & \cdots \\ 1 & 2 & 4 & 7 & & \\ 2 & 5 & 8 & & & \\ \vdots & \vdots & \vdots & & & \end{array} . \tag{7.6}$$

Ova implementacija je navedena samo da se vidi primjer bijektivnog kodiranja — ali ponovimo, za naše potrebe bijektivnost nije nužna. Funkcije fst i snd ovdje možemo implementirati grubom silom, kao u (4.12), (5.37) i (5.38):

$$\mathsf{fst}(p) := (\mu\, i \le p)(\exists\, j \le p)\big(p = \mathsf{pair}(i,j)\big), \tag{7.7}$$

$$\mathsf{snd}(p) := (\mu\, j \le p)(\exists\, i \le p)\big(p = \mathsf{pair}(i,j)\big), \tag{7.8}$$

što je primitivno rekurzivno jer smo i i j ograničili s p, a što smo mogli jer je uvijek $\mathsf{pair}(i,j) \ge \sum_{t \le i+j} t \ge i + j \ge i, j$. □

Ubuduće smatramo da smo fiksirali neku implementaciju, ali nećemo od nje ništa „privatno” koristiti osim sučelja opisanog u lemi 7.9. Za početak možemo dokazati ostatak tvrdnji potrebnih da bismo imali kodiranje.

Propozicija 7.10: Funkcija pair je injekcija, a slika joj je primitivno rekurzivna.

Dokaz. Pretpostavimo da su $a, b, c, d \in \mathbb{N}$ takvi da vrijedi $\mathsf{pair}(a,b) = \mathsf{pair}(c,d)$. Tada je

$$a = \mathsf{l}_1^2(a,b) = \mathsf{fst}\big(\mathsf{pair}(a,b)\big) = \mathsf{fst}\big(\mathsf{pair}(c,d)\big) = \mathsf{l}_1^2(c,d) = c, \tag{7.9}$$

i analogno $b = d$, pa je $(a,b) = (c,d)$, odnosno pair je injekcija.

Za sliku koristimo tehniku iz dokaza korolara 3.16: $p \in \mathcal{I}_{\mathsf{pair}} \iff \big(\exists\,(x,y) \in \mathbb{N}^2\big)\big(p = \mathsf{pair}(x,y)\big)$, a jedini kandidati za x i y su redom $\mathsf{fst}(p)$ i $\mathsf{snd}(p)$. Dakle, vrijedi $p \in \mathcal{I}_{\mathsf{pair}} \iff p = \mathsf{pair}\big(\mathsf{fst}(p), \mathsf{snd}(p)\big)$, što je primitivno rekurzivno jer su pair, fst, snd i $\chi_=$ takve. □

Sad kada imamo specifikaciju sparivanja, pogledajmo detaljnije što smo napravili s relacijom Final na kraju prethodne točke. Dva njena zadnja argumenta (e i n) „spojili” smo 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 možemo ih imati i više — pa čak i sve prirodne brojeve na njihovu mjestu.

Definicija 7.11: Neka je $k \in \mathbb{N}_+$ te P^{k+1} relacija. Za k-mjesnu relaciju $\hat{P}$, zadanu s

$$\hat{P}(\vec{x}, y) :\iff P\big(\vec{x}, \mathsf{fst}(y), \mathsf{snd}(y)\big), \tag{7.10}$$

kažemo da je dobivena *kontrakcijom* zadnja dva argumenta od P. ◁

Prvo, uočimo da su P i $\hat{P}$ „jednako teške” kad ih promatramo kao probleme. Za relaciju svedivosti $\le$ smo u propoziciji 5.13 vidjeli da je refleksivna i tranzitivna. Ipak, ona nije parcijalni uređaj jer nije antisimetrična: relacija i njena kontrakcija čine kontraprimjer.

Lema 7.12: Za svaku relaciju P mjesnosti barem 2, vrijedi $P \leq \hat{P} \leq P$.

Dokaz. Desna „nejednakost" slijedi iz (7.10) i činjenice da su fst i snd (primitivno) rekurzivne. Lijeva će slijediti iz rekurzivnosti od pair, čim dokažemo da za sve $\vec{x}, y, z$ vrijedi

$$P(\vec{x}, y, z) \iff \hat{P}\big(\vec{x}, \mathsf{pair}(y, z)\big). \tag{7.11}$$

A to pak vrijedi jer je po definiciji $\hat{P}$,

$$\hat{P}\big(\vec{x}, \mathsf{pair}(y, z)\big) \iff P\big(\vec{x}, \mathsf{fst}\big(\mathsf{pair}(y, z)\big), \mathsf{snd}\big(\mathsf{pair}(y, z)\big)\big) \iff P(\vec{x}, y, z), \tag{7.12}$$

budući da je po lemi 7.9 $\mathsf{fst}\big(\mathsf{pair}(y, z)\big) = y$ i analogno $\mathsf{snd}\big(\mathsf{pair}(y, z)\big) = z$. □

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. ◁

Sada možemo i formalno dokazati da je više projekcija jednako izračunljivo kao i jedna — jer dvije projekcije su kao jedna projekcija kontrahirane relacije. Intuitivno, primjerice 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 $\vec{x} \in \exists_* \hat{P}$. To znači da postoji $t \in \mathbb{N}$ takav da vrijedi $\hat{P}(\vec{x}, t)$, odnosno po (7.10), $P\big(\vec{x}, \mathsf{fst}(t), \mathsf{snd}(t)\big)$. No to znači da je $\big(\vec{x}, \mathsf{fst}(t)\big) \in \exists_* P$, 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 $(\vec{x}, y, z) \in P$. Po (7.11) slijedi $\hat{P}(\vec{x}, t)$ za $t := \mathsf{pair}(y, z)$, odnosno $\vec{x} \in \exists_* \hat{P}$. □

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 $(\forall\exists\forall\cdots)$ općenito otežava probleme koje promatramo. Primjerice, $\{e\}^2$ je totalna ako i samo ako vrijedi $\forall x_1 \forall x_2 \exists z\, T_2(x_1, x_2, e, z)$ — i može se pokazati da je to bitno teže (u smislu svedivosti) od bilo kakvog problema koji sadrži samo egzistencijalne ili samo univerzalne kvantifikatore, nakon kojih slijedi rekurzivna relacija. Ideja da problemi postaju sve teži kako dodajemo sve više kvantifikatora, uvijek suprotne vrste od one koju smo upravo dodali, formalizirana je kroz pojam *aritmetičke hijerarhije*. Definiciju i neke osnovne teoreme o aritmetičkoj hijerarhiji možete naći u [VukIzr15], a mnogo detaljniju razradu u [Sho93]. ◁

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 $k \geq 2$ te 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 $\hat{P} \leq P$, pa je $\hat{P}$ također rekurzivna po propoziciji 5.14. □

Korolar 7.17: Neka su $k, l \in \mathbb{N}_+$ te R^{k+l} rekurzivno prebrojiva relacija. Tada je relacija Q^k, zadana s $Q(\vec{x}) :\Longleftrightarrow (\exists \vec{y} \in \mathbb{N}^l) R(\vec{x}, \vec{y})$, također rekurzivno prebrojiva.

Dokaz. Definiciju od Q možemo zapisati kao $Q = \exists_* \exists_* \cdots \exists_* R = (\exists_*)^l R$ te tvrdnju možemo dokazati indukcijom po l. Baza ($l = 1$) je 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. □

7.2. Teorem o grafu za parcijalne funkcije

Teorem o grafu za totalne funkcije (teorem 3.44) opravdava skupovnoteorijsko gledanje funkcija kao njihovih grafova i u kontekstu izračunljivosti — totalna funkcija je jednako izračunljiva kao i njen graf. Za parcijalne funkcije f stvar je kompliciranija, jer (3.73) kaže da moramo uzeti u obzir i domenu $\mathcal{D}_f$ — koja ne mora biti izračunljiva, čak ni ako je f izračunljiva.

Sada kad smo napokon vidjeli da izračunljivost parcijalnih funkcija odgovara poluizračunljivosti relacija, 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 u dokazu teorema 7.7(3); jedina razlika između domene i grafa je što kod grafa moramo voditi računa i o funkcijskim vrijednostima. Zato nam više nije dovoljna relacija $\exists_* \mathsf{Final}$ koja kaže „postoji neki broj koraka do završne konfiguracije", 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 $k \in \mathbb{N}_+$ te F^k parcijalno rekurzivna. Po korolaru 3.55 F ima indeks; fiksirajmo jedan od njih i označimo ga s e (napomena 3.57). Tada po teoremu 3.50 imamo:

$$\mathcal{G}_F(\vec{x}, y) \Longleftrightarrow \vec{x} \in \mathcal{D}_{\{e\}^k} \wedge \{e\}^k(\vec{x}) \simeq y \Longleftrightarrow \exists z \big(T_k(\vec{x}, e, z) \wedge U(z) = y \big). \tag{7.13}$$

Doista, ako je $(\vec{x}, y) \in \mathcal{G}_F$, tada je $\mathcal{G}_F \neq \varnothing$, odnosno $F \neq \otimes$, iz čega slijedi $\mathsf{Prog}(e)$ (kontrapozicijom propozicije 3.54(2)), pa postoji (jedinstveni) RAM-program P takav da je $e = \ulcorner P \urcorner$. Po propoziciji 3.54(1), P^k računa $\{e\}^k = F$.

Iz $(\vec{x}, y) \in \mathcal{G}_F$ zaključujemo $\vec{x} \in \mathcal{D}_F$, pa P-izračunavanje s $\vec{x}$ stane po definiciji 1.11. Označimo sa z kod tog izračunavanja. Tada po propoziciji 3.46 vrijedi $T_k(\vec{x}, e, z)$, a $y = U(z)$ 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 1.11 $y = F(\vec{x})$, odnosno $(\vec{x}, y) \in \mathcal{G}_F$.

Sada zaključujemo ovako: za fiksni e, relacija zadana s

$$R(\vec{x}, y, z) :\Longleftrightarrow T_k(\vec{x}, e, z) \wedge U(z) = y \tag{7.14}$$

je primitivno rekurzivna po propoziciji 2.41, kao konjunkcija dvije primitivno rekurzivne relacije. Tada je po (7.13) $\mathcal{G}_F = \exists_* R$, rekurzivno prebrojiva po teoremu 7.7. $\square$

Sljedeći veliki zalogaj je dokazati obrat teorema 7.18. Kod teorema za totalne funkcije koristili smo minimizaciju: po lemi 3.43, 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 za $x = 5$, 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 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 $R(5, 0)$, $R(5, 1)$, $R(5, 2)$, ... dok neko od njih ne stane — i ako to upravo bude $R(5, 0)$, onda imamo sreće: znamo da je $f(5) = 0$. 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 izraza $f(5)$. 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 mnogo vremena) dobili da vrijedi $R(5, 4)$. Imamo li se razloga radovati? Eliminirali smo brojeve 5, 6 i 7 kao moguće vrijednosti za $f(5)$, ali i dalje ne znamo koliko je to. 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 upravo opisanom scenariju je $g(5) = 7$), jest izračunljiva, i vidimo da *dominira* funkciju f: kad god $f(x)$ ima vrijednost, ima je i $g(x)$, i tada je $g(x) \geq f(x)$.

Doduše, nama nije zadana funkcija f, nego relacija R. Možemo li g opisati pomoću R? 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 možemo osigurati tako da tražimo $R\big(x, g(x)\big)$. Reći da to vrijedi za sve x zapravo znači zahtijevati inkluziju $\mathcal{G}_g \subseteq R$, a tada po lemi 3.43 slijedi (projekcija je monotona — dokažite!) $\mathcal{D}_g = \exists_* \mathcal{G}_g \subseteq \exists_* R$, što s prvim svojstvom daje $\mathcal{D}_g = \exists_* R$. Formalizirajmo ta svojstva.

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 $\mathcal{G}_g \subseteq R$. $\triangleleft$

Točkovno, uvjete na selektor možemo zapisati kao

$$\exists y\big(y \simeq g(\vec{x})\big) \iff \exists y R(\vec{x}, y) \iff R\big(\vec{x}, g(\vec{x})\big). \tag{7.15}$$

Selektor je sasvim općenit pojam koji u skupovnoteorijskom smislu odgovara *izbornoj funkciji*: Ako za svaki $\vec{x} \in \mathbb{N}^k$ definiramo „prerez" (*section*) $R_{\vec{x}}(y) :\iff R(\vec{x}, y)$, 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 [VukTS15, 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, a 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 rekurzivni selektor.

Dokaz. Neka je $k' \geq 2$ te $R^{k'}$ 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 $F := \mathsf{fst} \circ \mu \hat{P}$ zadovoljava sve uvjete. Parcijalno je 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, po korolaru 2.7 vrijedi

$$\mathcal{D}_F = \mathcal{D}_{\mathsf{fst} \circ \mu \hat{P}} = \mathcal{D}_{\mu \hat{P}} = \exists_* \hat{P} = \exists_* \exists_* P = \exists_* R, \tag{7.16}$$

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 $y = \mathsf{fst}\big(\mu t \hat{P}(\vec{x}, t)\big)$. 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 $y = \mathsf{fst}(v)$. No ovo prvo po definiciji znači da vrijedi $P\big(\vec{x}, \mathsf{fst}(v), \mathsf{snd}(v)\big)$, što je zbog drugog ekvivalentno s $P\big(\vec{x}, y, \mathsf{snd}(v)\big)$. To pak znači da postoji $z \in \mathbb{N}$ (konkretno, $z := \mathsf{snd}(v)$) takav da je $(\vec{x}, y, z) \in P$, odnosno $(\vec{x}, y) \in \exists_* P = R$. □

Primijetimo sličnost upravo napisanog dokaza s dokazom Kleenejeva teorema o normalnoj formi: ako definiramo primitivno rekurzivnu relaciju M^4 s

$$M(x, e, y, n) :\Longleftrightarrow \mathsf{Final}(x, e, n) \wedge \mathsf{result}(\mathsf{Reg}(x, e, n)) = y, \tag{7.17}$$

specifikacije „$e = \ulcorner P \urcorner$, $x = \langle \vec{x} \rangle$ te P-izračunavanje s $\vec{x}$ stane u najviše n koraka s rezultatom y", tada je $\mathsf{univ} = \mathsf{fst} \circ \mu \hat{M}$ (i $\mathsf{step} = \mathsf{snd} \circ \mu \hat{M}$ za uobičajene implementacije sparivanja), što je istog oblika kao nađeni selektor, a i kao Kleenejev teorem $\mathsf{univ} = U \circ \mu \check{T}$. Još preciznije, vrijedi $\mathcal{G}_{\mathsf{univ}} = \exists_* M$. (Dokažite to!)

Dakle, za svaku rekurzivno prebrojivu relaciju možemo iz svakog nepraznog prereza $R_{\vec{x}}$ izračunljivo odabrati po jedan element y: ne nužno najmanji, već onaj čiju je pripadnost

prerezu „najlakše” utvrditi — recimo, u prvoj implementaciji sparivanja, onaj s najmanjim kodom $\langle y, n\rangle$, gdje je n broj koraka potrebnih da se utvrdi $R(\vec{x}, y)$. No točno koji element selektor bira 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. To znači da svaki prerez ima najviše jedan element — pa „najmanji”, „bilo koji” i „jedini” element prereza znače jedno te isto, nedefinirano samo za prazan prerez.

Lema 7.21: Za svaku brojevnu funkciju F, jedini selektor grafa $\mathcal{G}_F$ je upravo F.

Dokaz. F jest selektor za $\mathcal{G}_F$: uvjet $\mathcal{D}_F = \exists_* \mathcal{G}_F$ imamo iz leme 3.43, a trivijalno je $\mathcal{G}_F \subseteq \mathcal{G}_F$.

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 $\big(\vec{x}, G(\vec{x})\big) \in \mathcal{G}_G \subseteq \mathcal{G}_F$, pa po definiciji grafa vrijedi $F(\vec{x}) = G(\vec{x})$.

Drugim riječima, G i F se podudaraju na zajedničkoj domeni, pa su jednake. □

Teorem 7.22 (Teorem o grafu za parcijalne funkcije): Neka je F brojevna 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 Kleenejeva teorema o normalnoj formi (teorem 7.18).

Za smjer ($\Leftarrow$), neka je $\mathcal{G}_F$ rekurzivno prebrojiv. Prema lemi 7.20, postoji parcijalno rekurzivni selektor G za $\mathcal{G}_F$. No kako je prema lemi 7.21 jedini selektor za $\mathcal{G}_F$ upravo F, zaključujemo $F = G$, dakle F je parcijalno rekurzivna. □

7.2.1. Primjene teorema o grafu

Jedna posljedica teorema o grafu je poopćenje propozicije 2.49 na parcijalne funkcije. Tamo smo vidjeli da možemo uzeti izračunljivu totalnu funkciju i promijeniti joj konačno mnogo vrijednosti, ne kvareći njenu izračunljivost. Ključno je bilo da funkcija pritom ostane 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.

Za pripremu dokažimo nekoliko lema koje govore o skupovnim operacijama koje čuvaju rekurzivnu prebrojivost. Za unije i presjeke smo to već vidjeli (propozicije 7.8 i 7.6), sada se pozabavimo razlikama. Napomenimo samo da općenito skupovna razlika dvije rekurzivno prebrojive relacije nije rekurzivno prebrojiva, jer *komplement* kao posebni 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 te R rekurzivna relacija. Tada je $P \setminus R$ također rekurzivno prebrojiva.

Dokaz. Iz teorije skupova znamo da je $P \setminus R = P \cap R^c$ (uz $\mathbb{N}^k$ kao univerzalni skup). Sada je po propoziciji 2.39, R^c rekurzivna, pa je po propoziciji 7.2 rekurzivno prebrojiva — a onda tvrdnja slijedi po propoziciji 7.6. □

Za *simetričnu* razliku, više nije dovoljno zahtijevati da R bude rekurzivna, pa čak ni primitivno rekurzivna — isti kontraprimjer, zapisan u obliku $(P^k)^c = P \triangle \mathbb{N}^k$, funkcionira. Ali ako je R *konačna*, dokaz prolazi.

Lema 7.24: Neka je $k \in \mathbb{N}_+$, P^k rekurzivno prebrojiva relacija te 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 2.48 i 2.35, 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 — primitivno rekurzivna po korolaru 2.48, rekurzivna po korolaru 2.35, i rekurzivno prebrojiva po propoziciji 7.2. Sada tvrdnja slijedi iz propozicije 7.8 ($l = 2$). □

Propozicija 7.25: Neka je $k \in \mathbb{N}_+$, F^k parcijalno rekurzivna te G^k 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 $k+1$). Ako označimo zadani konačni „*diff*" grafova s $E^{k+1} := \mathcal{G}_F \triangle \mathcal{G}_G$, iz teorije skupova znamo — $\big(\mathcal{P}(\mathbb{N}^{k+1}), \triangle\big)$ 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. □

Napisani dokaz primjer je tehnike kojom možemo dokazati da razne operacije na funkcijama čuvaju parcijalnu rekurzivnost: prebacimo se na grafove, dokažemo da odgovarajuća operacija na grafovima čuva rekurzivnu prebrojivost, i vratimo se na funkcije. Evo još jednog primjera.

Dokaz teorema o grafu, i još prije toga 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 samo nismo dovoljno dugo čekali.

U tom kontekstu, paralelizacija kaže 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 odgovor da/ne (kao u propoziciji 7.8 ili 7.16), ali ako imamo funkcije s kompliciranijim vrijednostima, to može biti problem.

Da bismo sačuvali determinizam, moramo „izvana" osigurati da se ne može dogoditi više stvari, odnosno imati nekoliko u parovima disjunktnih poluodlučivih relacija. Ako 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 2.46 i 3.60) — možemo *čekati* dok se neki uvjet ne ispuni i tada izračunati odgovarajuću granu. Pri tome treba paziti da po definiciji poluodlučivosti **ne možemo imati granu „inače"** — koliko god dugo čekali, ne možemo biti sigurni da se nijedan navedeni 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 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, ..., G_l^k parcijalno rekurzivne funkcije, a R_1^k, R_2^k, ..., R_l^k u parovima disjunktne rekurzivno prebrojive relacije, sve iste mjesnosti.
Tada je i funkcija $F := \mathrm{if}\{R_1 : G_1, R_2 : G_2, \ldots, R_l : G_l\}$ također parcijalno rekurzivna.

Dokaz. Za početak osigurajmo da nijedna grana nije definirana izvan svog uvjeta. Dakle, za svaki $i \in [1 \cdot\cdot\, l]$ definiramo $H_i := G_i|_{R_i}$. Svaka H_i je parcijalno rekurzivna po lemi 7.5, i vrijedi $F = \mathrm{if}\{\mathcal{D}_{H_1} : H_1, \mathcal{D}_{H_2} : H_2, \ldots, \mathcal{D}_{H_l} : H_l\}$. Doista, domene su im iste: $\dot\bigcup_{i=1}^{l} \mathcal{D}_{H_i} = \dot\bigcup_{i=1}^{l} (\mathcal{D}_{G_i} \cap R_i)$; i za svaki $\vec{x}$ iz te domene postoji jedinstveni $j \in [1 \cdot\cdot\, l]$ takav da je $\vec{x} \in \mathcal{D}_{H_j}$, pa je $F(\vec{x}) = G_j(\vec{x}) = H_j(\vec{x})$.

Sljedeći korak je dokazati

$$\mathcal{G}_F = \bigcup_{i=1}^{l} \mathcal{G}_{H_i}. \tag{7.18}$$

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) $j \in [1 \cdot\cdot\, l]$ takav da vrijedi $\vec{x} \in \mathcal{D}_{H_j}$, pa 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 $j \in [1 \cdot\cdot\, l]$ 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 $\mathcal{G}_{G_j}$, iz čega je $y = y'$ jer $\mathcal{G}_{G_j}$ ima funkcijsko svojstvo. Slijedi $F(\vec{x}) = y$, odnosno $(\vec{x}, y) \in \mathcal{G}_F$.

Sada primijenimo opisanu tehniku: 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. □

7.3. Postov teorem

Korolar 3.61 posebni je slučaj teorema 7.26 (a ne samo teorema 3.60), zbog propozicije 7.2. No teorem 3.60 *nije* posebni slučaj teorema 7.26, jer ima „podrazumijevanu" 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 3.60 i 2.46, „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 (Postov teorem): Neka je R brojevna 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. Po propoziciji 2.39 je 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 vrijedi $\chi_R(\vec{x}) \simeq \left\{ \begin{smallmatrix} 1, & \vec{x} \in R \\ 0, & \vec{x} \in R^c \end{smallmatrix} \right.$, dakle ($k$ označava mjesnost od R) $\chi_R^k = \mathrm{if}\{R : C_1^k, R^c : C_0^k\}$ je parcijalno rekurzivna po teoremu 7.26 i propoziciji 2.21 — 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. □

Postov teorem opravdava naziv „poluodlučivost" za intuiciju rekurzivne prebrojivosti: figurativno, ako je problem „napola odlučiv" s jedne strane, i „napola" sa suprotne strane, onda je zapravo sasvim odlučiv.

Pomoću Postova teorema možemo dokazati da razni skupovi nisu rekurzivno prebrojivi. Zapravo, kad god imamo skup koji nije rekurzivan — a imamo ih hrpu po Riceovu 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. Postov teorem često koristimo kad znamo da jedan od ta dva skupa jest 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 Postova teorema. □

Primjer 7.29: Po korolaru 7.28 i napomeni 7.4, K^c nije rekurzivno prebrojiva.

(Možete li naći relaciju takvu da ni ona ni njen komplement nisu rekurzivno prebrojive? Pokušajte dokazati da relacija B^2 zadana s $B(x, y) :\Longleftrightarrow K(x) \leftrightarrow K(y)$, a onda i njena kontrakcija $\hat{B}^1$, imaju 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 [Post44], i zato su nazvani *rekurzivno prebrojivima*: to su oni skupovi koji se mogu rekurzivno prebrojiti (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, tada je $S = \mathcal{I}_a$. Ipak, u međuvremenu se terminologija malo pomaknula, pa s tom intuicijom danas postoje dva problema.

Prvo, rekurzivnost danas podrazumijeva totalnost, a svaka totalna brojevna funkcija ima nepraznu domenu, pa mora imati i nepraznu sliku. Htjeli bismo da prazan skup $\varnothing^1$ također bude rekurzivno prebrojiv, pa ćemo ga morati odvojiti kao posebni slučaj za totalne funkcije.

Drugo, enumeracija često podrazumijeva injektivnost, bez ponavljanja elemenata — no tako možemo dobiti samo *prebrojive* skupove. Konačne skupove bismo također htjeli zvati rekurzivno prebrojivima (jer su rekurzivni — korolar 2.48) pa ćemo morati dopustiti ponavljanja.

Štoviše, kao u teoremu 7.7, enumeracija će moći biti *primitivno* rekurzivna — uz izuzetak praznog skupa, kao što smo već rekli.

Teorem 7.30 (Teorem enumeracije): Neka je R jednomjesna brojevna relacija ($R \subseteq \mathbb{N}$).

Tada su sljedeće tvrdnje ekvivalentne:

(1) R je rekurzivno prebrojiva;

(2) R je slika neke parcijalno rekurzivne funkcije;

(3) R je slika neke primitivno rekurzivne funkcije, ili $R = \varnothing$.

Dokaz. Kao i u teoremu 7.7, da (3) povlači (2) je trivijalno: svaka primitivno rekurzivna funkcija je parcijalno rekurzivna, a $\varnothing^1$ je slika parcijalno rekurzivne funkcije $\otimes^1$ (primjer 2.31).

Da (2) povlači (1) je malo kompliciranije, ali i dalje sasvim jasno koristeći ono što znamo o projekcijama: $y \in \mathcal{I}_F$ znači $(\exists \vec{x} \in \mathcal{D}_F)\big(y \simeq F(\vec{x})\big)$, 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 $P \preceq \mathcal{G}_F$, a po teoremu 7.18 je $\mathcal{G}_F$ rekurzivno prebrojiv, pa je po lemi 7.3 i P rekurzivno prebrojiva. Sada je $\mathcal{I}_F(y) \iff \exists \vec{x}\, P(y, \vec{x})$, što je rekurzivno prebrojivo po korolaru 7.17.

Opet je najzanimljiviji dokaz da (1) povlači (3). Ovdje možemo učiniti nešto bolje od oponašanja dokaza teorema 7.7 — možemo *iskoristiti* taj teorem da se dočepamo primitivne rekurzivnosti: postoji primitivno rekurzivna relacija P^2 takva da je $R^1 = \exists_* P$. To znači da vrijedi $R(x) \iff \exists y (x \mathrel{P} y)$, 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 $x \mathrel{P} y$ — no što ako ne vrijedi? Ako *nikada* ne vrijedi, tada je $R = \varnothing$ pa nemamo što dokazivati. Ako pak $R \neq \varnothing$, tada postoji neki fiksni (recimo, najmanji) element $r \in R$, koji možemo iskoristiti kao *joker* za $x \not\mathrel{P} y$. Sve u svemu, funkcija $G := \mathrm{if}\big\{P : I_1^2, C_r^2\big\}$, točkovno $G(x, y) := \begin{cases} x, & x \mathrel{P} y \\ r, & \text{inače} \end{cases}$, je primitivno rekurzivna po teoremu 2.46, i tvrdimo da je $R = \mathcal{I}_G$.

Za ($\subseteq$), neka vrijedi $x_0 \in R = \exists_* P$. To po definiciji znači da postoji $y_0 \in \mathbb{N}$ takav da vrijedi $x_0 \mathrel{P} y_0$. No tada je $x_0 = G(x_0, y_0) \in \mathcal{I}_G$.

Za ($\supseteq$), neka je $z \in \mathcal{I}_G$ proizvoljan. To znači da postoji $(x_0, y_0) \in \mathbb{N}^2$ takav da je $G(x_0, y_0) = z$. Sada se pitamo vrijedi li $P(x_0, y_0)$. Ako ne, tada je očito $z = G(x_0, y_0) = r \in R$. Ako da, tada je $z = G(x_0, y_0) = x_0$, što je (zbog $x_0 \mathrel{P} y_0$) element od $\exists_* P = R$. □

Napomena 7.31: Enumeracija G koju smo konstruirali je dvomjesna; ako baš želimo *niz*, možemo ga definirati kontrakcijom: $F(n) := G\big(\mathrm{fst}(n), \mathrm{snd}(n)\big)$. Tehnikama kao u dokazu leme 7.12 (samo za funkcije umjesto relacija) možemo dokazati $\mathcal{I}_F = \mathcal{I}_G$ — inkluzija ($\subseteq$) slijedi iz definicije od F, a ($\supseteq$) iz $G(x, y) = F\big(\mathrm{pair}(x, y)\big)$.

Simbolički, $F = G \circ (\mathrm{fst}, \mathrm{snd})$, a $G = F \circ \mathrm{pair}$. Sada su obje inkluzije posljedice toga da za svaku kompoziciju vrijedi $\mathcal{I}_{H \circ (G_1, \ldots, G_l)} \subseteq \mathcal{I}_H$ — raspišite detalje! ◁

7.4. Rekurzivno prebrojivi jezici

Rekurzivno prebrojive brojevne 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 4.3 definirali smo kodiranje jezika $\langle L \rangle$ kao jednomjesnu brojevnu relaciju, i rekli da svojstva izračunljivosti jezika L možemo gledati kroz analogna svojstva te relacije. 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 jezika̅ 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, a umjesto završnog stanja imaju *izlazno* stanje q_p. Enumerator nema ulaza (obje trake su na početku prazne) i radi beskonačno dugo (jer 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.

Završimo naše istraživanje izračunljivosti važnim rezultatom: sva ta četiri pristupa karakteriziraju istu klasu jezika.

Teorem 7.32: Neka je Σ abeceda te $L \subseteq \Sigma^*$ 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 $\mathcal{D}_{\mathsf{F}} = \langle L \rangle$. Jezična funkcija $\varphi := \mathbb{N}^{-1}\mathsf{F}$ je Turing-izračunljiva po korolaru 4.43, a 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 $L = \mathcal{D}_\varphi$ te $\mathcal{T}$ 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 simulacija paralelizacije na Turingovu stroju. Skicu dokaza možete vidjeti u [Sip13, *theorem* 3.21].

$(\mathcal{E}) \Rightarrow (\mathcal{R})$ Treba provesti čitav postupak iz točke 4.1 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 $\delta_2 : Q \times \Gamma^2 \to Q \times \Gamma^2 \times \{-1, 0, 1\}^2$ (jer enumerator ima dvije trake) je malo kompliciranija za kodirati, ali iste ideje kao u dokazu leme 4.20 (proširenje konačnih funkcija nulom, korolar 2.50) prolaze.

Dobijemo pet jednomjesnih (primaju samo broj koraka, jer enumerator nema ulaz) funkcijā 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 2.22 i korolar 2.36 — uvođenjem irelevantnog argumenta) iz primitivno rekurzivnih funkcija, pa su primitivno rekurzivne. Sada definicija nabrajanja 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 $\mathsf{State}(n) = \mathbb{N}Q(q_p) = 1$ i $\mathsf{OutTape}(n) = \langle w_\sqcup \ldots \rangle = \mathsf{Recode}(\langle w \rangle, b', b)$ (s b' i b je označen broj znakova u Σ i Γ redom). Drugim riječima,

$$t \in \langle L \rangle \iff \exists n \big(\mathsf{State}(n) = 1 \wedge \mathsf{OutTape}(n) = \mathsf{Recode}(t, b', b) \big), \tag{7.19}$$

pa je $\langle L \rangle$ rekurzivno prebrojiv po teoremu 7.7. □

Popis definicija, propozicija, teorema, primjera, lema, napomena i korolara

Bibliografija

[BD05] 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.

[Blo06] 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.

[CBH19] Alex Churchill, Stella Biderman i Austin Herrick. *Magic: The Gathering is Turing Complete*. Travanj 2019. URL: https://arxiv.org/pdf/1904.09828.pdf.

[Chen16] Raymond Chen. *Why does the Windows calculator generate tiny errors when calculating the square root of a perfect square?* Lipanj 2016. URL: https://blogs.msdn.microsoft.com/oldnewthing/20160628-00/?p=93765.

[Čač20] Vedran Čačić. *Calculus of Inductive Constructions*. Svibanj 2020. URL: https://meduza.carnet.hr/index.php/media/watch/17301.

[Emd14] 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/.

[EWD82] Edsger Wybe Dijkstra. „Why numbering should start at zero". Kolovoz 1982. URL: http://www.cs.utexas.edu/users/EWD/ewd08xx/EWD831.PDF.

[HW03] James Huggins i Charles Wallace. *An Abstract State Machine Primer*. Teh. izv. Travanj 2003.

[Lov18] Sandro Lovnički. „Interpreter za λ-račun". Diplomski rad. Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet, rujan 2018.

[LV16] Ori Lahav i Viktor Vafeiadis. „Explaining relaxed memory models with program transformations". *International Symposium on Formal Methods*. Springer. 2016., str. 479–495. URL: http://plv.mpi-sws.org/trns/paper.pdf.

[Maz08] Eric Mazur. *The make-believe world of real-world physics*. Srpanj 2008. URL: https://youtu.be/8dU_WNf8Wg0.

[Net13] Netch. *Why is the copying instruction usually named MOV?* 2013. URL: https://softwareengineering.stackexchange.com/questions/222254/why-is-the-copying-instruction-usually-named-mov.

[Pos16] 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.

[Post44] 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.

[Sho93] Joseph Robert Shoenfield. *Recursion theory*. Lecture notes in logic. Springer-Verlag, 1993. ISBN: 9783540570936.

[Sip13] Michael Sipser. *Introduction to the Theory of Computation*. 3. izdanje. Boston, MA: Course Technology, 2013. ISBN: 113318779X.

[Smu92] Raymond Merrill Smullyan. *Godel's Incompleteness Theorems*. Oxford Logic Guides. Oxford University Press, 1992. ISBN: 9780195364378.

[Soa96] Robert Irving Soare. „Computability and Recursion". *Bulletin of Symbolic Logic* 2.3 (1996.), str. 284–321. DOI: 10.2307/420992. URL: http://www.people.cs.uchicago.edu/~soare/History/handbook.pdf.

[SW11] Robert Sedgewick i Kevin Wayne. *Algorithms*. 2011.

[Tao09] Terence Tao. *The "no self-defeating object" argument*. 2009. URL: https://terrytao.wordpress.com/2009/11/05/the-no-self-defeating-object-argument/.

[Tur37] Alan Mathison Turing. „On computable numbers, with an application to the Entscheidungsproblem". *Proceedings of the London mathematical society* 2.1 (1937.), str. 230–265.

[VukIzr15] Mladen Vuković. *Izračunljivost — predavanja*. 2015.

[VukML09] Mladen Vuković. *Matematička logika*. Element, 2009. ISBN: 978-953-197-519-3.

[VukTS15] Mladen Vuković. *Teorija skupova — predavanja*. Siječanj 2015.