Uvod u teorijsko računarstvo

Milan Banković
Katedra za računarstvo i informatiku
Matematički fakultet
Univerzitet u Beogradu

5. jun 2026.

Sadržaj

Glava 1

Uvod

Teorijsko računarstvo (engl. *theoretical computer science* (TCS)) je oblast računarstva koja proučava matematičke osnove izračunavanja. Kao takvo, teorijsko računarstvo je takođe i oblast matematike, tj. nalazi se u *preseku* matematike i računarstva.

Teorijsko računarstvo nam daje odgovore na sledeća pitanja:

- na koji način se izračunavanje može formalno matematički opisati (tj. kakvi sve *modeli izračunavanja* postoje)?
- koje su granice formalnog izračunavanja (tj. šta je moguće *efektivno* izračunati?
- koja je cena takvog izračunavanja (tj. koja je *složenost*)?
- na koji način možemo formalno matematički *rezonovati* o procesima izračunavanja (tj. proučavati svojstva izračunavanja i dokazivati korektnost izračunavanja)?

1.1 Oblasti teorijskog računarstva

Savremeno teorijsko računarstvo uključuje veliki broj oblasti. Pritom, granice između različitih oblasti nisu uvek sasvim jasne, jer postoji dosta preklapanja. Takođe, mnoge oblasti računarstva koje imaju jaku matematičku podlogu se ponekad svrstavaju u teorijsko računarstvo, iako to istorijski gledano nije slučaj. Najzad, neke oblasti koje se obično svrstavaju u teorijsko računarstvo u sebi imaju sadržaje koji imaju mnoge praktične aspekte, pa se zato te oblasti ponekad svrstavaju i u druge discipline računarsva koje nisu deo teorijskog računarstva. Otuda spisak koji sledi nije konačan ni definitivan, ali jeste sastavljan iz oblasti koje su najznačajnije i za koje ne postoji nikakva dilema da spadaju u teorijsko računarstvo u najvećoj mogućoj meri.

1.1.1 Teorija izračunljivosti

Teorija izračunljivosti (engl. *computability theory*) se bavi matematičkim modelima izračunavanja i daje nam odgovor na pitanje šta je efektivno izračunljivo. Istorijski, ovo je prva grana teorijskog računarstva koja je nastala,

pre svega za potrebe formalizacije matematike i automatizacije formalnog rezonovanja. U okviru ove teorije, razvijeni su različiti formalni modeli izračunavanja (tj. opisivanja algoritama) za koje je pokazano da su međusobno ekvivalentni, ali su nam dali različite poglede na proces računanja i omogućili bolje razumevanje tog procesa. Ovo je omogućilo dalji praktičan razvoj računarstva, kroz, na primer, dizajn različitih vrsta programskih jezika. Sa druge strane, teorija izračunljivosti je poznata po brojnim *negativnim* rezultatima, koji se tiču granica formalne izračunljivosti. Zanimljivo je da su te granice uspostavljene pre nego što je napravljen prvi elektronski računar (tokom 30tih godina 20. veka).

1.1.2 Teorija složenosti izračunavanja

Teorija složenosti izračunavanja (engl. *computational complexity theory*) proučava cenu izračunavanja u smislu resursa koji su za neko izračunavanje potrebni. U te resurse pre svega ubrajamo potrebno vreme (izraženo u broju računskih koraka), ali i memorijski prostor. Teorija složenosti razmatra osnovne *klase složenosti*, tj. razvrstava probleme prema tome da li su npr. rešivi u *polinomskom* ili *eksponencijalnom* vremenu ili prostoru. Mnogi odnosi među ovim klasama su još uvek otvoreni, tako da je ova oblast poznata po mnogim još uvek nerazrešenim pitanjima (jedno od najčuvenijih je „da li je P = NP?").

1.1.3 Algoritmi i strukture podataka

Oblast *algoritmi i strukture podataka* (engl. *algorithms and data structures*) se bavi dizajnom efikasnih postupaka izračunavanja. To podrazumeva metode dizajna i konstrukcije efikasnih algoritama, kao i struktura podataka nad kojima ti algoritmi rade. Takođe, ova oblast proučava i metode za praktičnu analizu algoritama, u smislu utvrđivanja njihove složenosti. Za razliku od teorije složenosti izračunavanja koja se bavi „grubom" analizom (npr. da li se neki problem može rešiti u polinomskom vremenu ili ne), analiza algoritama podrazumeva preciznu analizu složenosti konkretnog algoritma (npr. da li je algoritam složenosti $O(n^2)$ ili $O(n \log n)$?). Za takvu analizu se proučavaju razne matematičke metode poput *rekurentnih relacija* i *funkcija generatrisa*.

1.1.4 Teorija formalnih jezika i automata

Teorija formalnih jezika i automata (engl. *formal languages and automata theory*) je vrlo bliska teoriji izračunljivosti, a proučava klase formalnih jezika, poput *regularnih*, *kontekstno slobodnih* i *kontekstno zavisnih jezika*. U praksi, u ove formalne jezike spadaju razni jezici koji se koriste u računarstvu, poput programskih jezika, jezika za obeležavanje, konfiguracionih jezika, jezika za računarsko i matematičko modelovanje, ali i jezika koji se koriste u, na primer, matematičkoj logici. Takođe, ova oblast razmatra formalne modele algoritama za *prepoznavanje* različitih klasa jezika (tj. koji omogućavaju da se utvrdi da li proizvoljna niska pripada jeziku ili ne). Ovakvi modeli se nazivaju *automati* (npr. *konačni automati*, *potisni automati*, *linearno-ograničeni automati*). Automati su jednostavniji modeli izračunavanja u odnosu na opšte modele koji se izučavaju u okviru teorije izračunljivosti. U praksi, automati se koriste prilikom dizajna i konstrukcije *programskih prevodilaca*.

1.1.5 Semantika programskih jezika

Oblast *semantika programskih jezika* (engl. *programming languages semantics*) je oblast koja je u preseku teorijskog računarstva i *teorije programskih jezika* (koja, pored teorijskih, ima i svoje praktične aspekte). Semantika programskih jezika omogućava da formalno logički opisujemo *značenje* naših programa. Ovo je u praksi veoma značajno za konstrukciju programskih prevodilaca. Takođe, formalni opis semantike nam omogućava i da rezonujemo o našim programima i dokazujemo njihovu ispravnost.

1.1.6 Teorija tipova

Teorija tipova (engl. *type theory*) je još jedna oblast koja ima svoje teorijske i praktične aspekte, pa se kao takva nalazi u preseku teorijskog računarstva i drugih oblasti, poput pomenute teorije programskih jezika. U teorijskom smislu, teorija tipova uspostavlja korespondenciju između matematičke logike i računarskih programa („teorema = tip", „dokaz = program"). U praktičnom smislu, teorija tipova je značajna zato što izučava različite modele tipizacije podataka u programskim jezicima, što ima svoju praktičnu primenu prilikom dizajna novih i unapređenja postojećih programskih jezika.

1.1.7 Kvantno računarstvo

Kvantno računarstvo (engl. *quantum computing*) je jedna od novijih oblasti teorijskog računarstva koja proučava modele izračunavanja zasnovane na kvantnoj mehanici. Kao i u klasičnom računarstvu, matematički modeli predstavljaju prethodnicu realnim mašinama koje ta izračunavanja mogu da obavljaju. Tako su različiti algoritmi kvantnog računarstva poznati već decenijama, a tek u novije vreme nastaju realni kvantni računari koji neke od tih algoritama mogu i izvršavati.

1.1.8 Matematička logika u računarstvu

Matematička logika u računarstvu (engl. *logic in computer science*) je široka oblast koja pokriva različite veze između matematičke logike i računarstva. Kao takva, ona je deo teorijskog računarstva, jer izučava matematičke osnove izračunavanja i veze matematike i računarstva. Ipak, matematička logika u računarstvu nije izolovana oblast teorijskog računarstva, već predstavlja „lepak" koji povezuje više oblasti teorijskog računarstva u jednu celinu. Na primer, teorija izračunljivosti je iznikla iz matematičke logike i predstavlja neraskidivu vezu između matematike i računarstva. Teorija složenosti se takođe prožima sa matematičkom logikom (na primer, prvi problem za koji je dokazano da je *NP-kompletan* je problem zadovoljivosti formule iskazne logike, *problem SAT*). Semantika programskih jezika je zasnovana na metodama matematičke logike za formalizovanje semantike programskih jezika i rezonovanje o njima. Teorija tipova direktno uspostavlja vezu između matematičke logike i računarskih programa.

Sa druge strane, računarske metode se široko koriste u automatizaciji procesa logičkog rezonovanja. Oblasti poput *automatskog rezonovanja* i *interaktivnog dokazivanja teorema* koriste algoritme za automatsko generisanje i proveru

ispravnosti formalnih matematičkih dokaza, kao i za automatsko rešavanje problema zadatih na jeziku matematičke logike. Može se reći da na ovaj način računarstvo vraća svoj dug matematičkoj logici, za njen nemerljiv doprinos razvoju računarstva.

1.1.9 Ostale oblasti teorijskog računarstva

Često se u literaturi kao deo teorijskog računarstva spominju i mnoge druge oblasti, poput *kriptografije*, *paralelno i distribuirano izračunavanje*, *teorija informacija*, *simboličko izračunavanje*, *računarska geometrija* i sl. Mnoge od ovih oblasti zaista imaju značajan presek sa teorijskim računarstvom. Ipak, ne može se reći da apsolutno pripadaju teorijskom računarstvu, s obzirom da u značajnoj meri pripadaju i drugim oblastima matematike i računarstva.

1.2 Teorijsko računarstvo u okviru računarstva

Računarstvo (engl. *computer science* (CS)) danas predstavlja samostalnu naučnu disciplinu. Ona je istorijski iznikla iz teorijskog računarstva, pa samim tim i iz matematike. Međutim, danas postoji živa akademska rasprava da li je savremeno računarstvo više matematika ili je više inžinjerstvo. Ono što je sigurno je da računarstvo danas predstavlja hibridnu disciplinu koja uključuje svoje matematičke, inžinjerske, pa čak i društvene aspekte.

U klasičnoj definiciji, računarstvo je nauka koja se bavi teorijskim i praktičnim aspektima izračunavanja, obrade podataka i procesiranja informacija. Teorijsko računarstvo dominantno pokriva ove ,,teorijske" aspekte. Pored teorijskog računarstva, računarstvo uključuje mnoge praktične discipline, poput arhitekture i organizacije računara, operativnih sistema i računarskih mreža, dizajna programskih jezika i programskih prevodilaca, softverskog inžinjerstva, baza podataka, informacionih sistema, veštačke inteligencije i sl. Iako ove oblasti ne spadaju u teorijsko računarstvo, većina njih imaju dodirnih tačaka sa teorijskim računarstvom ili vuku svoje korene iz nekih od oblasti teorijskog računarstva.

1.3 Značaj teorijskog računarstva

Teorijsko računarstvo je od fundamentalnog značaja za razvoj računarstva i formalnog matematičkog rezonovanja. Ovaj značaj se ogleda u sledećem:

- iz teorijskog računarstva je izniklo čitavo moderno računarstvo. Otuda glavni rezultati teorijskog računarstva prožimaju i sve druge oblasti računarstva i utiču na njihov razvoj. Mnoge ideje nastale u teorijskom računarstvu se kasnije preuzimaju u drugim oblastima računarstva, gde dobijaju svoju praktičnu realizaciju (npr. *računari sa uskladištenim programom* su praktična realizacije *univerzalnih programa*).

- teorijsko računarstvo uspostavlja granice formalne ali i praktične izračunljivosti. Dakle, ono nam govori šta je moguće rešiti algoritamski, šta je teorijski moguće, ali je u praksi veoma teško sa postojećim resursima, a šta nikada neće moći da bude rešeno na taj algoritamski način.

- teorijsko računarstvo proučava modele koji omogućavaju razvoj metoda za formalnu verifikaciju softvera i hardvera. Metode koje se koriste u statičkoj analizi softvera poput *provere modela*, *apstraktne interpretacije* i *simboličkog izvršavanja* se direktno oslanjaju na matematičku logiku, teoriju automata i semantiku programskih jezika.

- teorijsko računarstvo omogućava i usmerava razvoj drugih postojećih, kao i novih oblasti računarstva, kroz razvoj programskih jezika, novih softverskih tehnologija, novih metoda i tehnika koje se primenjuju raznim oblastima računarstva, i sl.

1.4 Struktura teksta

U nastavku ovog teksta biće izložene osnove nekih od oblasti teorijskog računarstva. Akcenat će biti na teoriji izračunljivosti, teoriji složenosti, kao i na vezama matematičke logike i teorijskog računarstva. Sa druge strane, neke od oblasti će namerno biti izostavljene, s obzirom da se u većem obimu proučavaju u okviru drugih obaveznih predmeta na Matematičkom fakultetu. U te oblasti spadaju teorija jezika i automata, kao i algoritmi i strukture podataka. Najzad, neke oblasti će prosto biti izostavljene zbog ograničenog obima teksta.

Glava 2

Matematička logika

Matematička logika predstavlja glavnu sponu između matematike i teorijskog računarstva. Zbog toga u ovoj glavi dajemo pregled osnovnih logičkih pojmova koji će nam biti potrebni u nastavku teksta. Iako će mnogi od ovih pojmova čitaocu biti poznati, ovde ih iznosimo zarad potpunosti izlaganja. Fokus će biti na iskaznoj logici i logici prvog reda.

2.1 Iskazna logika

Iskazna logika predstavlja najjednostavniji logički okvir u kome se može vršiti formalno rasuđivanje. Iskazne formule se grade nad iskazima, koji predstavljaju elementarne činjenice koje mogu biti tačne ili netačne. Iskazna logika je značajna jer predstavlja osnovu za druge, bogatije logike. Takođe, mnogi praktični problemi se mogu izraziti i rešavati u okviru iskazne logike, te je ona stoga korisna i sama po sebi.

2.1.1 Sintaksa iskazne logike

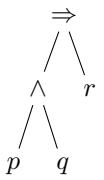
Definicija 2.1. Neka je dat prebrojiv skup *iskaznih slova* (*atoma*) P. *Iskazna formula* se dobija konačnom primenom sledećih pravila:

- $\top$ i $\bot$ su iskazne formule (*logičke konstante*)
- $p \in P$ je iskazna formula (*atom*)
- ako je F iskazna formula, tada je i $\neg F$ iskazna formula
- ako su F i G iskazne formule, onda su i $F \wedge G$, $F \vee G$, $F \Rightarrow G$ i $F \Leftrightarrow G$ iskazne formule

Skup svih iskazni formula nad P označavaćemo sa $\mathcal{F}_P$.

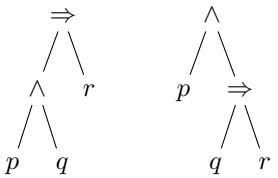
Napomena 2.1. Treba napraviti razliku između *konkretne sintakse* i *apstraktne sintakse*. U apstraktnoj sintaksi, formula se posmatra kao hijerarhijska struktura, izgrađena primenom logičkih veznika na jednostavnije formule. Formulu predstavljamo *stablom apstraktne sintakse* (engl. *abstract syntax tree* (AST)) u čijim se unutrašnjim čvorovima nalaze iskazni veznici ($\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$), a u listovima se nalaze logičke konstante ili atomi. Iz strukture ovog stabla, jasno je koji

se iskazni veznik primenjuje na koje potformule, tj. u potpunosti je definisana struktura formule. Na primer, ako imamo sledeće stablo:

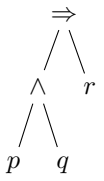


jasno je da je u pitanju *implikacija* ($\Rightarrow$) primenjena na dve potformule, od kojih je prva *konjunkcija* ($\wedge$) dve atomičke formule p i q, a druga je atomička formula r. Apstraktna sintaksna stabla su zgodna kao strukture podataka za predstavljanje formula u računarskim programima.

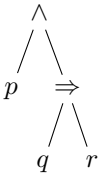
Sa druge strane, apstraktna sintaksa nije pogodna za zapisivanje formula u okviru teksta. U tu svrhu koristimo konkretnu sintaksu. U konkretnoj sintaksi, formula se predstavlja niskom simbola, poput $p \wedge q \Rightarrow r$. Problem sa konkretnom sintaksom je što se ona može tumačiti *višeznačno*: ista konkretna sintaksa može odgovarati različitim apstraktnim sintaksama, tj. može predstavljati različite formule. Na primer, formula $p \wedge q \Rightarrow r$ u konkretnoj sintaksi može odgovarati dvema različitim formulama u apstraktnoj sintaksi:



Kako bi konkretna sintaksa jednoznačno određivala apstraktnu sintaksu formule, potrebno je definisati *prioritete* i *asocijativnosti* operatora. Prema uobičajenoj konvenciji, redosled prioriteta, počev od najvišeg, je sledeći: $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, pri čemu su svi binarni veznici desno asocijativni. Prioritet i asocijativnost se mogu promeniti *zagradama*. Imajući to u vidu, formula $p \wedge q \Rightarrow r$ je sintaksno identična formuli $(p \wedge q) \Rightarrow r$, tj. odgovara apstraktnoj sintaksi:



Sa druge strane, konkretna sintaksa $p \wedge (q \Rightarrow r)$ bi odgovarala apstraktnoj sintaksi:



Primer 2.1. U nastavku navodimo primere iskaznih formula u konkretnoj sintaksi, uz komentare vezane za prioritete i asocijativnosti iskaznih veznika:

- $p \vee q \wedge r$ je u sintaksnom smislu isto što i $p \vee (q \wedge r)$, s obzirom da $\wedge$ ima viši prioritet od $\vee$
- $(p \vee q) \wedge r$: ovde smo zagradama promenili prioritet, tj. sada se najpre primenjuje $\vee$, pa zatim $\wedge$
- $p \wedge q \wedge r$: ova formula predstavlja isto što i $p \wedge (q \wedge r)$, s obzirom da $\wedge$ ima desnu asocijativnost
- $(p \wedge q) \wedge r$: ovim smo promenili asocijativnost operatora, tako da se najpre primenjuje levi $\wedge$ veznik
- $p \Leftrightarrow q \Rightarrow r$ je isto što i $p \Leftrightarrow (q \Rightarrow r)$, zbog prioriteta operatora
- $p \Rightarrow q \Rightarrow r$ je isto što i $p \Rightarrow (q \Rightarrow r)$, zbog desne asocijativnosti operatora $\Rightarrow$.

Napomena 2.2. Konkretna sintaksa iskaznih formula predstavlja *jezik* nad azbukom $P \cup \{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, \top, \bot, (,)\}$, tj. skup niski nad ovom azbukom koje su formirane u skladu sa sintaksnim pravilima datim definicijom 2.1, uz dodatak da ako niska F predstavlja formulu, tada i (F) takođe predstavlja formulu.

Definicija 2.2. *Složenost formule* $c(F)$ se definiše na sledeći način:

- $c(\top) = c(\bot) = c(p) = 0$ (gde je $p \in P$)
- $c(\neg F) = c(F) + 1$
- $c(F \wedge G) = c(F \vee G) = c(F \Rightarrow G) = c(F \Leftrightarrow G) = c(F) + c(G) + 1$

Napomena 2.3. Iz gornje definicije sledi da je složenost formule jednaka broju iskaznih veznika koje formula sadrži.

Primer 2.2. Složenost formule $(p \vee q) \wedge \neg r$ je 3, dok je složenost formule $\neg(\top \Rightarrow q)$ jednaka 2.

Definicija 2.3. *Zamena* formule G formulom H u formuli F (u oznaci $F[G \mapsto H]$ se definiše na sledeći način:

- $F[G \mapsto H] = H$, ako je $F = G$
- $\top[G \mapsto H] = \top$, $\bot[G \mapsto H] = \bot$
- $p[G \mapsto H] = p$, ako je $p \in P$ i $G \neq p$
- $(\neg F)[G \mapsto H] = \neg F[G \mapsto H]$
- $(F_1 \wedge F_2)[G \mapsto H] = F_1[G \mapsto H] \wedge F_2[G \mapsto H]$
- $(F_1 \vee F_2)[G \mapsto H] = F_1[G \mapsto H] \vee F_2[G \mapsto H]$
- $(F_1 \Rightarrow F_2)[G \mapsto H] = F_1[G \mapsto H] \Rightarrow F_2[G \mapsto H]$
- $(F_1 \Leftrightarrow F_2)[G \mapsto H] = F_1[G \mapsto H] \Leftrightarrow F_2[G \mapsto H]$

Napomena 2.4. Zamena $F[G \mapsto H]$ se, prema prethodnoj definiciji, vrši tako što se sva pojavljivanja potformule G u formuli F zamene formulom H. Zamena je čisto sintaksna operacija – njome se vrši transformacija jedne formule u drugu.

Primer 2.3. Zamenom formule $p \wedge q$ formulom $q \wedge p$ u formuli $p \wedge q \Rightarrow r$ dobijamo formulu $q \wedge p \Rightarrow r$, dok zamenom formule $p \wedge q$ formulom q u formuli $(p \wedge q) \vee (r \Rightarrow (p \wedge q))$ dobijamo $q \vee (r \Rightarrow q)$.

2.1.2 Semantika iskazne logike

Definicija 2.4. *Iskazna valuacija* je funkcija $v : P \to \{0, 1\}$, tj. funkcija koja svakom iskaznom slovu pridružuje vrednost 0 (netačno) ili 1 (tačno).

Napomena 2.5. S obzirom da je P prebrojiv skup, valuacije možemo razumeti i kao beskonačne nizove nula i jedinica. Otuda je broj mogućih valuacija nad P neprebrojivo beskonačan.

Definicija 2.5. *Interpretacija* I_v određena valuacijom v je funkcija $I_v : \mathcal{F}_P \to \{0, 1\}$ koja svakoj formuli pridružuje vrednost 0 (netačno) ili 1 (tačno) na sledeći način:

- $I_v(\top) = 1$
- $I_v(\bot) = 0$
- $I_v(p) = v(p)$ (za $p \in P$)
- $I_v(\neg F) = 1$ akko $I_v(F) = 0$
- $I_v(F \wedge G) = 1$ akko je $I_v(F) = 1$ i $I_v(G) = 1$
- $I_v(F \vee G) = 1$ akko je $I_v(F) = 1$ ili $I_v(G) = 1$
- $I_v(F \Rightarrow G) = 1$ akko je $I_v(F) = 0$ ili $I_v(G) = 1$
- $I_v(F \Leftrightarrow G) = 1$ akko je $I_v(F) = I_v(G)$

Primer 2.4. Neka je data valuacija v takva da je $v(p) = 1$, $v(q) = 0$ i $v(r) = 1$. Tada je formula $p \wedge q \Rightarrow r$ logički tačna u valuaciji v (tj. $I_v(p \wedge q \Rightarrow r) = 1$), dok je formula $p \Rightarrow q \wedge r$ logički netačna u v (tj. $I_v(p \Rightarrow q \wedge r) = 0$).

Definicija 2.6. Ako je $I_v(F) = 1$, tada kažemo da je valuacija v *model* formule F. Formula je *zadovoljiva* ako ima bar jedan model. U suprotnom, formula je *nezadovoljiva*.

Napomena 2.6. Zadovoljivost iskazne formule se teorijski može ispitati veoma jednostavno. Naime, iako je skup iskaznih slova P beskonačan (pa samim tim je takav i skup valuacija nad P), u svakoj formuli se pojavljuje samo konačno mnogo iskaznih slova iz P. Ako se u formuli F pojavljuje n različitih iskaznih slova, tada mi imamo 2^n suštinski različitih valuacija koje treba razmotriti (sve valuacije koje se poklapaju na tih n promenljivih su suštinski iste iz ugla formule F). Kako je broj valuacija koje treba ispitati konačan, možemo odrediti vrednost formule u svakoj od njih i videti da li bar jedna od njih zadovoljava formulu. Ovaj postupak je poznat i kao *metod istinitosnih tablica*.

Primer 2.5. Neka je data formula $p \wedge q \Rightarrow r$. Njoj odgovara sledeća istinitosna tablica:

p	q	r	$p \wedge q$	$p \wedge q \Rightarrow r$
0	0	0	0	1
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	1	0
1	1	1	1	1

Iz ove tablice je jasno da je formula zadovoljiva.

Napomena 2.7. Prethodni postupak, iako jednostavan, nije naročito efikasan, s obzirom da broj valuacija koje treba ispitati raste eksponencijalno sa brojem promenljivih n. Zbog toga se često razmatraju ,,pametniji" postupci utvrđivanja zadovoljivosti iskaznih formula koji u praksi rade efikasnije.

Definicija 2.7. Formula F je *tautologija* (*valjana*, *validna*) ako je tačna u svakoj valuaciji. Formula je F *poreciva* ako nije tautologija.

Teorema 2.1. *Formula F je tautologija, akko je $\neg F$ nezadovoljiva. Formula F je poreciva akko je $\neg F$ zadovoljiva.*

Dokaz. Očigledno sledi iz definicije ovih pojmova. □

Primer 2.6. Formula iz primera 2.5 nije tautologija, s obzirom da postoji valuacija u kojoj nije tačna (tj. formula je poreciva).

Primer 2.7. Formula $(p \Rightarrow q) \Rightarrow (\neg q \Rightarrow \neg p)$ je tautologija. Zaista, njena tablica istinitosti izgleda ovako:

p	q	$\neg p$	$\neg q$	$p \Rightarrow q$	$\neg q \Rightarrow \neg p$	$(p \Rightarrow q) \Rightarrow (\neg q \Rightarrow \neg p)$
0	0	1	1	1	1	1
0	1	1	0	1	1	1
1	0	0	1	0	0	1
1	1	0	0	1	1	1

Dakle, vidimo da je tačna u svim valuacijama različitim po p i q.

Definicija 2.8. Skup formula Δ je *zadovoljiv* ako postoji valuacija v takva da istovremeno zadovoljava sve formule skupa Δ.

Primer 2.8. Skup $\Delta = \{p \wedge \neg q, q \Rightarrow p\}$ je zadovoljiv, jer valuacija u kojoj je $v(p) = 1$ i $v(q) = 0$ zadovoljava obe formule skupa Δ. Sa druge strane, skup $\Gamma = \{p \wedge q, q \Rightarrow \neg p\}$ je nezadovoljiv, jer da bi prva formula bila zadovoljena, mora da važi $v(p) = v(q) = 1$, što drugu formulu čini netačnom. Dakle, ne postoji valuacija u kojoj su obe formule tačne.

Definicija 2.9. Formala F je logička posledica skupa formula Δ (u oznaci $\Delta \vDash F$) ako svaka valuacija v koja zadovoljava skup formula Δ ujedno zadovoljava i formulu F. Specijalno, F je logička posledica formule G (u oznaci $G \vDash F$) ako svaka valuacija koja zadovoljava G zadovoljava i F.

Primer 2.9. Formula $p \vee q$ je logička posledica formule $p \wedge q$. Zaista, valuacija v zadovoljava $p \wedge q$ akko je $v(p) = v(q) = 1$. U svim takvim valuacijama tačna je i formula $p \vee q$. Otuda važi $p \wedge q \vDash p \vee q$.

Stav 2.1. *Važi $\Delta \vDash A$ akko je skup $\Delta \cup \{\neg A\}$ nezadovoljiv.*

Dokaz. Neka je v proizvoljna valuacija koja zadovoljava skup Δ. Kako je $\Delta \vDash A$, to znači da je i A tačna u valuaciji v, pa je $\neg A$ netačna u v. Otuda skup $\Delta \cup \{\neg A\}$ nije zadovoljen ni jednom valuacijom. Obratno, ako je skup $\Delta \cup \{\neg A\}$ nezadovoljiv i ako je v proizvoljna valuacija koja zadovoljava skup Δ, tada ta valuacija ne može zadovoljavati $\neg A$. Otuda ona zadovoljava A, pa je $\Delta \vDash A$. □

Definicija 2.10. Formule F i G su *logički ekvivalentne* (u oznaci $F \equiv G$) ako je $F \vDash G$ i $G \vDash F$.

Napomena 2.8. Prema prethodnoj definiciji, formule F i G su logički ekvivalentne ako je za svaku valuaciju v ispunjeno $I_v(F) = I_v(G)$, tj. ako se formule isto interpretiraju u svakoj valuaciji.

Teorema 2.2. *Relacija $\equiv$ je relacija ekvivalencije (refleksivna, simetrična i tranzitivna) na skupu svih formula $\mathcal{F}_P$.*

Dokaz. Očigledno. □

Napomena 2.9. Formule koje pripadaju istoj klasi ekvivalencije po relaciji $\equiv$ su *semantički* jednake, tj. imaju isto značenje, iako se sintaksno mogu razlikovati.

Primer 2.10. U nastavku navodimo neke dobro poznate logičke ekvivalencije koje se nazivaju i *logički zakoni*:

- $\neg\neg F \equiv F$ (zakon dvojne negacije)
- $F \wedge G \equiv G \wedge F$, $F \vee G \equiv G \vee F$ (zakoni komutacije)
- $(F \wedge G) \wedge H \equiv F \wedge (G \wedge H)$, $(F \vee G) \vee H \equiv F \vee (G \vee H)$ (zakoni asocijacije)
- $F \wedge F \equiv F$, $F \vee F \equiv F$ (zakoni idempotencije)
- $F \wedge \neg F \equiv \bot$, $F \vee \neg F \equiv \top$ (zakon kontradikcije i zakon tautolgoje)
- $F \wedge (G \vee H) \equiv (F \wedge G) \vee (F \wedge H)$, $F \vee (G \wedge H) \equiv (F \vee G) \wedge (F \vee H)$ (zakoni distribucije)
- $F \wedge (F \vee G) \equiv F$, $F \vee (F \wedge G) \equiv F$ (zakoni apsorpcije)
- $F \Rightarrow G \equiv \neg F \vee G$ (zakon eliminacije implikacije)
- $F \Leftrightarrow G \equiv (F \Rightarrow G) \wedge (G \Rightarrow F) \equiv (\neg F \vee G) \wedge (\neg G \vee F)$ (zakon eliminacije ekvivalencije)
- $(F \Rightarrow G) \equiv (\neg G \Rightarrow \neg F)$ (zakon kontrapozicije)
- $\neg(F \wedge G) \equiv \neg F \vee \neg G$, $\neg(F \vee G) \equiv \neg F \wedge \neg G$ (De Morganovi zakoni)
- $F \wedge \bot \equiv \bot$, $F \wedge \top \equiv F$ (i slični zakoni za ostale veznike)

Ispravnost ovih logičkih ekvivalencija ostavljamo čitaocu za vežbu.

Sledeću važnu teoremu navodimo bez dokaza. Ona nam omogućava da izvodimo zaključke o zadovoljivosti beskonačnih skupova na osnovu zadovoljivosti njihovih konačnih potskupova.

Teorema 2.3 (Kompaktnost iskazne logike)**.** *Skup Δ iskaznih formula je zadovoljiv akko je zadovoljiv svaki njegov konačan potskup.*

Ekvivalentno, ako je Δ nezadovoljiv, tada postoji neki njegov konačan potskup koji je nezadovoljiv.

Posledica 2.1. *$\Delta \models F$, akko postoji neki konačan podskup $\Delta' \subseteq \Delta$ takav da $\Delta' \models F$.*

Dokaz. Važi $\Delta \models F$ akko je $\Delta \cup \{\neg F\}$ nezadovoljiv, a to važi akko postoji konačan potskup od $\Delta \cup \{\neg F\}$ koji je nezadovoljiv. Bez ograničenja opštosti, možemo pretpostaviti da taj potskup sadrži $\neg F$ (uvek ga možemo dodati i potskup će ostati nezadovoljiv), tj. da je taj potskup oblika $\Delta' \cup \{\neg F\}$, gde je $\Delta' \subseteq \Delta$. Ovaj skup je nezadovoljiv akko je $\Delta' \models F$. □

Sledeća teorema daje semantičko opravdanje sintaksne operacije zamene.

Teorema 2.4 (Teorema o zameni)**.** *Ako je $G \equiv H$, tada je $F[G \mapsto H] \equiv F$.*

Dokaz. Indukcijom po strukturi formule. □

Napomena 2.10. Prema prethodnoj teoremi, ako u formuli F zamenjujemo neku potformulu njoj ekvivalentnom potformulom, rezultat će biti formula koja je ekvivalentna sa polaznom formulom F. Ovakve zamene se nazivaju *ekvivalentne transformacije.*

Primer 2.11. Kako važi $p \wedge q \equiv q \wedge p$ (zakon komutacije za konjunkciju), na osnovu teoreme o zameni važi $p \wedge q \Rightarrow r \equiv q \wedge p \Rightarrow r$.

2.1.3 Normalne forme u iskaznoj logici

Definicija 2.11. *Literal* je ili iskazno slovo ili negacija iskaznog slova iz P. *Elementarna disjunkcija* (ili *klauza*) je ili literal ili disjunkcija literala. *Elementarna konjunkcija* je ili literal ili konjunkcija literala. Za formulu kažemo da je u *konjunktivnoj normalnoj formi* (KNF) ako je konjunkcija klauza. Za formulu kažemo da je u *disjunktivnoj normalnoj formi* (DNF) ako je disjunkcija elementarnih konjunkcija.

Teorema 2.5. *Za svaku formulu F postoji njoj ekvivalentna formula u KNF-u (DNF-u) F' koja joj je logički ekvivalentna.*

Dokaz. Teorema sledi iz efektivnog postupka svođenja na KNF (DNF) putem ekvivalentnih transformacija. Pritom, koriste se zakoni eliminacije implikacije i ekvivalencije, De Morganovi zakoni, zakon dvojne negacije, kao i zakoni distribucije. □

Napomena 2.11. Formulu F' iz prethodne teoreme nazivamo KNF-om (DNF-om) formule F.

Primer 2.12. Formula $(p \wedge q) \Rightarrow (r \wedge s)$ se može transformisati ekvivalentnim transformacijama na sledeći način:

$$\begin{array}{ll} (p \wedge q) \Rightarrow (r \wedge s) & \equiv \\ \neg(p \wedge q) \vee (r \wedge s) & \equiv \\ (\neg p \vee \neg q) \vee (r \wedge s) & \end{array}$$

Ova formula je u DNF-u, jer je disjunkcija elementarnih konjunkcija $\neg p$, $\neg q$ i $r \wedge s$. Ako želimo KNF, dalje možemo primeniti distibutivni zakon $F \vee (G \wedge H) \equiv (F \vee G) \wedge (F \vee H)$:

$$\begin{array}{ll} (\neg p \vee \neg q) \vee (r \wedge s) & \equiv \\ (\neg p \vee \neg q \vee r) \wedge (\neg p \vee \neg q \vee s) & \end{array}$$

Sada imamo KNF koji se sastoji iz dve klauze: $\neg p \vee \neg q \vee r$ i $\neg p \vee \neg q \vee s$.

Napomena 2.12. Primetimo da KNF (DNF) neke formule u opštem slučaju nije jedinstven. U praksi smo često zainteresovani za KNF (DNF) minimalne složenosti. Postupci pronalaženja takvog KNF-a (DNF-a) date formule su poznati kao *postupci minimizacije*. Među poznatijim takvim postupcima su *metod Karnoovih mapa* i *metod Kvin-Meklaskog*.

2.1.4 Problem SAT

Definicija 2.12 (Problem SAT)**.** Problem ispitivanja zadovoljivosti iskazne formule u KNF-u naziva se *problem SAT*.

Napomena 2.13. Naziv SAT potiče od engleske reči *Satisfiability*.

Napomena 2.14. Ponekad se u literaturi problemom SAT naziva problem ispitivanja zadovoljivosti proizvoljne iskazne formule. U tom slučaju se problem ispitivanja zadovoljivosti KNF formule eksplicitno označava kao *CNF-SAT* (engl. *conjunctive normal form satisfiability*).

Napomena 2.15. Da bi neka valuacija zadovoljila iskaznu formulu u KNF-u, ona mora istovremeno zadovoljiti sve njene klauze, tj. potrebno je u toj valuaciji bude tačan bar po jedan literal iz svake klauze. Ovo problem SAT čini kombinatornim problemom, koji je, iako veoma jednostavno definisan, često vrlo težak za rešavanje. O ovome ćemo više govoriti u poglavlju 7.

Primer 2.13. Razmotrimo zadovoljivost KNF formule $(p_1 \vee \neg p_3) \wedge (p_2 \vee p_3 \vee \neg p_1) \wedge (\neg p_1 \vee \neg p_3 \vee p_2) \wedge (\neg p_1 \vee \neg p_2) \wedge (p_1 \vee p_3)$. Pretpostavimo da postoji valuacija v koja zadovoljava ovu formulu. Ispitaćemo slučajeve po vrednosti $v(p_3)$. Pretpostavimo da je $v(p_3) = 1$. Tada je $I_v(\neg p_3) = 0$, pa kako bi klauza $p_1 \vee \neg p_3$ bila zadovoljena, mora da važi $v(p_1) = 1$. Dalje, kako je $I_v(\neg p_1) = 0$, da bi klauza $\neg p_1 \vee \neg p_2$ bila zadovoljena, mora da važi $I_v(\neg p_2) = 1$, tj. $v(p_2) = 0$. Međutim, sada je $I_v(\neg p_1 \vee \neg p_3 \vee p_2) = 0$. Dakle, ne može da važi $v(p_3) = 1$. Pretpostavimo sada da važi $v(p_3) = 0$. Sada, zbog klauze $p_1 \vee p_3$ mora da važi $v(p_1) = 1$. Otuda je ponovo $v(p_2) = 0$ (zbog klauze $\neg p_1 \vee \neg p_2$), pa će važiti $I_v(p_2 \vee p_3 \vee \neg p_1) = 0$. Dakle, ni za $v(p_3) = 1$ ni za $v(p_3) = 0$ formula ne može biti zadovoljena takvom valuacijom. Otuda formula nije zadovoljiva.

Napomena 2.16. Jedan od najpoznatijih postupaka za ispitivanje zadovoljivosti KNF formule (tj. za rešavanje problema SAT) je *DPLL procedura* (engl. *Davis–Putnam–Logemann–Loveland*). Zasnovan je na pretrazi sa elementima rezonovanja, od kojih je najznačajniji *jedinična propagacija* (ako su u klauzi svi

literali osim jednog netačni, tada taj preostali literal mora biti tačan). Postupak demonstriran u prethodnom primeru predstavlja jednostavan primer primene DPLL procedure.

Problem SAT je izuzetno primenljiv u praksi, jer se mnogi problemi mogu svesti na instance ovog problema. O tome govore sledeći primeri.

Primer 2.14 (Bojenje grafa). Pretpostavimo da je dat neusmeren graf $G = (V, E)$, sa skupom čvorova $V = \{1, \ldots, n\}$ i skupom grana $E \subseteq \{\{u, v\} \mid u, v \in V, u \neq v\}$. Potrebno je svaki čvor grafa obojiti jednom od tri boje *crvena*, *zelena* ili *plava*, tako da nikoja dva susedna čvora u i v (tj. čvora za koje postoji grana $\{u, v\} \in E$) nisu obojena istom bojom.

Ovaj problem se može jednostavno svesti na problem SAT. Neka iskazno slovo c_i znači „čvor i je obojen crvenom bojom". Slično neka p_i i z_i imaju isto značenje za plavu i zelenu boju, respektivno. Svaki čvor mora biti obojen tačno jednom bojom, što možemo opisati sledećim klauzama:

$$(c_i \vee p_i \vee z_i) \wedge (\neg c_i \vee \neg p_i) \wedge (\neg c_i \vee \neg z_i) \wedge (\neg p_i \vee \neg z_i)$$

Ovakve klauze formiramo za svako $i \in \{1, \ldots, n\}$. Dalje, za svaka dva susedna čvora i i j imaćemo sledeće klauze:

$$(\neg c_i \vee \neg c_j) \wedge (\neg p_i \vee \neg p_j) \wedge (\neg z_i \vee \neg z_j)$$

Ove klauze zabranjuju da čvorovi i i j budu obojeni istom bojom. Konjunkcija svih navedenih klauza daje KNF formulu koja će biti zadovoljiva akko je moguće obojiti graf na opisani način. Pritom, svaka valuacija koja zadovoljava ovu formulu će odgovarati jednom ispravnom bojenju grafa.

Primer 2.15. Tri đaka, Miloš, Zoran i Petar uče strane jezike: engleski i nemački. Ako se zna da bar dvojica govore engleski, i bar dvojica govore nemački, tada postoji učenik koji govori oba jezika. Dokazati.

Opisani problem možemo predstaviti iskaznom formulom. Neka su m_e i m_n redom iskazna slova sa značenjem „Miloš govori engleski" i „Miloš govori nemački". Sa sličnim značenjem, uvodimo i iskazna slova z_e i z_n (za Zorana) i p_e i p_n (za Petra). Prethodno tvrđenje se može predstaviti kao sledeća iskazna formula:

$$\begin{array}{ll} ((m_e \wedge z_e) \vee (m_e \wedge p_e) \vee (z_e \wedge p_e)) & \wedge \\ ((m_n \wedge z_n) \vee (m_n \wedge p_n) \vee (z_n \wedge p_n)) & \Rightarrow \\ ((m_e \wedge m_n) \vee (z_e \wedge z_n) \vee (p_e \wedge p_n)) & \end{array}$$

Potrebno je dokazati da je ova formula tautologija. Da bismo to uradili, možemo dokazati da je negacija ove formule nezadovoljiva. Ova formula je oblika $A \wedge B \Rightarrow C$, pa je njena negacija oblika $A \wedge B \wedge \neg C$, tj. imamo formulu:

$$\begin{array}{ll} ((m_e \wedge z_e) \vee (m_e \wedge p_e) \vee (z_e \wedge p_e)) & \wedge \\ ((m_n \wedge z_n) \vee (m_n \wedge p_n) \vee (z_n \wedge p_n)) & \wedge \\ \neg\,((m_e \wedge m_n) \vee (z_e \wedge z_n) \vee (p_e \wedge p_n)) & \end{array}$$

Svođenjem na logički ekvivalentnu KNF formu, dobijamo:

$$\begin{array}{ll} (m_e \vee z_e) \wedge (m_e \vee p_e) \wedge (z_e \vee p_e) & \wedge \\ (m_n \vee z_n) \wedge (m_n \vee p_n) \wedge (z_n \vee p_n) & \wedge \\ (\neg m_e \vee \neg m_n) \wedge (\neg z_e \vee \neg z_n) \wedge (\neg p_e \vee \neg p_n) & \end{array}$$

Prilikom svođenja na KNF, primenjivali smo i zakon idempotencije $A \vee A \equiv A$ i apsorpcije $(A \vee B) \wedge A \equiv A$ radi uprošćavanja. Za dobijenu KNF formulu se može utvrditi da je nezadovoljiva (proveriti za vežbu!), odakle je polazna formula tautologija.

Napomena 2.17. Prethodni primer je demonstrirao postupak *dokazivanja tautologičnosti pobijanjem* (engl. *refutation*). Ovaj postupak se zasniva na teoremi 2.1, tj. činjenici da je formula F tautologija akko je njena negacija $\neg F$ nezadovoljiva. Time se problem tautologičnosti svodi na problem zadovoljivosti, a zatim svođenjem na KNF na problem SAT.

Softverski alati za rešavanje problema SAT su poznati pod nazivom *SAT rešavači* (engl. *SAT solvers*). Moderni SAT rešavači su zasnovani na algoritmima koji su mnogo efikasniji od metode istinitosnih tablica (poput CDCL algoritma). SAT rešavači zasnovani na CDCL algoritmu su u stanju da za relativno kratko vreme ispitaju zadovoljivost KNF formula koje sadrže na hiljade iskaznih slova, kao i desetine hiljada klauza. Zahvaljujući tome, mnogi teški kombinatorni problemi su u prethodnih par decenija rešeni upravo svođenjem na SAT.

2.2 Logika prvog reda

Logika prvog reda proširuje iskaznu logiku time što omogućava prezicniji opis atomičnih iskaza, uz pomoć funkcijskih i predikatskih simbola koji u datom kontekstu mogu imati specifično značenje. Na primer, umesto da elementarne iskaze predstavljamo iskaznim atomima $p, q, \ldots$ (za koje samo možemo da zadamo da li su tačni ili netačni), možemo imati atome poput $a < b$ ili $(a + b)|c$ ili $parno(a)$, čije značenje zavisi od značenja pridruženog simbolima koji se u njima pojavljuju. Takođe, logika prvog reda uvodi pojam *promenljivih* koje se mogu *kvantifikovati*, čime se omogućava izražavanje svojstava poput ,,za svako x važi...'' ili ,,postoji x za koje važi...''.

2.2.1 Sintaksa logike prvog reda

Definicija 2.13. *Signatura* (ili *jezik*) prvog reda je uređena trojka $\mathcal{L} = (\Sigma, \Pi, ar)$ pri čemu je Σ neprazan skup *funkcijskih simbola*, Π je neprazan skup *predikatskih (relacijskih) simbola* $(\Pi \cap \Sigma = \varnothing)$, a $ar : \Sigma \cup \Pi \to \mathbb{N}$ je funkcija koja svakom simbolu iz $\Sigma \cup \Pi$ pridružuje *arnost*.

Definicija 2.14. Neka je data signatura $\mathcal{L} = (\Sigma, \Pi, ar)$ i neprazan skup *promenljivih* V. *Term* nad $\mathcal{L}$ se dobija konačnom primenom sledećih pravila:

- promenljiva $x \in V$ je term
- ako je $a \in \Sigma$ i $ar(a) = 0$, tada je a term
- ako je $f \in \Sigma$ i $ar(f) = n > 0$, a $t_1, \ldots, t_n$ su termovi, tada je i $f(t_1, \ldots, t_n)$ term.

Atom nad $\mathcal{L}$ je ili $p \in \Pi$ takvo da je $ar(p) = 0$, ili je oblika $p(t_1, \ldots, t_n)$, pri čemu je $p \in \Pi$ i $ar(p) = n > 0$, a $t_1, \ldots, t_n$ su termovi nad $\mathcal{L}$. *Formula* nad $\mathcal{L}$ se dobija konačnom primenom sledećih pravila:

- $\top$ i $\bot$ su formule

- atom nad $\mathcal{L}$ je formula
- ako je F formula, tada je i $\neg F$ formula
- ako su F i G formule, tada su i $F \wedge G$, $F \vee G$, $F \Rightarrow G$ i $F \Leftrightarrow G$ formule
- ako je F formula, a x promenljiva iz V, tada su $\exists x.F$ i $\forall x.F$ formule.

Napomena 2.18. Iz prethodne definicije vidimo koji je smisao *arnosti*: arnost simbola određuje na koliko se podtermova on može primeniti. Specijalan slučaj su funkcijski simboli arnosti 0 – oni se ne primenjuju na podtermove i sami za sebe predstavljaju termove (dakle, ne pišemo $a()$, već samo a). Takve simbole nazivamo *simbolima konstanti*. Slično, predikatski simboli arnosti 0 se ne primenjuju na termove, već sami za sebe predstavljaju atome, pa samim tim i formule. Ovakvi atomi odgovaraju iskaznim slovima u iskaznoj logici. Samim tim, svaka iskazna formula je i ujedno i formula prvog reda, u skladu sa ovom definicijom. Otuda se logika prvog reda može smatrati uopštenjem iskazne logike, gde sada dozvoljavamo i složeniju strukturu atoma.

Primer 2.16. Neka je data signatura $\mathcal{L} = (\Sigma, \Pi, ar)$, gde je $\Sigma = \{a, f, g\}$, pri čemu je $ar(a) = 0$, $ar(f) = 1$ i $ar(g) = 2$. Neka su x i y promenljive iz V. Tada su a, x, $f(a)$, $g(a, f(x))$, $g(g(x, y), f(f(a)))$ termovi nad $\mathcal{L}$. Sa druge strane, $f(x, y)$ nije ispravan term nad $\mathcal{L}$ jer f nije arnosti dva. Dalje, ako je $\Pi = \{p, q\}$, gde je $ar(p) = 1$, a $ar(q) = 2$, tada su $p(f(a)$ i $q(f(a), g(x, y))$ atomi nad $\mathcal{L}$, dok $p(x, y)$ to nije, s obzirom da je arnost simbola p jednaka 1.

Napomena 2.19. Kao i u iskaznoj logici, i ovde je potrebno definisati prioritete i asocijativnosti veznika, kako bi konkretna sintaksa bila jednoznačna. Za iskazne veznike zadržaćemo iste prioritete i asocijativnosti kao i u iskaznoj logici. Što se tiče kvantifikatora $(\exists, \forall)$, smatraćemo da oni imaju najniži mogući prioritet. Otuda, formula $\exists x.p(x) \wedge q(x, a)$ će biti sintaksno identična formuli $\exists x.(p(x) \wedge q(x, a))$. Sa druge strane, ako želimo da se kvantifikator odnosi samo na $p(x)$, moramo ga staviti u zagrade: $(\exists x.p(x)) \wedge q(x, a)$.

Napomena 2.20. Ponekad se i funkcijski i predikatski simboli mogu zapisivati u *operatorskoj* notaciji. Na primer, ako imamo funkcijski simbol + arnosti 2, obično umesto $+(a, b)$ pišemo $a + b$ (tj. zapisujemo ga kao infiksni operator). Slično, ako imamo funkcijski simbol ? arnosti 1, tada umesto $?(a)$ često pišemo $?a$ ili $a?$ (tj. zapisujemo ga kao prefiksni ili sufiksni operator). U slučaju da imamo takve simbole u signaturi, moramo za njih takođe definisati prioritete i asocijativnosti, kako bi konkretna sintaksa takvih termova i atoma bila jednoznačna. Na primer, ako imamo binarne funkcijske operatore + i $\cdot$, tada moramo znati koji su im prioriteti, ako želimo da konkretna sintaksa terma $a + b \cdot c$ bude jednoznačna.

Definicija 2.15. *Slobodna* i *vezana* pojavljivanja promenljive u formuli definišemo na sledeći način:

- ako je A atom, tada su sva pojavljivanja promenljivih u A slobodna
- sva slobodna (vezana) pojavljivanja promenljivih u formuli F su slobodna (vezana) i u $\neg F$
- sva slobodna (vezana) pojavljivanja promenljivih u formulama F i G su slobodna (vezana) i u formulama $F \wedge G$, $F \vee G$, $F \Rightarrow G$ i $F \Leftrightarrow G$

- sva slobodna (vezana) pojavljivanja promenljivih u formuli F su slobodna (vezana) i u formulama $\exists x.F$ i $\forall x.F$, izuzev slobodnih pojavljivanja promenljive x u formuli F koja su sada vezana kvantifikatorom u formuli $\exists x.F$ (odnosno $\forall x.F$).

Napomena 2.21. Iz gornje definicije sledi da su sve promenljive slobodne, osim onih koje su vezane kvantifikatorom. Takođe, svaki kvantifikator vezuje samo *slobodna pojavljivanja* kvantifikovane promenljive u potformuli. Na primer, ako imamo formulu $\forall x.p(x) \Rightarrow (\exists x.q(x))$, spoljni kvantifikator $\forall x$ vezuje samo pojavljivanje promenljive x u $p(x)$, s obzirom da je pojavljivanje u $q(x)$ već bilo vezano i u potformuli unutrašnjim kvantifikatorom $\exists x$.

Primer 2.17. U formuli $\forall x.p(x, y) \Rightarrow (\exists y.q(y))$ pojavljivanje promenljive x je vezano kvantifikatorom $\forall x$. Sa druge strane, prvo pojavljivanje promenljive y (u $p(x, y)$) je slobodno, dok je drugo pojavljivanje (u $q(y)$) vezano kvantifikatorom $\exists y$.

Primer 2.18. U formuli $(\forall x.p(x)) \wedge (\forall x.q(x))$ imamo dva pojavljivanja promenljive x i oba su vezana, ali različitim kvantifikatorima.

Definicija 2.16. Formula je *zatvorena* (ili *rečenica*) ako ne sadrži slobodna pojavljivanja promenljivih.

Primer 2.19. U formuli $\forall x.\forall y.p(x, y) \Rightarrow (\exists z.q(x, z) \wedge q(z, y))$, sve promenljive su vezane odgovarajućim kvantifikatorima. Dakle, ova formula je rečenica.

Definicija 2.17. Za formulu (ili term) kažemo da je bazna, ako ne sadrži promenljive (ni slobodne ni vezane).

Napomena 2.22. Svaka bazna formula je rečenica, ali obrnuto ne mora da važi.

Primer 2.20. Term $f(a)$ je bazni, dok $f(x)$ to nije. Slično, formula $p(f(a)) \Rightarrow q(a) \vee q(b)$ je bazna, dok $\forall x.\forall y.p(f(x)) \Rightarrow q(x) \vee q(y)$ to nije (ako je rečenica).

Napomena 2.23. Umesto $\forall x_1.\forall x_2. \ldots \forall x_n.F$ ćemo kraće pisati $\forall x_1 x_2 \ldots x_n.F$. Slično za egzistencijalni kvantifikator.

Definicija 2.18. *Zamena* formule G formulom H u formuli F (u oznaci $F[G \mapsto H]$ se definiše na sledeći način:

- $F[G \mapsto H] = H$, ako je $F = G$
- $\top[G \mapsto H] = \top$, $\bot[G \mapsto H] = \bot$
- $p[G \mapsto H] = p$, ako je $p \in P$ i $G \neq p$
- $(\neg F)[G \mapsto H] = \neg F[G \mapsto H]$
- $(F_1 \wedge F_2)[G \mapsto H] = F_1[G \mapsto H] \wedge F_2[G \mapsto H]$
- $(F_1 \vee F_2)[G \mapsto H] = F_1[G \mapsto H] \vee F_2[G \mapsto H]$
- $(F_1 \Rightarrow F_2)[G \mapsto H] = F_1[G \mapsto H] \Rightarrow F_2[G \mapsto H]$
- $(F_1 \Leftrightarrow F_2)[G \mapsto H] = F_1[G \mapsto H] \Leftrightarrow F_2[G \mapsto H]$
- $(\forall x.F)[G \mapsto H] = \forall x.F[G \mapsto H]$

- $(\exists x.F)[G \mapsto H] = \exists x.F[G \mapsto H]$.

Napomena 2.24. Prethodna definicija je gotovo identična definiciji zamene u iskaznoj logici (definicija 2.3) uz dodatak slučaja koji pokriva kvantifikatore. Dakle, i ovde se zamena vrši tako što se sva pojavljivanja potformule G u formuli F zamenjuju formulom H.

Pored zamene *formule formulom*, u logici prvog reda definišemo i zamenu *promenljive termom*.

Definicija 2.19. *Zamena promenljive x termom t u termu s (formuli F)*, u oznaci $s[x \mapsto t]$ ($F[x \mapsto t]$) definiše se na sledeći način:

- $x[x \mapsto t] = t$
- $y[x \mapsto t] = y$ ($y \in V$ i $y \neq x$)
- $a[x \mapsto t] = a$, gde je a simbol konstante
- $f(t_1, \ldots, t_n)[x \mapsto t] = f(t_1[x \mapsto t], \ldots, t_n[x \mapsto t])$, gde je f funkcijski simbol arnosti n
- $\top[x \mapsto t] = \top$
- $\bot[x \mapsto t] = \bot$
- $p[x \mapsto t] = p$, gde je p iskazni atom
- $p(t_1, \ldots, t_n)[x \mapsto t] = p(t_1[x \mapsto t], \ldots, t_n[x \mapsto t])$, gde je p predikatski simbol arnosti n
- $(\neg F)[x \mapsto t] = \neg F[x \mapsto t]$
- $(F \wedge G)[x \mapsto t] = F[x \mapsto t] \wedge G[x \mapsto t]$
- $(F \vee G)[x \mapsto t] = F[x \mapsto t] \vee G[x \mapsto t]$
- $(F \Rightarrow G)[x \mapsto t] = F[x \mapsto t] \Rightarrow G[x \mapsto t]$
- $(F \Leftrightarrow G)[x \mapsto t] = F[x \mapsto t] \Leftrightarrow G[x \mapsto t]$
- $(Qx.F)[x \mapsto t] = Qx.F$
- $(Qy.F)[x \mapsto t] = Qy.F[x \mapsto t]$, gde je $y \neq x$ i t ne sadrži promenljivu y
- $(Qy.F)[x \mapsto t] = Qy'.F[y \mapsto y'][x \mapsto t]$, gde je $y \neq x$, y je sadržana u t, a y' nije sadržana ni u $Qy.F$ ni u t

gde je $Q \in \{\forall, \exists\}$.

Napomena 2.25. Iz prethodne definicije, zaključujemo da se zamena promenljive x termom t vrši tako što se sva *slobodna* pojavljivanja promenljive x zamene termom t. Pritom, važno je da ni jedna od promenljivih koje sadrži t ne postanu vezane prilikom zamene. Da se to ne bi dogodilo, može biti neophodno izvršiti *preimenovanje vezane promenljive*, što je pokriveno poslednjim slučajem u gornjoj definiciji (y se preimenuje u novu promenljivu y', a zatim se izvrši zamena x sa t).

Pod preimenovanjem vezane promenljive (koje nazivamo i *α-konverzija*) u formuli $Qy.F$ ($Q \in \{\forall, \exists\}$) podrazumevamo sledeću transformaciju:

$$Qy.F \longmapsto Qz.F[y \mapsto z]$$

gde je z promenljiva koja se ne pojavljuje u $Qy.F$. Za formule F i G kažemo da su *α-ekvivalentne* (u oznaci $F \equiv_\alpha G$) ako se jedna može dobiti od druge samo primenom α-konverzija. Na primer, formule $\forall x.p(x)$ i $\forall y.p(y)$ su α-ekvivalentne. Slično, formule $\forall x.p(x) \Rightarrow (\exists y.q(x, y))$ i $\forall u.p(u) \Rightarrow (\exists v.q(u, v))$ su α-ekvivalentne.

Primer 2.21. Pretpostavimo da želimo da izvršimo zamenu $[x \mapsto f(y)]$ u formuli $\forall y.p(y) \Rightarrow q(x, y) \vee (\exists x.h(x, y))$. Zameniće se samo prvo pojavljivanje promenljive x, jer je drugo pojavljivanje (u $h(x, y)$) vezano egzistencijalnim kvantifikatorom. Dodatno, prilikom zamene prvog pojavljivanja promenljive x sa $f(y)$, promenljiva y u $f(y)$ bi postala vezana spoljnim univerzalnim kvantifikatorom, tj. dobili bismo $\forall y.p(y) \Rightarrow q(f(y), y) \vee (\exists x.h(x, y))$. Prema gornjoj definiciji, ovo nije ispravno, već je potrebno prvo preimenovati vezanu promenljivu y nekim novim imenom koje se do tada nije pojavljivalo u formuli (npr. sa z). Dakle, najpre imamo α-konverziju polazne formule u $\forall z.p(z) \Rightarrow q(x, z) \vee (\exists x.h(x, z))$. Sada možemo bezbedno zameniti slobodno pojavljivanje promenljive x sa $f(y)$, pa imamo: $\forall z.p(z) \Rightarrow q(f(y), z) \vee (\exists x.h(x, z))$.

2.2.2 Semantika logike prvog reda

Definicija 2.20 ($\mathcal{L}$-struktura)**.** *Struktura* nad datom signaturom $\mathcal{L}$ (*$\mathcal{L}$-struktura*) $\mathcal{D} = (D, \_^{\mathcal{D}})$ se sastoji iz nepraznog skupa D (koji nazivamo *domen* ili *univerzum* strukture $\mathcal{D}$) i preslikavanja $\_^{\mathcal{D}}$ koje:

- svakom simbolu konstante a signature $\mathcal{L}$ pridružuje fiksirani element $a^{\mathcal{D}} \in D$
- svakom funkcijskom simbolu f arnosti $n > 0$ signature $\mathcal{L}$ pridružuje *funkciju* $f^{\mathcal{D}} : D^n \to D$
- svakom iskaznom atomu p signature $\mathcal{L}$ pridružuje vrednost $p^{\mathcal{D}}$ iz skupa $\{0, 1\}$ (*netačno* ili *tačno*)
- svakom predikatskom simbolu p arnosti $n > 0$ signature $\mathcal{L}$ pridružuje *relaciju* $p^{\mathcal{D}}$ dužine n nad D, tj. $p^{\mathcal{D}} \subseteq D^n$.

Napomena 2.26. $\mathcal{L}$-struktura zapravo određuje interpretaciju jezika $\mathcal{L}$, tako što fiksira skup i funkcije (operacije) i relacije nad tim skupom kojima će se interpretirati funkcijski i predikatski simboli jezika. Na primer, ako u imamo jezik $\mathcal{L}$ gde imamo konstante 0 i 1, binarne funkcijske simble $+$ i $\cdot$ i binarni predikatski simbol $\leq$, jedna $\mathcal{L}$-struktura bi mogla da bude $(\mathbb{N}, 0, 1, +, \cdot, \leq)$, gde su sa $0, 1, +, \cdot, \leq$ redom označeni prirodni brojevi nula i jedan, operacije sabiranja i množenja prirodnih brojeva, kao i relacija ,,manje ili jednako" nad skupom prirodnih brojeva.

Definicija 2.21 (Valuacija prvog reda)**.** *Valuacija prvog reda* nad $\mathcal{L}$-strukturom $\mathcal{D} = (D, \_^{\mathcal{D}})$ je funkcija $v : V \to D$ koja svakoj promenljivoj iz V pridružuje jedan element domena D.

Napomena 2.27. $\mathcal{L}$-struktura nam interpretira simbole signature $\mathcal{L}$. Međutim, u našim formulama se pojavljuju i promenljive koje nisu deo signature. Valuacija nam služi da odredi značenje promenljivih koje se u formuli javljaju.

Napomena 2.28. Treba jasno razlikovati valuaciju prvog reda od iskazne valuacije. Iskazna valuacija interpretira iskazne atome kao tačne ili netačne. U tom smislu, uopštenje iskazne valuacije je zapravo $\mathcal{L}$-struktura, koja takođe, između ostalog, interpretira iskazne atome u jeziku $\mathcal{L}$ (ako postoje). Sa druge strane, valuacija prvog reda interpretira promenljive prvog reda kao elemente domena D. Nesrećna okolnost je to što se za ova dva suštinski različita pojma u literaturi obično koristi isti naziv, što može da dovede do zabune.

Definicija 2.22 (Interpretacija)**.** Neka je data $\mathcal{L}$-struktura $\mathcal{D}$ i valuacija prvog reda v. *Interpretacija* (ili *vrednost*) terma t u strukturi $\mathcal{D}$ i valuaciji v (u oznaci $\mathcal{D}_v(t)$) se definiše na sledeći način:

- $\mathcal{D}_v(x) = v(x)$, za $x \in V$
- $\mathcal{D}_v(a) = a^{\mathcal{D}}$, ako je a simbol konstante iz $\mathcal{L}$
- $\mathcal{D}_v(f(t_1, \ldots, t_n)) = f^{\mathcal{D}}(\mathcal{D}_v(t_1), \ldots, \mathcal{D}_v(t_n))$, ako je f funkcijski simbol arnosti $n > 0$ iz $\mathcal{L}$.

Interpretacija (ili *vrednost* formule F u strukturi $\mathcal{D}$ i valuaciji v (u oznaci $\mathcal{D}_v(F)$) se definiše na sledeći način:

- $\mathcal{D}_v(\top) = 1$
- $\mathcal{D}_v(\bot) = 0$
- $\mathcal{D}_v(p) = p^{\mathcal{D}}$, ako je p iskazni atom iz $\mathcal{L}$
- $\mathcal{D}_v(p(t_1, \ldots, t_n)) = 1$ akko je $(\mathcal{D}_v(t_1), \ldots, \mathcal{D}_v(t_n)) \in p^{\mathcal{D}}$, ako je p predikatski simbol arnosti $n > 0$ iz $\mathcal{L}$
- $\mathcal{D}_v(\neg F) = 1$ akko $\mathcal{D}_v(F) = 0$
- $\mathcal{D}_v(F \wedge G) = 1$ akko je $\mathcal{D}_v(F) = 1$ i $\mathcal{D}_v(G) = 1$
- $\mathcal{D}_v(F \vee G) = 1$ akko je $\mathcal{D}_v(F) = 1$ ili $\mathcal{D}_v(G) = 1$
- $\mathcal{D}_v(F \Rightarrow G) = 1$ akko je $\mathcal{D}_v(F) = 0$ ili $\mathcal{D}_v(G) = 1$
- $\mathcal{D}_v(F \Leftrightarrow G) = 1$ akko je $\mathcal{D}_v(F) = \mathcal{D}_v(G)$
- $\mathcal{D}_v(\forall x.F) = 1$ akko za svaku valuaciju v' koja je dobijena od v samo eventualnom izmenom vrednosti promenljive x, važi $\mathcal{D}_{v'}(F) = 1$
- $\mathcal{D}_v(\exists x.F) = 1$ akko postoji valuacija v' koja je dobijena od v samo eventualnom izmenom vrednosti promenljive x, takva da je $\mathcal{D}_{v'}(F) = 1$.

Napomena 2.29. Iz prethodne definicije vidimo da se termovi interpretiraju elementima domena D, dok se formule (kao i u iskaznoj logici) interpretiraju kao *tačne* ili *netačne*.

Primer 2.22. Neka je dat jezik $\mathcal{L}$ koji sadrži konstante $0, 1, 2, \ldots$, funkcijske simbole $+$ i $\cdot$ arnosti dva, kao i predikatske simbole $=$ i $\leq$ arnosti 2 (svi binarni simboli se zapisuju u operatorskoj notaciji, tj. infiksno). Neka je $\mathcal{D}$ standardna struktura prirodnih brojeva $\mathbb{N}$ sa uobičajenim interpretacijama gornjih simbola (koje ćemo, uz zloupotrebu notacije, označavati na isti način kao i same simbole). Neka je, dalje, valuacija v takva da je $v(x) = 3$ i $v(y) = 5$. Sada imamo:

- $\mathcal{D}_v(x + 1) = \mathcal{D}_v(x) + \mathcal{D}_v(1) = v(x) + 1 = 3 + 1 = 4$
- $\mathcal{D}_v(x \cdot y) = \mathcal{D}_v(x) \cdot \mathcal{D}_v(y) = v(x) \cdot \mathcal{D}(y) = 3 \cdot 5 = 15$
- $\mathcal{D}_v(y \leq 1) = 0$ (netačno), zato što $(\mathcal{D}_v(y), \mathcal{D}_v(1)) = (v(y), 1) = (5, 1) \notin \leq$
- $\mathcal{D}_v(\exists y.x + y = 3) = 1$ (tačno), zato što postoji valuacija v' koja se od v može eventualno razlikovati na y takva da je $\mathcal{D}_{v'}(x + y = 3) = 1$. Zaista, možemo uzeti valuaciju v' za koju je $v'(y) = 0$. Sada je $\mathcal{D}_{v'}(x+y = 3) = 1$, jer je $(\mathcal{D}_{v'}(x + y), \mathcal{D}_{v'}(3)) = (v'(x) + v'(y), 3) = (3 + 0, 3) = (3, 3) \in =$ (s obzirom da se simbol $=$ interpretira kao relacija $\{(x, x) \mid x \in \mathbb{N}\}$, tj. kao relacija *jednakosti* nad $\mathbb{N}$)
- $\mathcal{D}_v(\forall x.\neg(x + y \leq 2)) = 1$ (tačno), zato što za svaku valuaciju v' koja se od v može razlikovati eventualno na x važi $\mathcal{D}_{v'}(\neg(x + y \leq 2)) = 1$. Zaista, kako god da odaberemo $v'(x)$, važiće da *nije* $v'(x) + v'(y) \leq 2$ (jer je $v'(y) = v(y) = 5$).
- $\mathcal{D}_v(\forall xy.x+y = y+x) = 1$ (tačno), zato što za sve valuacije v' koje se od v mogu eventualno razlikovati na x i na y važi da $v'(x)+v'(y) = v'(y)+v'(x)$
- $\mathcal{D}_v(\forall xy.x \leq y \Rightarrow x + 1 \leq y + 1) = 1$ (tačno), sličnom analizom kao i u prethodnom primeru

Napomena 2.30. Primetimo da vrednost formule F u strukturi $\mathcal{D}$ i valuaciji v ne zavisi od vrednosti koje vezane promenljive formule F imaju u valuaciji v, već samo od vrednosti slobodnih promenljivih formule F. Na primer, u formuli $\exists y.x + y = 3$ bila nam je od značaja vrednost $v(x)$, jer je x slobodna promenljiva. U gornjem primeru, imali smo da je $v(x) = 3$, pa je formula bila tačna. Sa druge strane, da smo imali $v(x) = 10$, tada bi ova formula bila netačna, jer ne postoji vrednost iz $\mathbb{N}$ takva da sabrana sa 10 daje 3. Dakle, vrednost formule zavisi od valuacije $v(x)$ slobodne promenljive x. Sa druge strane, vrednost formule ne zavisi od vrednosti $v(y)$, jer je y vezana. Kod vezanih promenljivih, mi posmatramo valuacije v' kod kojih se vrednost $v'(y)$ može razlikovati od $v(y)$, što samu vrednost $v(y)$ čini irelevantnom.

Posledično, vrednost zatvorene formule (rečenice) ne zavisi uopšte od valuacije v. Na primer, formula $\forall xy.x \leq y \Rightarrow x + 1 \leq y + 1$ je u strukturi prirodnih brojeva tačna, nezavisno od odabrane valuacije v. Zbog toga ćemo za zatvorene formule obično pisati samo $\mathcal{D}(F)$ umesto $\mathcal{D}_v(F)$, jer nam valuacija nije relevantna.

Definicija 2.23. Za uređeni par $(\mathcal{D}, v)$ kažemo da *zadovoljava* formulu F (ili da je *model* formule F) ako je $\mathcal{D}_v(F) = 1$. Za formulu F kažemo da je *zadovoljiva* ako ima bar jedan model. Specijalno, rečenica F je zadovoljiva ako postoji struktura $\mathcal{D}$ takva da je $\mathcal{D}(F) = 1$ (takva struktura je model rečenice F).

Za formulu koja nije zadovoljiva kažemo da je *nezadovoljiva*.

Definicija 2.24. Formula F je *valjana* (ili *validna*) ako je tačna za svako $(\mathcal{D}, v)$. Specijalno, rečenica je valjana ako je tačna u svakoj strukturi $\mathcal{D}$.

Za formulu koja nije valjana kažemo da je *poreciva*.

Teorema 2.6. *Formula F je valjana akko je $\neg F$ nezadovoljiva. Formula F je zadovoljiva akko je $\neg F$ poreciva.*

Napomena 2.31. Pojmovi zadovoljivosti skupa formula, logičke posledice i logičke ekvivalencije se definišu na postpuno isti način kao i u iskaznoj logici.

Primer 2.23. Može se pokazati da je $\forall x.F \equiv \forall z.F[x \mapsto z]$ gde je z nova promenljiva koja se ne pojavljuje u F. Slično važi i za egzistencijalni kvantifikator. Dakle, α-konverzijom se dobija logički ekvivalentna formula.

Primer 2.24. Može se pokazati da važi: $\neg(\forall x.F) \equiv \exists x.\neg F$ i $\neg(\exists x.F) \equiv \forall x.\neg F$. Ove logičke ekvivalencije su poznate i kao *De Morganovi zakoni* za kvantifikatore.

U logici prvog reda takođe važi teorema kompaktnosti, kao i u iskaznoj logici. Ovu teoremu takođe navodimo bez dokaza.

Teorema 2.7 (Kompaktnost logike prvog reda)**.** *Skup Δ formula prvog reda je zadovoljiv akko je zadovoljiv svaki njegov konačan potskup.*

Ekvivalentno, ako je Δ nezadovoljiv, tada postoji neki njegov konačan potskup koji je nezadovoljiv.

U logici prvog reda takođe važi teorema o zameni koju navodimo bez dokaza.

Teorema 2.8 (Teorema o zameni)**.** *Ako je $G \equiv H$, tada je $F[G \mapsto H] \equiv F$.*

Teorema o zameni se koristi kao opravdanje za *ekvivalentne transformacije* kojima se formula može svesti na pogodniji oblik, čuvajući pritom logičku ekvivalentnost.

Kada je u pitanju zamena promenljive termom, tu imamo sledeću teoremu.

Teorema 2.9. *Neka je E term ili formula. Za svaku strukturu $\mathcal{D}$ i valuaciju v važi da je $\mathcal{D}_v(E[x \mapsto t]) = \mathcal{D}_{v'}(E)$, gde je v' valuacija koja se poklapa sa v svuda osim na x, gde važi da je $v'(x) = \mathcal{D}_v(t)$.*

Dokaz. Indukcijom po strukturi formule i termova. □

Napomena 2.32. Intuitivno, ova teorema kaže da kada zamenimo promenljivu x termom t, dobijena formula (ili term) se interpretira onako kako bi se polazna formula (ili term) interpretirala kada bismo tu promenljivu x interpretirali onako kako interpretiramo t.

Primetimo da je, kako bi gornja teorema važila, neophodno da se prilikom zamene x sa t zamenjuju samo *slobodna* pojavljivanja promenljive x, kao i da se tom prilikom ni jedna promenljiva iz t ne vezuje drugim kvantifikatorima u formuli. Ovo je zato što kvantifikatori menjaju semantiku promenljivih koje vezuju. Upravo zbog toga smo zamenu i definisali na takav način, jer smo želeli da ona ima semantički smisao iskazan gornjom teoremom.

2.2.3 Normalne forme u logici prvog reda

U logici prvog reda, pojmovi elementarne disjunkcije (klauze), elementarne konjunkcije, KNF-a i DNF-a se definišu na isti način kao u iskaznoj logici. Jedina razlika je u tome što literali više nisu samo iskazni atomi, već atomi prvog reda ili njihove negacije (npr. $x + y \leq 5$, $\neg(x \cdot y = 2)$ i sl.). Analogno teoremi 2.5, i u logici prvog reda se može pokazati da se bilo koja formula koja *ne sadrži* kvantifikatore može svesti na logički ekvivalentnu formulu u KNF-u ili DNF-u. Međutim, prisustvo kvantifikatora u formuli komplikuje proces transformacije formule. Može se pokazati sledeća teorema:

Teorema 2.10 (Prenex)**.** *Za proizvoljnu formulu F postoji njoj logički ekvivalentna formula u* prenex normalnoj formi*, tj. formula oblika:*

$$Q_1 x_1 . Q_2 x_2 . \ldots Q_n x_n . F'$$

gde je $Q_i \in \{\exists, \forall\}$ a formula F' ne sadrži kvantifikatore.

Dokaz. Da bi se formula transformisala u prenex normalnu formu, potrebno je ,,izvući" kvantifikatore ispred formule. Najpre se iz formule eliminišu implikacije i ekvivalencije na uobičajen način. Zatim se kvantifikatori izvlače primenom sledećih logičkih zakona:

$$\begin{array}{lcl} \neg(\forall x.F) & \equiv & \exists x.\neg F \\ \neg(\exists x.F) & \equiv & \forall x.\neg F \\ (Qx.F) \wedge G & \equiv & Qx.F \wedge G \\ (Qx.F) \vee G & \equiv & Qx.F \vee G \end{array}$$

pri čemu je $Q \in \{\exists, \forall\}$, a formula G ne sadrži slobodna pojavljivanja promenljive x (ovo ne predstavlja ograničenje opštosti, s obzirom da se vezana promenljiva uvek može prethodno preimenovati u $Qx.F$, zadržavajući logičku ekvivalentnost). □

Primer 2.25. Neka je data formula $\exists x.p(x) \Rightarrow (\forall x.p(x))$. Ova formula se transformiše u prenex normalnu formu tako što se najpre eliminiše implikacija, čime dobijamo formulu $\exists x.\neg p(x) \vee (\forall x.p(x))$. Dalje se kvantifikator $\forall x$ izvlači ispred disjunkcije. Pritom, kako se x pojavljuje kao slobodno u prvom disjunktu $\neg p(x)$, potrebno je najpre preimenovati vezanu promenljivu u $\forall x.p(x)$. Na primer, preimenujmo x u y. Sada imamo da je $\exists x.\neg p(x) \vee (\forall y.p(y)) \equiv \exists x.\forall y.\neg p(x) \vee p(y)$. Dobijena formula je u prenex normalnoj formi.

Primetimo da je bilo neophodno preimenovati promenljivu x prilikom izvlačenja univerzalnog kvantifikatora. Naime, da to nismo uradili, dobili bismo formulu $\exists x.\forall x.\neg p(x) \vee p(x)$, koja nije ekvivalentna polaznoj (sada su oba pojavljivanja promenljive x vezana univerzalnim kvantifikatorom, a u originalnoj formuli je prvo x bilo vezano egzistencijalnim kvantifikatorom). Dakle, bila bi promenjena semantika formule.

Glava 3

Izračunljivost

3.1 Intuitivni pojam algoritma

Definicija 3.1 (Intuitivni pojam algoritma)**.** *Algoritam* predstavlja efektivni računski postupak za rešavanje nekog zadatog problema.

Napomena 3.1. *Problem* koji rešavamo može imati više različitih *instanci* koje imaju istu formu, ali se razlikuju po konkretnim vrednostima iz kojih se sastoje. Na primer, problem može biti *sortiranje niza*, a instanca ovog problema je bilo koji konačni niz brojeva koji treba sortirati. Algoritam treba da bude primenjiv na svaku instancu problema koji se rešava tim algoritmom. *Ulaz* algoritma je neka instanca problema, a *izlaz* je traženo rešenje, čija priroda i forma zavisi od samog problema. Na primer, u problemu sortiranja niza, izlaz je takođe niz – onaj koji se dobija sortiranjem ulaznog niza. U nekim drugim problemima, izlaz može biti jedan broj (npr. kod problema sumiranja elemenata niza), ili samo odgovor *da/ne* (npr. kada se pitamo da li se neka data vrednost nalazi u nizu ili ne).

Napomena 3.2. Iz definicije 3.1 sledi da kada govorimo o algoritmu, mislimo na *računski* postupak, tj. postupak koji se sastoji iz niza računskih operacija koje treba primeniti nad ulaznim podacima (koji su, samim tim, predstavljeni brojevima) da bi se dobio željeni rezultat (koji je opet predstavljen brojevima). Ovo je u skladu sa tradicionalnim shvatanjem algoritma kao postupka za rešavanje računskih problema koji se pre svega javljaju u matematici. U praksi, ovo nas previše ne ograničava, zato što se mnoge vrste informacija mogu *digitalizovati*, tj. predstaviti brojevima (npr. tekst, zvuk, slika, video, i sl.). Naravno, algoritam se može definisati i opštije – kao bilo koji precizno opisani postupak za rešavanje nekog problema (npr. postupak za pripremu nekog jela). Ipak, u ovom tekstu ćemo se pre svega fokusirati na računske postupke, jer su to postupci koji se mogu izvršavati na računarima.

Napomena 3.3. Definicija 3.1 navodi da je algoritam *efektivan* postupak. To znači sledeće:

- algoritam se sastoji iz konačno mnogo koraka. Samim tim, moguće ga je opisati konačnim sredstvima, sačuvati u konačnoj memoriji i sl.
- algoritam je precizno definisan u smislu da su njegovi koraci precizno zadati i njihov redosled je jasno određen. Samim tim, za izvršavanje algoritma

potrebno je samo slepo pratiti njegove korake, tj. nije potrebna nikakva kreativnost. Ovo znači da su algoritmi pogodni za izvršavanje od strane mašine.

- izvršavanje algoritma zahteva konačne resurse (vreme, prostor i sl.). Primetimo da to što se algoritam sastoji iz konačno mnogo koraka ne znači obavezno da će se izvršavati u konačnom vremenu, s obzirom da se isti korak može ponavljati više puta (npr. u petlji). Zbog toga je za korektnost algoritma obično neophodno obezbediti da se za svaki dopustivi ulaz algoritam *zaustavlja* nakon konačno mnogo koraka.
- za svaki dopustivi ulaz – *instancu* datog problema, algoritam daje odgovarajući izlaz. Dakle, algoritam zaista mora da rešava dati problem i da radi ispravno za svaki dopustivi ulaz. Naravno, mogu postojati ulazi koji nisu dopustivi (tj. koji ne predstavljaju legalne instance problema). Za takve ulaze, ponašanje algoritma može biti nepredvidivo (npr. može dati besmisleni rezultat ili se može izvršavati zauvek).

Napomena 3.4. Pojam algoritma mnogo je stariji od savremenih elektronskih računara na koje obično mislimo kada govorimo o algoritmima. Prvi algoritmi potiču još iz drevnih civilizacija, a u njih spadaju:

- algoritmi za izračunavanje aritmetičkih operacija u pozicionim brojevnim sistemima
- Euklidov algoritam za izračunavanje najvećeg zajedničkog delioca
- Gausov algoritam za rešavanje sistema linearnih jednačina
- ...

Slika 3.1: Persijski matematičar al-Horezmi

Sama reč *algoritam* nastala je kao vesternizovana varijanta prezimena persijskog matematičara *al-Horezmija* (*Muḥammad ibn Mūsā al-Khwārizmī*, slika 3.1), koji je u 9. veku nove ere napisao prve knjige o tada poznatim algoritmima, pre svega kada je u pitanju izvođenje računskih operacija u pozicionim brojevnim sistemima (*Knjiga o indijskim izračunavanjima* i *Sabiranje i oduzimanje u indijskoj aritmetici*).

3.2 Formalni pojam algoritma

Napomena 3.5. U ovom tekstu, pod skupom prirodnih brojeva $\mathbb{N}$ podrazumevamo skup $\{0, 1, 2, \ldots\}$. Dakle, nulu smatramo prirodnim brojem i to dalje nećemo posebno naglašavati.

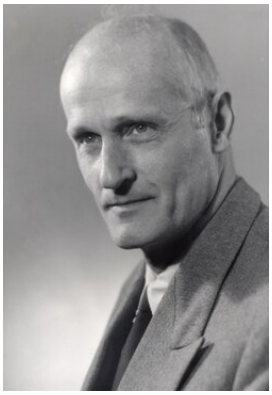
Definicija 3.2. *Parcijalna funkcija arnosti* k ($k \in \mathbb{N}$) je bilo koja funkcija $f : A \to \mathbb{N}$, gde je $A \subseteq \mathbb{N}^k$. Skup A nazivamo *domen* funkcije f i označavamo ga sa $dom(f)$. Pritom, ako je $(x_1, \ldots, x_k) \notin A$, kažemo da je funkcija f *nedefinisana* za $(x_1, \ldots, x_k)$ i pišemo $f(x_1, \ldots, x_k) = -$. Za parcijalnu funkciju arnosti k kažemo da je *totalna* ako je definisana za svako $(x_1, \ldots, x_k) \in \mathbb{N}$, tj. ako je $dom(f) = \mathbb{N}^k$.

Napomena 3.6. Primetimo da arnost k funkcije f može biti i nula. Takva funkcija nema argumente, pa uvek ,,vraća" istu vrednost, npr. $f() = c \in \mathbb{N}$. Takva funkcija se poistovećuje sa samim brojem $c \in \mathbb{N}$.

Da bi se parcijalna funkcija smatrala algoritmom, potrebno je da bude *efektivno izračunljiva*. Postoje razni načini da se definiše pojam *efektivne izračunljivosti*. U narednim odeljcima ilustrujemo neke najpoznatije formalizme kojima se opisuju efektivno izračunljive funkcije. Može se pokazati da su svi oni međusobno ekvivalentni.

3.2.1 μ-rekurzivne funkcije

μ-rekurzivne funkcije predstavljaju formalizam kod koga se parcijalne funkcije koje smatramo izračunljivim konstruišu konačnom primenom unapred datog skupa funkcijskih operatora. Svakim operatorom se od jednostavnijih funkcija grade složenije, kao što u programskim jezicima pomoću raspoloživih programskih konstrukcija od jednostavnijih programa gradimo složenije. μ-rekurzivne funkcije formalizovao je američki matematičar Stiven Klini (slika 3.2).



Slika 3.2: Američki matematičar Stiven Klini (1909-1994)

Definicija 3.3. *Nula-funkcija arnosti* k je funkcija definisana na sledeći način:

$$Z_k(x_1, \ldots, x_k) = 0$$

za svako $(x_1, \ldots, x_k) \in \mathbb{N}^k$. *Funkcija sledbenik* je funkcija arnosti 1, definisana na sledeći način:

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

za svako $x \in \mathbb{N}$. *Funkcija i-te projekcije arnosti k* $(i \leq k)$ je funkcija definisana na sledeći način:

$$P_k^i(x_1, \ldots, x_k) = x_i$$

za svako $(x_1, \ldots, x_k) \in \mathbb{N}^k$. Ove funkcije nazivamo *osnovne rekurzivne funkcije.*

Primer 3.1. Primetimo da je funkcija $P_1^1(x) = x$ zapravo *identička* funkcija. Označavaćemo je i sa $I(x)$.

Napomena 3.7. Intuitivno, osnovne rekurzivne funkcije predstavljaju elementarne algoritamske korake koje možemo da primenimo. Naime, s obzirom da je prirodan broj ili nula ili sledbenik nekog prirodnog broja, funkcije Z_k i S nam omogućavaju da izračunamo bilo koji prirodan broj, kao i da pomoću njih izrazimo uobičajene aritmetičke operacije. Dakle, intuitivno, sve što je izračunljivo se može na kraju svesti na uzastopnu primenu ove dve funkcije. Sa druge strane, funkcija projekcije nam omogućava da među argumentima funkcije izaberemo onaj koji nam treba da nad njim vršimo izračunavanje.

Definicija 3.4. Ako su date parcijalne funkcije $g_1, \ldots, g_m$ arnosti k i parcijalna funkcije h arnosti m, tada je je parcijalna funkcija f arnosti k dobijena *supstitucijom* funkcija g_i u h (u oznaci $Sub(h, g_1, \ldots, g_m)$) definisana na sledeći način:

$$f(x_1, \ldots, x_k) = h(g_1(x_1, \ldots, x_k), \ldots, g_m(x_1, \ldots, x_k))$$

za svako $(x_1, \ldots, x_k)$ za koje je izraz na desnoj strani definisan (u suprotnom, funkcija f je nedefinisana za takvo $(x_1, \ldots, x_k)$).

Napomena 3.8. Intuitivno, operator supstitucije omogućava kompoziciju algoritama. Funkcije $g_1, \ldots, g_m$ možemo da razumemo kao ,,potprograme" kojima smo izračunali neke međurezultate, a funkcija h je glavni program koji se primenjuje na te međurezultate i daje konačan rezultat. Otuda je ceo program zapravo supstitucija $Sub(h, g_1, \ldots, g_m)$. Intuitivno, kompozicijom algoritama dobijamo algoritam.

Primer 3.2. Neka je data proizvoljna parcijalna funkcija f arnosti 1. Posmatrajmo funkciju $g = \underbrace{Sub(f, Sub(f, \ldots, Sub(f}_{n}, I)\ldots)$, tj. funkciju:

$$g(x) = \underbrace{f(f(\ldots f(}_{n} I(x))\ldots)$$

Ovo je zapravo funkcija koja se dobija kada n puta primenimo funkciju f na argument x. Označavaćemo je sa f^n.

Napomena 3.9. U prethodnom primeru, kao i u primerima koji slede, često ćemo imati dve notacije. Jedna notacija predstavlja *konstruktivni oblik* u kome formalno zapisujemo operatore koji se primenjuju prilikom konstrukcije funkcije. Na primer, $g = Sub(h, f, Sub(f, P_2^2, P_2^1))$. Istu funkciju možemo zapisati i *u eksplicitnom obliku*:

$$g(x, y) = h(f(x, y), f(y, x))$$

Konstruktivni oblik je formalniji i precizniji, jer nam govori tačno kako je funkcija konstruisana. Sa druge strane, eksplicitni zapis je obično čitljiviji, jer nam prikazuje šta funkcija izračunava, tj. koje operacije primenjuje nad svojim argumentima. Ipak, kada zapišemo funkciju u eksplicitnom obliku, tada nam nije jasno koji je argument koji po redu (npr. u zapisu $h(f(x,y), f(y,x))$ znamo da imamo dva argumenta, x i y, ali ne znamo koji je argument prvi, a koji drugi).

Primer 3.3. Ako u prethodnom primeru uzmemo da je $f = S$, dobijamo funkciju S^n. Važi da je $S^n(x) = x + n$ za svako $x \in \mathbb{N}$.

Primer 3.4. Posmatrajmo funkciju $C_k^n = Sub(S^n, Z_k)$, tj. funkciju za koju važi:

$$C_k^n(x_1, \ldots, x_k) = S^n(Z_k(x_1, \ldots, x_k))$$

Ova funkcija ima vrednost n za svako $(x_1, \ldots, x_k)$. Zovemo je *konstantnom funkcijom*. Specijalno, za $k = 0$ ova funkcija predstavlja sam broj n.

Primer 3.5. Posmatrajmo funkciju $SP_k^{i,n} = Sub(S^n, P_k^i)$, tj. funkciju:

$$SP_k^{i,n}(x_1, \ldots, x_k) = S^n(P_k^i(x_1, \ldots, x_k))$$

Važi da je $SP_k^{i,n}(x_1, \ldots, x_k) = x_i + n$ za svako $(x_1, \ldots, x_k)$.

Napomena 3.10. Posmatrajmo sledeći pseudokod:

```
{
  x := f_1(x);
  x := f_2(x);
  ...
  x := f_n(x);
}
return  x;
```

Dakle, imamo jednu blok naredbu koja postupno računa rezultat tako što na ulazni podatak x primenjuje različite operacije. Ovaj algoritam se može predstaviti pomoću supstitucije:

$$g(x) = f_n(f_{n-1}(\ldots f_1(x)\ldots)$$

odnosno, u *konstruktivnom obliku*:

$$g = Sub(f_n, Sub(\ldots, Sub(f_2, f_1)\ldots))$$

Iz ovog primera vidimo da blok naredbe koje postoje u većini programskih jezika možemo predstaviti operatorom supstitucije.

Definicija 3.5. Ako su date parcijalne funkcije g arnosti k i h arnosti $k+2$, tada je funkcija f arnosti $k+1$ dobijena *(primitivnom) rekurzijom* od g i h (u oznaci $Rec(g,h)$) definisana na sledeći način:

$$\begin{aligned} f(0, x_1, \ldots, x_k) &= g(x_1, \ldots, x_k) \\ f(y+1, x_1, \ldots, x_k) &= h(y, x_1, \ldots, x_k, f(y, x_1, \ldots, x_k)) \end{aligned}$$

za svako $y, x_1, \ldots, x_k$ za koje su desne strane gornjih jednakosti definisane (u suprotnom, funkcija f je nedefinisana za takve ulaze).

Napomena 3.11. Operator rekurzije omogućava da se vrednost funkcije za neko $y+1$ svede na izračunavanje vrednosti te iste funkcije za vrednost y (pri istim vrednostima ostalih argumenata $x_1, \ldots, x_k$). Pritom, pretpostavljamo da se rekurzija uvek vrši po prvom argumentu. Intuitivno, da bismo neku funkciju implementirali primenom operatora rekurzije, potrebno je da znamo:

1. postupak za izračunavanje vrednosti funkcije za $y = 0$ (izlaz iz rekurzije). Ova vrednost zavisi samo od ostalih argumenata funkcije $x_1, \ldots, x_k$, a postupak za njeno izračunavanje dat je funkcijom g arnosti k

2. postupak za izračunavanje vrednosti funkcije za $y+1$, pod pretpostavkom da nam je poznata vrednost funkcije za y. Ovaj postupak može, pored vrednosti koju vraća rekurzivni poziv, koristiti i vrednosti ostalih argumenata $x_1, \ldots, x_k$, ali i vrednost argumenta y. Otuda je on predstavljen funkcijom h arnosti $k+2$.

Pritom, jasno je da će „dubina" rekurzije biti unapred poznata – imaćemo y rekurzivnih poziva, jer će u svakom od njih vrednost prvog argumenta biti manja za jedan u odnosu na prethodni. Na kraju ćemo imati izlaz iz rekurzije, za $y = 0$.

Postoji i drugačija intuicija vezana za operator rekurzije – možemo ga posmatrati kao način da se formalizuje pojam *petlje*. Naime, operator rekurzije odgovara sledećem pseudokodu:

$z := 0;$
$res := g(x_1, \ldots, x_k);$
$while(z < y)$
$\{$
    $res := h(z, x_1, \ldots, x_k, res);$
    $z := z + 1;$
$\}$
$return \quad res;$

Dakle, funkcija (algoritam) g se koristi inicijalizaciju rezultata res, a zatim se on dalje izračunava u petlji čije je telo predstavljeno funkcijom (algoritmom) h. Brojač petlje z ide od 0 do y. Dakle, u pitanju je brojačka petlja, pri čemu je broj njenih iteracija unapred poznat i određen argumentom y.

Definicija 3.6. Parcijalna funkcija f je *primitivno rekurzivna* ako se može dobiti od osnovnih elementarnih funkcija konačnom primenom operatora supstitucije i primitivne rekurzije.

Primer 3.6. Posmatrajmo funkciju *prev* arnosti 1 definisanu na sledeći način:

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

Ovu funkciju možemo konstruisati pomoću operatora rekurzije – njen konstruktivni zapis je $Rec(Z_0, P_2^1)$. Zaista, kako je funkcija *prev* arnosti 1, potrebna nam je funkcija arnosti 0 (konstanta) koja implementira izlaz iz rekurzije – to je funkcija Z_0 (jer je $Z_0() = 0$). Takođe, potrebna nam je funkcija arnosti 2 koja prihvata samo y i $prev(y)$ (jer funkcija *prev* nema drugih argumenata) i vraća $prev(y+1) = y$. To je upravo funkcija projekcije P_2^1 (jer je $P_2^1(y, prev(y)) = y$). Dakle, funkcija *prev* je primitivno rekurzivna. Zovemo je *funkcija prethodnik*,

jer za svaki prirodni broj y vraća njegovog prethodnika $y-1$ (osim u slučaju nule, kada vraća nulu).

Napomena 3.12. Prethodni primer prikazuje jedan degenerisani slučaj primene operatora rekurzije u kome zapravo i nema rekurzije, s obzirom da nema rekurzivnog poziva. Naime, funkcija h može da ne zavisi od svog poslednjeg argumenta, tj. možemo imati da je:

$$h(y, x_1, \ldots, x_k, z) = \overline{h}(y, x_1, \ldots, x_k)$$

U prethodnom primeru, to smo postigli primenom funkcije projekcije, koja je vraćala samo svoj prvi argument, ignorišući drugi. U opštem slučaju, to znači da je funkcija $f = Rec(g, h)$ zapravo definisana ovako:

$$\begin{array}{lcl} f(0, x_1, \ldots, x_k) & = & g(x_1, \ldots, x_k) \\ f(y+1, x_1, \ldots, x_k) & = & \overline{h}(y, x_1, \ldots, x_k) \end{array}$$

Intuitivno, ovo je ekvivalentno sa sledećim pseudokodom:

$$\begin{array}{l} if(y = 0) \\ \{ \\ \quad return \quad g(x_1, \ldots, x_k); \\ \} \\ else \\ \{ \\ \quad return \quad \overline{h}(y-1, x_1, \ldots, x_k); \\ \} \end{array}$$

Na ovaj način zapravo implementiramo *grananje*.

Napomena 3.13. Primetimo da kada funkcije implementirane operatorom rekurzije zapisujemo u eksplicitnom obliku, neophodno je navesti dva slučaja – jedan za izlaz iz rekurzije, a drugi za rekurzivni korak. Sa druge strane, konstruktivni zapis operatora rekurzije $Rec(f, h)$ je vrlo kompaktan i omogućava jednostavnu dalju konstrukciju složenijih funkcija.

Primer 3.7. Posmatrajmo funkciju $prev^n(x) = x - n$. Ova funkcija je primitivno rekurzivna, kao stepen primitivno rekurzivne funkcije. Primetimo da je $prev^n(x) = 0$ kad god je $n \geq x$.

Primer 3.8. Neka je data proizvoljna parcijalna funkcija f arnosti 1. Posmatrajmo funkciju g arnosti 2 definisanu na sledeći način:

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

Ova funkcija se može dobiti operatorom rekurzije od funkcije f: $g = Rec(I, Sub(f, P_3^3))$, s obzirom da je $I(x) = x$ i $Sub(f, P_3^3)(y, x, g(y, x)) = f(P_3^3(y, x, g(y, x))) = f(g(y, x))$. Ova funkcija zapravo radi tako što funkciju f primenjuje y puta na argument x. Na primer: $g(3, x) = f(g(2, x)) = f(f(g(1, x))) = f(f(f(g(0, x)))) = f(f(f(x)))$. Ovu funkciju ćemo označavati sa $rep(f)$. Ako je f primitivno rekurzivna, onda je i $rep(f)$ primitivno rekurzivna. Primetimo razliku funkcije $rep(f)$ u odnosu na f^n: funkcija f^n primenjuje funkciju f na argument *konstantan broj puta* (n je unapred poznato). Funkcija

$rep(f)$ primenjuje f na drugi argument onoliko puta koliko zahteva prvi argument (koji može biti proizvoljno zadat pri svakom pozivu funkcije). Otuda je za izračunavanje funkcije f^n dovoljno n puta primeniti f (nije potrebna petlja), dok je za izračunavanje funkcije $rep(f)$ potrebna brojačka petlja sa y iteracija.

Primer 3.9. Posmatrajmo funkciju $add = rep(S)$. Vrednost $add(y, x)$ dobijamo kada y puta primenimo S na x. Ovim se zapravo dobija zbir brojeva y i x. Na primer $add(3, 4) = S(S(S(4))) = S(S(5)) = S(6) = 7$.

Primer 3.10. Slično kao u prethodnom primeru, imamo da je $sub = rep(prev))$ funkcija oduzimanja, tj. $sub(y, x) = x - y$. Zaista, na primer, imamo da je $sub(3, 5) = prev(prev(prev(5))) = prev(prev(4)) = prev(3) = 2$. Primetimo da ako je $y \geq x$, tada je $sub(y, x) = 0$.

Primer 3.11. Neka je data proizvoljna parcijalna funkcija f arnosti 2. Posmatrajmo funkciju g arnosti 2 definisanu na sledeći način:

$$\begin{array}{lcl} g(0, x) & = & n \\ g(y + 1, x) & = & f(x, g(y, x)) \end{array}$$

Ova funkcija se može konstruisati operatorom primitivne rekurzije: $g = Rec(C_1^n, Sub(f, P_3^2, P_3^3))$ (jer je $C_1^n(x) = n$, a $Sub(f, P_3^2, P_3^3)(y, x, g(y, x)) = f(P_3^2(y, x, g(y, x)), P_3^3(y, x, g(y, x))) = f(x, g(y, x)))$. Razmotrimo šta ova funkcija radi. Na primer, imamo da je $g(4, x) = f(x, g(3, x)) = f(x, f(x, g(2, x))) = f(x, f(x, f(x, g(1, x)))) = f(x, f(x, f(x, f(x, g(0, x))))) = f(x, f(x, f(x, f(x, C_1^n(x))))) = f(x, f(x, f(x, f(x, n))))$. Dakle, polazeći od vrednosti n, ova funkcija akumulira rezultat primene funkcije f, y puta sa prvim argumentom x. Označavaćemo je sa $acc(n, f)$. Ako je f primitivno rekurzivna, onda je takva i $acc(n, f)$.

Primer 3.12. Sada imamo da je $mul = acc(0, add)$ funkcija množenja, tj. $mul(y, x) = y \cdot x$. Na primer:

$$\begin{array}{lcl} mul(4, 5) & = & \\ add(5, add(5, add(5, add(5, 0)))) & = & \\ add(5, add(5, add(5, 5))) & = & \\ add(5, add(5, 10)) & = & \\ add(5, 15) & = & 20 \end{array}$$

Primer 3.13. Funkcija $pow = acc(1, mul)$ izračunava stepen, tj. $pow(y, x) = x^y$. Na primer:

$$\begin{array}{lcl} pow(3, 5) & = & \\ mul(5, mul(5, mul(5, 1))) & = & \\ mul(5, mul(5, 5)) & = & \\ mul(5, 25) & = & 125 \end{array}$$

Primer 3.14. Posmatrajmo funkciju:

$$\begin{array}{lcl} fact(0) & = & 1 \\ fact(y + 1) & = & mul(fact(y), S(y)) \end{array}$$

tj. funkciju $fact = Rec(C_0^1, Sub(mul, P_2^2, Sub(S, P_2^1)))$. Ova funkcija računa faktorijel datog broja, tj. $fact(x) = x!$. U pitanju je standardna rekurzivna definicija faktorijela. Jasno je da je ova funkcija primitivno rekurzivna.

Primer 3.15. Posmatrajmo funkciju:

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

Dakle, ova funkcija vraća 1 ako je argument jednak nuli, a u suprotnom vraća 0. Ova funkcija je primitivno rekurzivna ($is\_zero = Rec(C_0^1, Z_2)$). Može se koristiti za testiranje da li je neka vrednost jednaka nuli.

Primer 3.16. Pomoću funkcija iz prethodnih primera možemo definisati logičke operatore. U tu svrhu, smatraćemo da je vrednost 0 logički *netačno*, dok je svaka vrednost različita od nule logički *tačno*. Sada imamo:

- $not(x) = is\_zero(x)$
- $and(x, y) = mul(x, y)$
- $or(x, y) = add(x, y)$
- $xor(x, y) = or(and(not(x), y), and(x, not(y)))$

Primer 3.17. Posmatrajmo funkciju:

$$cond(y, x_1, x_2) = \begin{cases} x_2, & \text{za } y = 0 \\ x_1, & \text{za } y \neq 0 \end{cases}$$

Ova funkcija, ispituje logičku vrednost svog prvog argumenta – ako je ona logički tačna, tada vraća svoj drugi argument, a u suprotnom vraća treći argument. Intuitivno, ova funkcija omogućava *grananje* u našem programu. Ova funkcija se može predstaviti kao primitivno rekurzivna funkcija:

$$\begin{aligned} cond(0, x_1, x_2) &= x_2 \\ cond(y+1, x_1, x_2) &= x_1 \end{aligned}$$

tj. važi: $cond = Rec(P_2^2, P_4^2)$.

Primer 3.18. Posmatrajmo funkciju:

$$ge(x, y) = is\_zero(sub(x, y))$$

Ova funkcija je primitivno rekurzivna, jer je dobijena supstitucijom od primitivno rekurzivnih funkcija ($ge = Sub(is\_zero, sub)$). Primetimo da je $y - x = 0$ akko je $x \geq y$. Otuda ova funkcija vraća 1 ako je $x \geq y$, dok u suprotnom vraća 0. Čitaocu prepuštamo za vežbu da, koristeći ovu funkciju i logičke funkcije iz primera 3.16 implementira i ostale relacione operatore (eq, le, gt, lt).

Primer 3.19. Funkcije:

- $max(x, y) = cond(ge(x, y), x, y)$
- $min(x, y) = cond(le(x, y), x, y)$

su primitivno rekurzivne, s obzirom da su dobijene od primitivno rekurzivnih funkcija primenom supstitucije. Ove funkcije računaju, respektivno, maksimum i minimum dva broja.

Primer 3.20. Posmatrajmo funkciju:

$$mod(x,y) = \begin{cases} 0, & \text{za } x = 0 \\ 0, & \text{ako je } mod(x-1,y)+1 = y \\ mod(x-1,y)+1, & \text{inače} \end{cases}$$

Za $y \neq 0$ ova funkcija vraća ostatak pri deljenju x sa y (u suprotnom vraća x). Kako bismo je predstavili pomoću primitivne rekurzije, predstavimo je ovako:

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

odnosno, važi da je:

$$mod = Rec(Z_1, Sub(cond, Sub(eq, Sub(S, P_3^3), P_3^2), Z_3, Sub(S, P_3^3)))$$

Slično:

$$div(x,y) = \begin{cases} 0, & \text{za } x = 0 \\ div(x-1,y)+1, & \text{ako je } mod(x,y) = 0 \\ div(x-1,y), & \text{inače} \end{cases}$$

Ova funkcija za $y \neq 0$ izračunava celobrojni količnik x pri deljenju sa y, a u suprotnom vraća nulu. Da bismo je izrazili pomoću primitivne rekurzije, predstavimo je u sledećem obliku:

$$\begin{aligned} div(0,y) &= 0 \\ div(x+1,y) &= cond(eq(mod(x+1,y),0), div(x,y)+1, div(x,y)) \end{aligned}$$

odnosno:

$$div = Rec(Z_1, Sub(cond, Sub(eq, Sub(mod, Sub(S, P_3^1), P_3^2)), Sub(S, P_3^3), P_3^3))$$

Teorema 3.1. *Svaka primitivno rekurzivna funkcija je totalna.*

Dokaz. Osnovne rekurzivne funkcije su totalne. Operatorima supstitucije i rekurzije se od totalnih funkcija dobijaju totalne funkcije. Otuda su sve primitivno rekurzivne funkcije totalne. □

Primitivno rekurzivne funkcije omogućavaju nam da izražavamo aritmetičke operacije, linijske i razgranate strukture programa, kao i brojačke petlje, tj. petlje kod kojih je broj iteracija poznat unapred, u trenutku ulaska u petlju. Sa druge strane, petlje oblika „*ponavljaj naredbu dokle god se neki uslov ne ispuni*", pri čemu ne znamo ni kada će ni da li će se uslov izlaska iz petlje ispuniti nije moguće predstaviti primitivno rekurzivnim funkcijama. Za tako nešto nam je potreban *operator minimizacije*.

Definicija 3.7. Ako je data parcijalna funkcija g arnosti $k+1$, tada je funkcija f arnosti k dobijena *minimizacijom* od funkcije g (u oznaci $\mu(g)$) definisana na sledeći način:

$$f(x_1, \ldots, x_k) = y$$

gde je $y \in \mathbb{N}$ takvo da važi:

- za svako $z < y$, vrednost $g(z, x_1, \ldots, x_k)$ je definisana i važi:

$$g(z, x_1, \ldots, x_k) > 0$$

- vrednost $g(y, x_1, \ldots, x_k)$ je definisana i važi:

$$g(y, x_1, \ldots, x_k) = 0$$

Ako takvo y ne postoji, funkcija f je nedefinisana za takvo $x_1, \ldots, x_k$.

Napomena 3.14. Oznaka $\mu(g)$ se koristi kada je g data u konstruktivnom obliku. Ako je g data u eksplicitnom obliku (kao funkcijski izraz $g(y, x_1, \ldots, x_k)$), tada ćemo koristiti sledeću *eksplicitnu* notaciju:

$$\mu y[g(y, x_1, \ldots, x_k) = 0]$$

Intuitivno, ovo znači: „*najmanje y takvo da je* $g(y, x_1, \ldots, x_k)$ *jednako* 0". Primetimo da je ovde neophodno eksplicitno navesti argument po kome se vrši minimizacija (μy). Naime, minimizacija se, prema definiciji, uvek vrši po prvom argumentu. Kada koristimo konstruktivnu notaciju, tada je jasno koji je argument prvi. Sa druge strane, kao što je ranije rečeno, u eksplicitnoj notaciji nije uvek jasno koji je argument koji po redu, pa je zato potrebno eksplicitno naglasiti po kom se argumentu vrši minimizacija.

Napomena 3.15. Operator minimizacije je intuitivno ekvivalentan sledećem pseudokodu:

$y := 0;$
$while(g(y, x_1, \ldots, x_k))$
$\{$
    $y := y + 1;$
$\}$
$return \quad y;$

Dakle, funkciju $g(y, x_1, \ldots, x_k)$ posmatramo kao logički izraz (uslov petlje), a rezultat je najmanje y za koje uslov taj uslov postaje netačan.

Pretpostavimo sada da želimo da u petlji izračunamo nešto drugo. Na primer:

$res := h(x_1, \ldots, x_k);$
$while(g(res, x_1, \ldots, x_k))$
$\{$
    $res := f(x_1, \ldots, x_k, res);$
$\}$
$return \quad res;$

Ovo je sada neka najopštija *while* petlja u kojoj izračunavamo neki rezultat res. Broj iteracija ove petlje nije unapred poznat, jer zavisi od uslova $g(res, x_1, \ldots, x_k)$. Ipak, razmotrimo šta bi se desilo ako bismo broj iteracija ove petlje ograničili unapred na neko y. Tada bi ova petlja bila ekvivalentna

sledećem algoritmu:

$$
\begin{array}{l}
z := 0; \\
res := h(x_1, \ldots, x_k); \\
while(z < y) \\
\{ \\
\quad res := f(x_1, \ldots, x_k, res); \\
\quad z := z + 1; \\
\} \\
return \quad res;
\end{array}
$$

Drugim rečima, vrednost rezultata res nakon y iteracija se može izračunati primitivnom rekurzijom, tj. $res = res(y, x_1, \ldots, x_k)$, gde je:

$$
\begin{array}{lcl}
res(0, x_1, \ldots, x_k) & = & h(x_1, \ldots, x_k) \\
res(y+1, x_1, \ldots, x_k) & = & f(x_1, \ldots, x_k, res(y, x_1, \ldots, x_k));
\end{array}
$$

tj. $res = Rec(h, f)$. Kako u polaznom algoritmu broj iteracija y ne znamo unapred, potrebno je najpre odrediti broj iteracija y minimizacijom, a zatim za tako dobijeno y izračunati rezultat primitivnom rekurzijom. Dakle, imamo funkciju:

$$res(\mu y[g(res(y, x_1, \ldots, x_k), x_1, \ldots, x_k) = 0], x_1, \ldots, x_k)$$

odnosno, u konstruktivnoj notaciji:

$$Sub(res, \mu(Sub(g, res, P^2_{k+1}, \ldots, P^{k+1}_{k+1})), P^1_k, \ldots, P^k_k)$$

Definicija 3.8. Parcijalna funkcija f je *μ-rekurzivna* ako se može dobiti od osnovnih rekurzivnih funkcija konačnom primenom operatora supstitucije, rekurzije i minimizacije.

Napomena 3.16. Pojam μ-rekurzivne funkcije je opštiji od pojma *primitivno rekurzivne* funkcije, zato što pored supstitucije i rekurzije dozvoljavamo i operator minimizacije. Sada pored brojačkih petlji imamo na raspolaganju i petlje sa proizvoljnim uslovom izlaska.

Često ćemo μ-rekurzivne funkcije kraće nazivati samo *rekurzivne funkcije*.

Primer 3.21. Setimo se funkcije div iz primera 3.20. Ovu funkciju možemo predstaviti i na sledeći način:

$$div(x, y) = \mu q[le(mul(S(q), y), x) = 0]$$

odnosno, u konstruktivnom obliku:

$$div = \mu(Sub(le, Sub(mul, Sub(S, P^1_3), P^3_3), P^2_3))$$

Drugim rečima, vrednost $div(x, y)$ je najmanje q takvo da je vrednost izraza $le(mul(S(q), y), x)$ jednaka nuli, tj. najmanje q da je uslov $(q+1) \cdot y \leq x$ logički netačan.

Napomena 3.17. Ponekad ćemo koristiti i oznaku:

$$\mu y[g(y, x_1, \ldots, x_k) = n]$$

što znači „*najmanje y takvo da je* $g(y, x_1, \ldots, x_k)$ *jednako* n". Ovo proširenje notacije nije suštinsko, jer ovo uvek možemo ekvivalentno zapisati ako:

$$\mu y[not(eq(g(y, x_1, \ldots, x_k), n)) = 0]$$

Na sličan način, možemo koristiti i druge relacione operatore u uslovu, npr.:

$$\mu y[g(y, x_1, \ldots, x_k) \leq n]$$

uz značenje „*Najmanje y takvo da je* $g(y, x_1, \ldots, x_k)$ *manje ili jednako* n". Ovo je ekvivalentno sa:

$$\mu y[not(le(g(y, x_1, \ldots, x_k), n)) = 0]$$

Tako smo u prethodnom primeru, umesto:

$$div(x, y) = \mu q[le(mul(S(q), y), x) = 0]$$

mogli smo da zapišemo:

$$div(x, y) = \mu q[(q + 1) \cdot y > x]$$

što je znatno čitljivije i intuitivnije („*najmanje q takvo da je* $(q + 1) \cdot y$ *veće od* x").

Napomena 3.18. Iz primera 3.20 i 3.21 vidimo da se ista parcijalna funkcija može predstaviti kao μ-rekurzivna funkcija na različite načine. Ovo je analogno činjenici da se isti rezultat može dobiti različitim računarskim programima. Za dve μ-rekurzivne reprezentacije kažemo da su *ekvivalentne*, ako izračunavaju istu parcijalnu funkciju.

Napomena 3.19. Sve funkcije koje smo prikazali u prethodnim primerima su bile primitivno rekurzivne. Slučaj funkcije *div* je zanimljiv jer smo nju predstavili na dva načina: prvi nije koristio minimizaciju, a drugi jeste. Ipak, funkcija *div* jeste primitivno rekurzivna, jer se *može* predstaviti bez korišćenja minimizacije. Dakle, dovoljno je da postoji jedna reprezentacija koja ne koristi minimizaciju, a koja realizuje neku funkciju, ta funkcija će biti primitivno rekurzivna. Ipak, ponekad operator minimizacije omogućava jednostavniju i intuitivniju konstrukciju.

Primer 3.22. Posmatrajmo funkciju:

$$isqrt(x) = max\{y \mid y^2 \leq x\}$$

Ova funkcija vraća ceo deo kvadratnog korena broja x. Na primer $isqrt(36) = 6$, a $isqrt(50) = 7$. Ovu funkciju možemo predstaviti minimizacijom na sledeći način:

$$\mu y[(y + 1)^2 > x]$$

Dakle, tražimo najmanje y takvo da je $(y + 1)^2$ veće od x. Napomenimo da ova funkcija *jeste* primitivno rekurzivna, ali je njena primitivno rekurzivna formulacija nešto komplikovanija. Sa druge strane, prikazana realizacija pomoću operatora minimizacije je krajnje jednostavna i intuitivna.

Napomena 3.20. Primetimo da ako koristimo operator minimizacije, tada μ-rekurzivna funkcija više ne mora biti totalna. Naime, čak i da je funkcija $g(y, x_1, \ldots, x_n)$ totalna, funkcija $\mu(g)$ to ne mora biti, jer se može dogoditi da za neko $x_1, \ldots, x_n$ funkcija $g(y, x_1, \ldots, x_n)$ nije jednaka nuli ni za jedno y.

Primer 3.23. Neka je data funkcija:

$$f(x,y) \quad = \quad \begin{cases} z, & \text{gde je } y \cdot z = x, \text{ ako takvo } z \text{ postoji} \\ -, & \text{inače} \end{cases}$$

Ova funkcija je μ-rekurzivna, jer se može predstaviti na sledeći način:

$$f(x,y) = \mu z[y \cdot z = x]$$

Ipak, ova funkcija nije primitivno rekurzivna, s obzirom da nije totalna.

Primer 3.24. Posmatrajmo funkciju:

$$undef(x) = -$$

za svako x. Ova funkcija je izračunljiva, jer se može prikazati npr. kao $\mu y[x+y+1=0]$. Ova funkcija je nedefinisana za svako x. Intuitivno, odgovara programu koji sadrži beskonačnu petlju po y.

Iz prethodnog izlaganja bi se moglo pogrešno zaključiti da je μ-rekurzivna funkcija totalna ako i samo ako je primitivno rekurzivna. Ipak, ovde implikacija važi samo u jednom smeru. Naime, važi teorema.

Teorema 3.2. *Postoji totalna μ-rekurzivna funkcija koja nije primitivno rekurzivna.*

Umesto dokaza ove teoreme, navodimo jedan zanimljiv primer.

Primer 3.25. Čuvena *Akermanova funkcija*:

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

je totalna, a može se pokazati da je μ-rekurzivna, ali nije primitivno rekurzivna. Iako data definicija funkcije sadrži rekurziju, to nije primitivna rekurzija, jer se rekurzija vrši po oba argumenta, a ne samo po prvom. Naravno, postoje i drugi načini da se ova funkcija definiše, ali se može pokazati da ni jedan od njih neće biti u obliku primitivne rekurzije. Naime, važi da je:

$$A(m+1,n) = A(m,A(m+1,n-1)) = A(m,A(m,A(m+1,n-2) = \ldots = \underbrace{A(m,A(m,\ldots,A(m,}_{n} A(m+1,0)\ldots) = \underbrace{A(m,A(m,\ldots,A(m,}_{n+1} 1)\ldots)$$

Intuitivno, potrebna nam je brojačka petja sa $n+1$ iteracija koje u sebi pozivaju funkciju A sa prvim argumentom m. Međutim, po istom principu, za izračunavanje $A(m,k)$ (za neko k) potrebna nam je petlja sa $k+1$ iteracijom koja poziva funkciju A sa prvim argumentom $m-1$. Dakle, imaćemo petlju u petlji. Dalje ćemo za izračunavanje funkcije $A(m-1,k')$ (za neko k') opet imati petlju sa $k'+1$ iteracijom koja poziva funkciju A sa prvim argumentom $m-2$, pa ćemo imati trostruku petlju i td. Ukupan broj ugnježdenih brojačkih petlji biće jednak $m+1$. Za fiksirano m, funkcija:

$$f(n) = A(m+1,n)$$

bi bila primitivno rekurzivna, jer bismo imali fiksirani broj ugnježdenih brojačkih petlji, pa bismo kompozicijom $m+1$ primitivno rekurzivne funkcije dobili primitivno rekurzivnu funkciju. Međutim, u Akermanovoj funkciji, prvi argument m može biti proizvoljno veliki. Otuda je nije moguće konstruisati konačnom primenom operatora primitivne rekurzije i supstitucije.

Sa druge strane, Akermanova funkcija se može dobiti upotrebom operatora minimizacije. Naime, može se dokazati da je funkcija $R(m,n,k)$ definisana sa:

$$R(m,n,k) \quad = \quad \begin{cases} 1, & \text{ako je } A(m,n)=k \\ 0, & \text{inače} \end{cases}$$

primitivno rekurzivna (dokaz je tehnički komplikovan, pa ga izostavljamo). Sada važi:

$$A(m,n) = \mu k[R(m,n,k)=1]$$

Najzad, nisu sve parcijalne funkcije μ-rekurzivne, o čemu govori sledeća teorema.

Teorema 3.3. *Postoji totalna funkcija koja nije μ-rekurzivna.*

Dokaz ove teoreme ostavićemo za kasnije.

3.2.2 URM programi

Pored rekurzivnih funkcija koje predstavljaju pogodnu matematičku notaciju za opisivanje algoritama, postoje i formalizmi koji su bliži realnim računarima i u većoj meri simuliraju proceduralni način razmišljanja uobičajen u programiranju. Jedan takav formalizam predstavljaju i *URM programi* koji predstavljaju apstraktnu varijantu jednog krajnje jednostavnog asemblerskog jezika.

Definicija 3.9 (URM program)**.** Pretpostavimo da nam je na raspolaganju beskonačni niz registara $R_1, R_2, \ldots$, pri čemu svaki registar može čuvati jedan prirodan broj. *URM program* (engl. *Unlimited Register Machine* (URM)) se sastoji iz konačnog niza *instrukcija* $I_1, I_2, \ldots, I_r$, pri čemu je svaka od instrukcija nešto od sledećeg:

- $Z(n)$: ova instrukcija postavlja registar R_n na nulu ($R_n \leftarrow 0$)
- $S(n)$: ova instrukcija uvećava vrednost registra R_n za jedan ($R_n \leftarrow R_n+1$)
- $T(m,n)$: ova instrukcija kopira vrednost registra R_m u registar R_n ($R_n \leftarrow R_m$)
- $J(m,n,k)$: ako su vrednosti registara R_m i R_n jednake, tada se vrši *skok* na instrukciju I_k. U suprotnom, instrukcija ne radi ništa.

Program se izvršava počev od instrukcije I_1, pri čemu se instrukcije izvršavaju redom, osim u slučaju kada dođe do *skoka* u instrukciji J-tipa, kada se izvršavanje nastavlja od navedene instrukcije. Program se zaustavlja kada dođe do nepostojeće instrukcije (tj. instrukcije sa indeksom većim od r).

Definicija 3.10. Parcijalna funkcija f arnosti k koju *izračunava* URM program P definisana je na sledeći način:

- za svaku k-torku $(x_1, \ldots, x_k) \in \mathbb{N}^k$, registri $R_1, \ldots, R_k$ se inicijalizuju vrednostima $x_1, \ldots, x_k$ redom, dok se ostali registri inicijalizuju nulom
- pokrenemo program počev od prve instrukcije
- ako se program zaustavi nakon konačno mnogo koraka, tada je vrednost funkcije $f(x_1, \ldots, x_k)$ jednaka vrednosti registra R_1 na kraju izvršavanja programa
- u suprotnom, funkcija nije definisana za datu k-torku

Napomena 3.21. Jasno je da isti URM program P definiše po jednu parcijalnu funkciju za svako $k \in \mathbb{N}$.

Definicija 3.11 (URM izračunljiva funkcija)**.** Za parcijalnu funkciju f arnosti k kažemo da je *URM izračunljiva* ako postoji URM program koji je izračunava.

Primer 3.26. Jedan mogući program koji izračunava funkciju $add(x, y) = x + y$ bi bio sledeći:

$$\begin{array}{ll} 1: & J(3,2,5) \\ 2: & S(1) \\ 3: & S(3) \\ 4: & J(1,1,1) \end{array}$$

Na početku je x u R_1, a y u R_2. Registar R_3 je inicijalno jednak nuli – on će biti brojač u petlji koja će se izvršavati y puta. Zato na početku petlje proveravamo da li je brojač R_3 jednak R_2 (tj. broju y). Ako jeste, skačemo na nepostojeću instrukciju (instrukcija broj 5), čime se program završava. Ako nije, tada izvršavamo instrukciju broj 2 kojom se vrednost registra R_1 uvećava za 1 (tj. dodajemo jedan na x). Zatim uvećavamo brojač (instrukcijom 3) i skačemo bezuslovno na početak petlje (instrukcija broj 1). Bezuslovni skok se postiže tako što upoređujemo registar R_1 sa samim sobom – time je jednakost uvek ispunjena, pa se skok uvek vrši. Kada se algoritam završi, u registru R_1 ostaje broj x uvećan y puta za 1 (tj. vrednost $x + y$).

Primer 3.27. Program u nastavku izračunava funkciju:

$$sub'(x, y) = \begin{cases} x - y, & \text{za } x \geq y \\ -, & \text{za } x < y \end{cases}$$

$$\begin{array}{ll} 1: & J(1,2,5) \\ 2: & S(2) \\ 3: & S(3) \\ 4: & J(1,1,1) \\ 5: & T(3,1) \end{array}$$

Registri R_1 i R_2 sadrže redom x i y. U registru R_3 biće izračunata razlika. Instrukcije $1 - 4$ predstavljaju petlju u kojoj se uvećava R_2 dok ne postane jednako R_1. Istovremeno, uvećava se i razlika u R_3. Ako je na početku bilo $x \geq y$, tada će se nakon $x - y$ iteracija vrednost R_2 izjednačiti sa R_1 i iz petlje će se izaći, a u R_3 će se nakon izlaska iz petlje nalaziti razlika $x - y$. Ovu razliku prebacujemo u R_1, čime se rad programa završava. Ako je na početku bilo $x < y$, tada R_2 nikada neće uvećanjem postati jednako R_1 (jer je u startu bilo veće). Zato će se petlja izvršavati zauvek, pa funkcija neće biti definisana.

Primer 3.28. Neka je data funkcija:

$$sub(x,y) \quad = \quad \begin{cases} x-y, & \text{za } x \geq y \\ 0, & \text{za } x < y \end{cases}$$

Ovu funkciju možemo izračunati sledećim URM programom:

$$\begin{array}{ll} 1: & T(1,3) \\ 2: & T(2,4) \\ 3: & J(1,4,11) \\ 4: & J(2,3,9) \\ 5: & S(3) \\ 6: & S(4) \\ 7: & S(5) \\ 8: & J(1,1,3) \\ 9: & Z(1) \\ 10: & J(1,1,12) \\ 11: & T(5,1) \end{array}$$

Ideja programa je da se u registrima R_3 i R_4 nakon k iteracija petlje nalaze, redom, vrednosti $x+k$ i $y+k$. Sama vrednost k nalaziće se u registru R_5. Petlja je implementirana instrukcijama $3-8$. Ako je R_1 jednako sa R_4, znači da smo pronašli k takvo da je $x = y + k$. Tada je k upravo razlika $x - y$, pa skačemo na instrukciju 11 kojom se vrednost k iz R_5 kopira u R_1, čime se program završava. U suprotnom, ako je R_2 jednako R_3, to znači da smo pronašli k takvo da je $x + k$ jednako y. Tada je $y > x$, pa skačemo na instrukciju 9 kojom se registar R_1 postavlja na nulu. Instrukcija 10 predstavlja beuslovni skok kojim se preskače instrukcija 11. U telu petlje uvećavamo registre R_3, R_4 i R_5, nakon čega bezuslovno skačemo na početak petlje (instrukcija 8).

Napomena 3.22. Ponekad će biti zgodno da za implementaciju nekog složenijeg algoritma iskoristimo program P koji smo već imali ranije za realizaciju neke jednostavnije funkcije. U tom slučaju, biće nam potrebno da u neki program P' ugradimo program P kao njegov *potprogram*. Pritom, program P u okviru programa P' možda neće početi od prve instrukcije, već od instrukcije sa rednim brojem $k+1$, za neko k. U tom slučaju, da bi program P i dalje radio korektno, biće potrebno da sve reference na instrukcije (u instrukcijama J tipa) u njemu uvećamo za k. Ovo ćemo nazivati *translacijom programa P za k instrukcija* i označavaćemo sa $P_{\to k}$.

Primer 3.29. Pretpostavimo da je data funkcija:

$$lt(x,y) \quad = \quad \begin{cases} 1, & \text{za } x < y \\ 0, & \text{za } x \geq y \end{cases}$$

Jasno je da je $x \geq y$ akko je $sub(y,x) = 0$. Otuda možemo iskoristiti program iz prethodnog primera. Označimo taj program sa P. Posmatrajmo program:

$$
\begin{array}{ll}
1: & T(1,3) \\
2: & T(2,1) \\
3: & T(3,2) \\
4-14: & P_{\to 3} \\
15: & Z(2) \\
16: & J(1,2,19) \\
17: & Z(1) \\
18: & S(1)
\end{array}
$$

Dakle, kako bismo pozvali potprogram P za argumente y i x, najpre je potrebno zameniti redosled argumenata u registrima R_1 i R_2. Ovo se radi pomoću prve tri instrukcije koje implementiraju klasični *algoritam zamene vrednosti promenljivih*. Potprogram P smeštamo počev od instrukcije 4, pa je potrebno sve adrese u njemu translirati za 3 (jer je instrukcija koja je bila prva sada četvrta, druga je peta i td.). Nakon što se potprogram završi, u registru R_1 se nalazi vrednost koja je jednaka nuli akko je $x \geq y$. Samim tim, ako je R_1 jednak nuli, treba ga takvog i ostaviti. Ovo postižemo tako što registar R_2 postavimo na nulu, a zatim ga poredimo sa R_1. Ako je uslov ispunjen, skačemo na nepostojeću instrukciju 19, čime se program završava. Sa druge strane, ako R_1 nije jednak nuli, tada ga je potrebno postaviti na 1. Ovo radimo tako što ga najpre postavimo na nulu, a zatim ga uvećamo za jedan.

Primer 3.30. Funkcija $mul(x, y) = x \cdot y$ se može implementirati tako što iskoristimo program iz primera 3.26 koji izračunava zbir dva broja. Označimo taj program sa P. Sada imamo program:

$$
\begin{array}{ll}
1: & T(1,4) \\
2: & T(2,5) \\
3: & J(5,6,13) \\
4: & T(7,1) \\
5: & T(4,2) \\
6-9: & P_{\to 5} \\
10: & T(1,7) \\
11: & S(6) \\
12: & J(1,1,3) \\
13: & T(7,1)
\end{array}
$$

Argumenti programa x i y (koji se inicijalno nalaze u R_1 i R_2) će se čuvati u R_4 i R_5. Registar R_6 će biti brojač u petlji, dok će R_7 akumulirati rezultat (inicijalno je 0). Na početku kopiramo x i y u R_4 i R_5 redom, a zatim ulazimo u petlju (instrukcije $3-12$) u kojoj proveravamo da li je brojač R_6 jednak y. Ukoliko jeste izlazimo iz petlje i skačemo na instrukciju 13, dok u suprotnom ulazimo u petlju. U svakoj iteraciji petlje, potrebno je na tekuću sumu (registar R_7) dodati vrednost x iz registra R_4. Zato ćemo R_7 i R_4 kopirati u R_1 i R_2, kao argumente potprograma. Potprogram P je transliran za 5 instrukcija i zauzima instrukcije $6-9$. Nakon njegovog završetka registar R_1 sadrži traženi zbir koji treba prebaciti u R_7. Nakon toga uvećavamo brojač u R_6, a zatim bezuslovno skačemo na početak petlje. Petlja će se završiti nakon što se izvrši y puta, tj. nakon što y puta dodamo x. Na kraju je samo potrebno izračunatu sumu iz R_7 kopirati u R_1.

Teorema 3.4. *Ako je parcijalna funkcija f μ-rekurzivna, tada je ona i URM izračunljiva.*

Dokaz. Da bismo dokazali da je svaka μ-rekurzivna funkcija URM izračunljiva, potrebno je osnovne rekurzivne funkcije, kao i svaki od konstruktivnih koraka (*Sub*, *Rec* i μ) simulirati URM programom. Ovo nije teško, imajući u vidu intuitivna značenja ovih operatora koja su data u prethodnom odeljku. Ostavljamo čitaocu za vežbu da se u to i uveri. □

Važi i obrnuto, o čemu govori i sledeća teorema.

Teorema 3.5. *Ako je parcijalna funkcija f URM izračunljiva, tada je ona i μ-rekurzivna.*

Napomena 3.23. Dokaz ove teoreme je tehnički znatno zahtevniji, jer je potrebno proizvoljan URM program predstaviti kao μ-rekurzivnu funkciju. Ipak, ovo je moguće uraditi, na šta ćemo se vratiti kasnije.

Iz prethodne dve teoreme sledi da se klase μ-rekurzivnih i URM izračunljivih funkcija poklapaju.

3.2.3 Čerčova teza

U prethodna dva odeljka prikazali smo dva često korišćena formalizma za opis algoritama, tj. efektivno izračunljivih funkcija. Pored njih, postoje i mnogi drugi načini formalnog zasnivanja izračunljivosti, od kojih ćemo neke prikazati kasnije, u drugim kontekstima. Može se pokazati da su svi oni ekvivalentni, tj. da definišu istu klasu parcijalnih funkcija. Ipak, postavlja se pitanje da li ta klasa zaista odgovara intuitivnom pojmu algoritma o kome smo govorili u poglavlju 3.1. O tome govori sledeća:

Čerčova teza Klasa formalno izračunljivih funkcija poklapa se sa intuitivnim pojmom algoritma.

Ova teza povezuje formalni matematički pojam sa neformalnim, intuitivnim pojmom. Otuda ona nije teorema u klasičnom smislu reči i ne može se dokazivati. Ipak, Čerčova teza je nešto što je opšte prihvaćeno u računarstvu i u šta se u velikoj meri veruje.

3.3 Kodiranje

U prethodnom poglavlju smo algoritme razmatrali kao izračunljive parcijalne funkcije nad $\mathbb{N}^k$ ($k \in \mathbb{N}$). To znači da smo podrazumevali da su njihovi argumenti prirodni brojevi. Sa druge strane, svesni smo da su ulazi naših algoritama u praksi često druge vrste podataka: tekstualni podaci, nizovi, grafovi, matrice, i sl. Samim tim, na prvi pogled deluje da takvi algoritmi nisu pokriveni našim formalnim pojmom izračunljivih funkcija. Na sreću, taj prvi utisak je pogrešan – ispostavlja se da se svi takvi ulazni podaci mogu predstaviti prirodnim brojevima. Takva brojevna reprezentacija podatka naziva se njegovim *kôdom*. Postupak prevođenja podatka u brojeve nazivamo *kodiranje*, dok se obrnuti postupak dobijanja originalnog podatka iz njegovog kôda naziva *dekodiranje*.

Prilikom kodiranja, najpre je potrebno podatak predstaviti nizom ili uređenom torkom prirodnih brojeva. Ovo obično ne predstavlja problem:

- ako je ulazni podatak ceo broj x, on se može kodirati npr. pomoću zapisa *znaka* i *apsolutne vrednosti*, tj. kao uređeni par prirodnih brojeva $(s, |x|)$ (znak $s = 0$ označava pozitivan, a $s = 1$ negativan broj). Alternativno, možemo koristiti zapis u *potpunom komplementu*, gde ćemo ponovo imati uređeni par prirodnih brojeva (n, u): prvi broj n će predstavljati broj pozicija u binarnom zapisu, a za drugi broj u će važiti:

$$u = \begin{cases} x, & \text{ako je } x \geq 0 \\ 2^n - |x|, & \text{ako je } x < 0 \end{cases}$$

- ako je ulazni podatak realni broj[1], on se obično predstavlja zapisom u *pokretnom zarezu*. Ovaj zapis se može kodirati trojkom (s, m, e) prirodnih brojeva koji predstavljaju redom *znak*, *mantisu* i *eksponent*

- ako je ulazni podatak niz prirodnih brojeva $x_1, \ldots, x_n$, onda ga možemo kodirati tako što prosto navedemo njegove elemente:

$$x_1, \ldots, x_n$$

  Niz celih ili realnih brojeva se može kodirati na sličan način, s tim što se umesto brojeva $x_1, \ldots, x_n$ navode uređeni parovi (odnosno trojke) prirodnih brojeva kojima se ti brojevi kodiraju.

- ako je ulazni podatak konačan skup brojeva, tada ga možemo predstaviti rastućim nizom svojih elemenata i kodirati kao što je malopre opisano

- ako je ulazni podatak matrica dimenzija $m \times n$:

$$\begin{matrix} x_{11} & x_{12} & \ldots & x_{1n} \\ x_{21} & x_{22} & \ldots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \ldots & x_{mn} \end{matrix}$$

  tada se ona može predstaviti npr. tako što se najpre navedu njene dimenzije, a zatim se njene vrste poređaju u niz:

$$m, n, x_{11}, \ldots, x_{1n}, x_{21}, \ldots, x_{2n}, \ldots, x_{m1}, \ldots, x_{mn}$$

- ako je ulazni podatak tekst, tada se svaki karakter teksta kodira prirodnim brojem, pri čemu dobijamo niz brojeva. Kodiranje pojedinačnih karaktera se vrši na osnovu nekog unapred odabranog pridruživanja kôdova elementima karakterskog skupa (poput ASCII, UTF-8, i sl.)

- graf se može predstaviti svojom *matricom povezanosti* – matricom dimenzije $n \times n$ (gde je n broj čvorova grafa), takve da je element na poziciji (i, j) jednak 1 ako je grana (i, j) prisutna u grafu, u suprotnom je jednak 0. Nakon toga se matrica može predstaviti nizom brojeva na ranije prikazan način. Alternativno, za retke grafove može biti efikasnije navesti najpre broj čvorova n i broj grana m, a zatim m parova brojeva koji predstavljaju grane

[1] Kada kažemo *realni broj*, mislimo na podatke realnog tipa u standardnim programskim jezicima (poput `float` ili `double` u C++-u). Matematički, u pitanju su racionalni brojevi, s obzirom da se uvek zapisuju sa konačno mnogo cifara.

- zvučni zapis se predstavlja nizom *uzoraka* (*semplova*) koji predstavljaju brojeve koji opisuju nivo zvučnog signala u datom trenutku
- slika se predstavlja matricom *piksela* – brojeva koji opisuju boju slike u datoj tački. Ova matrica se predstavlja nizom brojeva kao što je ranije opisano
- video zapis se predstavlja nizom slika, pri čemu se svaka slika predstavlja nizom brojeva kao što je malopre opisano
- itd.

Dakle, većina podataka sa kojima naši programi u praksi rade se prirodno kodiraju nizovima ili torkama prirodnih brojeva. Ipak, tehnički gledano, ako algoritam posmatramo kao parcijalnu funkciju, tada bi svaki od ulaznih argumenata trebalo da bude *jedan* broj, a ne niz brojeva. Da bismo to postigli, potrebno je da pronađemo način da proizvoljni niz ili torku prirodnih brojeva kodiramo jedinstvenim brojem. Takođe, potrebno je da omogućimo dekodiranje, tj. izdvajanje pojedinačnih elemenata polaznog niza ili torke brojeva iz odgovarajućeg kôda. Funkcija kodiranja bi trebalo da bude *bijektivna*[2] – tj. da svakom nizu ili torci brojeva odgovara jedinstven kôd, kao i obrnuto. Najzad, i kodiranje i dekodiranje mora biti izračunljivo u formalnom smislu.

3.3.1 Kodiranje uređenih parova i torki

Uređeni par prirodnih brojeva (x, y) se može kodirati prirodnim brojem na sledeći način:

$$\pi(x, y) = \frac{(x + y)(x + y + 1)}{2} + y$$

Ova funkcija je poznata i kao *Kantorova funkcija uparivanja*. Jasno je da je ova funkcija primitivno rekurzivna. Takođe, može se pokazati da je kodiranje π bijektivno. Intuitivno, kodiranje se vrši po sledećoj šemi:

$$\begin{array}{c} (0,0) \\ (1,0)\quad (0,1) \\ (2,0)\quad (1,1)\quad (0,2) \\ (3,0)\quad (2,1)\quad (1,2)\quad (0,3) \\ \dots \end{array} \quad \longrightarrow \quad \begin{array}{c} 0 \\ 1\quad 2 \\ 3\quad 4\quad 5 \\ 6\quad 7\quad 8\quad 9 \\ \dots \end{array}$$

U vrsti sa rednim brojem s ($s = 0, 1, \dots$) u gornjem rasporedu parova se nalaze parovi takvi da je $x+y = s$. Pritom, parovi su u vrsti raspoređeni prema rastućoj vrednosti druge komponente y. Kôd para (x, y) se dobija brojanjem parova u gornjem rasporedu od vrha na dole, redom po vrstama, počev od nule dok se ne dođe do para (x, y). Izraz $\frac{(x+y)(x+y+1)}{2}$ određuje broj elemenata u prethodnim vrstama, pa se dodavanjem y dobija kôd para (x, y).

Dekodiranje kôda c se vrši na sledeći način:

[2] Ponekad u praksi može biti zgodnije da razmatramo i *višeznačna kodiranja* – gde isti podatak može biti kodiran na više različitih načina. Takođe, mogu postojati i neispravni kôdovi, tj. kôdovi koji ne kodiraju ni jedan podatak. Ipak, ono što je važno je da svakom kôdu uvek odgovara najviše jedan podatak, tj. dekodiranje mora biti jednoznačno.

- najpre se odredi najmanje s takvo da je $\frac{(s+1)(s+2)}{2} > c$. Ovim se određuje redni broj vrste s koja sadrži traženi par.
- zatim se izračuna $y = c - \frac{s(s+1)}{2}$ (oduzimamo od kôda broj elemenata u prethodnim vrstama)
- na kraju odredimo $x = s - y$, s obzirom da znamo da je $x + y = s$.

Funkcije dekodiranja ćemo označiti sa $\pi_1(c) = x$ i $\pi_2(c) = y$. Jasno je da je dekodiranje takođe izračunljivo i može se predstaviti primitivnom rekurzijom.

Primer 3.31. Par $(2, 3)$ će biti kodiran sa $\pi(2, 3) = \frac{(2+3)(2+3+1)}{2} + 3 = 15 + 3 = 18$. Dekodiranjem se najpre odredi da je $s = 5$ (jer je $\frac{(4+1)\cdot(4+2)}{2} = 15 < 18$, a $\frac{(5+1)\cdot(5+2)}{2} = 42 > 18$). Zatim se odredi $y = 18 - \frac{5\cdot(5+1)}{2} = 3$, nakon čega se odredi $x = s - 3 = 5 - 3 = 2$.

Kodiranje uređenih torki možemo svesti na kodiranje uređenih parova. Na primer:

$$\sigma(x, y, z) = \pi(\pi(x, y), z)$$

Izračunljivost i bijektivnost funkcije σ slede iz izračunljivosti i bijektivnosti funkcije π. Dekodiranje se takođe svodi na dekodiranje funkcije π. Na sličan način možemo kodirati i n-torke za proizvoljno n.

Primer 3.32. Neka je dat realan broj $x = 2.5$. Binarni zapis ovog broja je 10.1, što se u normalizovanom obliku može predstaviti kao $1.01 \cdot 2^1$. Znak ovog broja je + (kodiran brojem 0), mantisa je 101 (odnosno 5 u dekadnom zapisu), a eksponent je +1. Najpre ćemo eksponent kodirati kao uređeni par $(0, 1)$ (znak i apsolutna vrednost). Važi da je $\pi(0, 1) = 2$. Sada je potrebno kodirati uređenu trojku $(0, 5, 2)$. Prema prethodnom, imamo:

$$\sigma(0, 5, 2) = \pi(\pi(0, 5), 2) = \pi(20, 2) = 255$$

Primer 3.33. Pretpostavimo da naš algoritam na ulazu prihvata jedan argument koji predstavlja tačku u ravni, zadatu parom koordinata. Algoritam treba da odredi kvadrant u kom se tačka nalazi.

Tačku bismo kodirali tako što najpre njene koordinate x i y (koje su realni brojevi) kodiramo kao prirodne brojeve x' i y' (kao u prethodnom primeru). Zatim kodiramo odgovarajući uredni par brojeva (x', y') funkcijom π.

Algoritam bi interno dekodirao x' i y', a zatim bi njihovim dekodiranjem odredio znakove brojeva x i y. Na osnovu tih znakova bi odredio kvadrant kome tačka pripada.

3.3.2 Kodiranje konačnih nizova brojeva

Ako nam je ulaz niz (lista, sekvenca) prirodnih brojeva proizvoljne konačne dužine, tada možemo koristiti sledeću formulu kodiranja:

$$\begin{aligned} \tau([\,]) &= 0 \\ \tau([x_0, x_1, \ldots, x_{k-1}]) &= \pi(x_0, \tau([x_1, \ldots, x_{k-1}])) + 1 \end{aligned}$$

Dakle, prazna lista se kodira nulom, dok se neprazne liste dele na *glavu* (x_0) i *rep* $([x_1, \ldots, x_{k-1}])$. Rep se rekurzivno kodira funkcijom τ, a zatim se primenjuje funkcija π za kodiranje parova. Dodavanje jedince je neophodno zbog

jednoznačnosti dekodiranja – bez toga bi npr. liste $[0]$, $[0,0]$ i $[0,0,0]$ sve bile kodirane nulom, kao i prazna lista, pa kôd 0 ne bi mogao jednoznačno da se dekodira. Jasno je da je postupak kodiranja izračunljiv i da se može predstaviti primitivnom rekurzijom.

Dekodiranje kôda c se vrši na sledeći način:

1. ako je $c = 0$, tada je u pitanju prazna lista $[\,]$
2. ako je $c > 0$, tada određujemo $x_0 = \pi_1(c-1)$ i $t_0 = \pi_2(c-1)$. Element x_0 je glava liste, dok t_0 predstavlja kôd repa liste
3. dalje na t_0 primenjujemo isti postupak $(1-3)$ čime se dobija x_1 i t_1
4. postupak se završava kada je $t_i = 0$.

Postupak dekodiranja je takođe efektivno izračunljiv i primitivno rekurzivan.

Primer 3.34. Za sekvencu $[2,3,1]$ imamo $\tau([2,3,1]) = \pi(2,\tau([3,1])) + 1 = \pi(2,\pi(3,\tau([1]))+1)+1 = \pi(2,\pi(3,\pi(1,\tau([\,]))+1)+1)+1 = \pi(2,\pi(3,\pi(1,0)+1)+1)+1 = \pi(2,\pi(3,2)+1)+1 = \pi(2,18)+1 = 229$. Obratno, da dekodiramo broj 229, umanjujemo ga za jedan i zatim ga dekodiramo funkcijama π_1 i π_2, odakle dobijamo $x_0 = 2$ i $t_0 = 18$. Dalje isti postupak primenjujemo na t_0, odakle dobijamo $x_1 = 3$ i $t_1 = 2$. Zatim umanjujemo t_1 za jedan i dekodiranjem dobijamo $x_2 = 1$ i $t_2 = 0$. Kada smo stigli do nule, znamo da je nastupio kraj sekvence, tj. rezultat je $[2,3,1]$.

U praksi, prilikom dekodiranja, mi obično nećemo raspakivati cele sekvence, već će nam biti važno da možemo da izdvojimo i-ti element iz sekvence ($i = 0,1,\ldots$), pod pretpostavkom da postoji. Ovo možemo uraditi tako što i puta primenimo funkciju π_2, a zatim primenimo funkciju π_1:

$$\tau^{-}(c,i) = \pi_1(\underbrace{\pi_2(\pi_2(c-1)-1)\ldots)}_{i}) = \pi_1(rep(Sub(\pi_2, pred))(i,c))$$

Dakle, u pitanju je primitivno rekurzivna funkcija. Slično, može nam biti potrebno da u sekvenci koja je kodirana kôdom c i-ti element postavimo na vrednost x i odredimo kôd tako izmenjene sekvence. Ovu funkciju ćemo označavati sa $\tau^{\rightarrow}(c,i,x)$. Može se pokazati da je i ova funkcija primitivno rekurzivna. Najzad, može biti potrebno da odredimo dužinu sekvence kodirane kôdom c. Ovu funkciju označavaćemo sa $len(c)$. I ova funkcija se može predstaviti primitivnom rekurzijom, što ostavljamo čitaocu za vežbu.

Napomena 3.24. Napomenimo da postoje i drugi načini kodiranja sekvenci brojeva. Na primer, jedan takav način je da se sekvenca $[x_1,\ldots,x_k]$ kodira kao:

$$p_1^{x_1+1} \cdot p_2^{x_2+1} \cdot \ldots \cdot p_k^{x_k+1}$$

gde je p_i i-ti prost broj ($p_1 = 2, p_2 = 3, p_3 = 5, \ldots$). Ovo kodiranje je poznato i kao *Gedelovo kodiranje*. Ovakvo kodiranje neće biti bijektivno, s obzirom da postoje brojevi koji neće biti ispravni kôdovi (npr. broj $2 \cdot 5$ neće biti ispravan kôd, s obzirom da nema faktora 3, a u kodiranju se uvek koriste prvih k prostih brojeva). Ipak, dekodiranje ispravnih kôdova će biti jednoznačno, s obzirom da je faktorizacija prirodnih brojeva jednoznačna.

Drugi način, bliži realnim reprezentacijama u računaru, je da se sekvenca $[x_0, \ldots, x_{k-1}]$ kodira kao:

$$\sigma(k, m, s)$$

gde je k dužina sekvence, m je najmanji broj bitova potreban da se predstavi svaki od brojeva x_i iz sekvence (tj. $m = max_{i=0}^{k-1}(\mu j[2^j > x_i])$), a s je broj dobijen na sledeći način:

$$s = x_0 + 2^m \cdot x_1 + 2^{2m} \cdot x_2 + \ldots + 2^{(k-1)\cdot m} x_{k-1}$$

Inutuitivno, ,,pakujemo" binarne zapise elemenata sekvence jedan do drugog i dobijeni broj s predstavlja kôd. Da bi dekodiranje bilo jednoznačno, potrebno je da u kôd ugradimo i informaciju o dužini sekvence i broju bitova za svaki element. Sa druge strane, kodiranje nije jednoznačno, zato što dodavanje dodatnih bitova više težine u reprezentaciju broja s ne utiče na vrednost dekodirane sekvence (pri fiksiranim vrednostima k i m). Drugim rečima, ista sekvenca se može kodirati različitim kôdovima, ali svakom kôdu odgovara tačno jedna sekvenca. Ovo kodiranje odgovara načinu reprezentacije podataka u realnim računarima – svaki podatak se predstavlja nizom bajtova koji možemo posmatrati kao jedan veliki, binarno zapisani, prirodan broj i koji možemo smatrati kôdom tog podatka.

3.3.3 Kodiranje hijerarhijskih struktura

U računarstvu je često potrebno predstaviti hijerarhijske strukture, poput izraza, formula, dokaza i sl. Ovakve strukture se po pravilu predstavljaju *stablima*. Svaki čvor stabla ima 0 ili više *dece*. Pritom, svi čvorovi imaju tačno jednog roditelja, izuzev jednog izdvojenog čvora – *korena* stabla koji nema roditelja.

Da bismo kodirali stablo, najpre treba sadržaj svakog čvora kodirati prirodnim brojem. Npr. ako je u pitanju stablo izraza, tada čvorovi mogu sadržati operatore (npr. $+, -, \cdot, /$ i sl.). Svaki od operatora se kodira jedinstvenim prirodnim brojem. Ako čvor sadrži tekst, tada najpre tekst kodiramo sekvencom prirodnih brojeva (kôdovima karaktera), a zatim tu sekvencu kodiramo prirodnim brojem, kao u prethodnom odeljku. Slično i u ostalim slučajevima.

Dakle, u nastavku možemo smatrati da je sadržaj svakog čvora stabla jedan prirodan broj. Funkciju kodiranja stabla $\rho(t)$ možemo definisati na sledeći način: posmatrajmo stablo t kao uređenu torku $t = (r, t_1, \ldots, t_k)$, gde je r broj koji je sadržan u korenu, a $t_1, \ldots, t_k$ su torke koje na analogan način kodiraju podstabla. Najpre ćemo rekurzivno kodirati podstabla brojevima $c_i = \rho(t_i)$. Sada je još potrebno kodirati sekvencu $[r, c_1, \ldots, c_k]$ funkcijom τ:

$$\rho((r, t_1, \ldots, t_k)) = \tau([r, \rho(t_1), \ldots, \rho(t_k)])$$

Ovo kodiranje je bijektivno, tj. svakom stablu odgovaraće jedinstven prirodan broj i obrnuto.

Primer 3.35. Posmatrajmo stablo $(5, (1, (0), (1)), (4))$. Ovo stablo ima 5 u korenu, levo podstablo ima u korenu 1 i kao svoju decu listove 0 i 1. Desno podstablo je samo list 4. Kodiranjem ovog stabla dobijamo:

$$\rho((5, (1, (0), (1)), (4))) = \tau([5, \rho((1, (0), (1))), \rho((4))])$$

Kako je:

$$\rho((1,(0),(1))) = \tau([1,\rho((0)),\rho((1))]) = \tau([1,\tau([0]),\tau([1])) = \tau([1,1,2]) = 252$$

imamo da je:

$$\tau([5,\rho((1,(0),(1))),\rho((4))]) = \tau([5,252,\tau([4])]) = \tau([5,252,11]) = 1306346050$$

3.4 Enumeracija izračunljivih funkcija

3.4.1 Kodiranje URM programa

U prethodnom poglavlju smo razmatrali na koji način možemo da kodiramo različite podatke, tj. da ih predstavimo jednim prirodnim brojem. U ovom poglavlju ćemo pokušati da primenimo to znanje na same algoritme. Naime, pretpostavimo da su nam algoritmi predstavljeni URM programima. Svaki URM program je konačni niz instrukcija. Pritom, svaka instrukcija se može kodirati brojem:

- Instrukcija $Z(k)$ se kodira kao $4(k-1)$
- Instrukcija $S(k)$ se kodira kao $4(k-1)+1$
- Instrukcija $T(m,n)$ se kodira kao $4\pi(m-1,n-1)+2$
- Instrukcija $J(m,n,k)$ se kodira kao $4\sigma(m-1,n-1,k-1)+3$

Oduzimanje jedinice u parametrima je zato što numeracija instrukcija i registara kreće od 1. Prilikom dekodiranja, najpre se odredi tip instrukcije pomoću funkcije *mod* (ostatak pri deljenju sa 4). U slučaju instrukcija Z i S se celobrojnim deljenjem sa četiri dobija $k-1$. Kod T instrukcije se celobrojnim deljenjem sa četiri odredi $\pi(m-1,n-1)$, a zatim se dekodiranjem funkcije π dobijaju m i n. Slično za J instrukcije.

Označimo opisanu funkciju kodiranja instrukcija sa ϕ_I. Program $P = I_1,\ldots,I_n$ se kodira kao:

$$\phi(P) = \tau([\phi_I(I_1),\phi_I(I_2),\ldots,\phi_I(I_n)])$$

Lako se vidi da je ovo kodiranje bijektivno i izračunljivo.

Primer 3.36. Setimo se programa $P = J(3,2,5),S(1),S(3),J(1,1,1)$ koji je računao zbir dva broja. Ovaj program možemo kodirati tako što najpre kodiramo njegove instrukcije:

- $\phi_I(J(3,2,5)) = 4\sigma(2,1,4)+3 = 283$
- $\phi_I(S(1)) = 4\cdot(1-1)+1 = 1$
- $\phi_I(S(3)) = 4\cdot(3-1)+1 = 9$
- $\phi_I(J(1,1,1)) = 4\sigma(0,0,0)+3 = 3$

Sada je:

$$\phi(P) = \tau([283,1,9,3]) = 60659322$$

Primer 3.37. Odredimo URM program čiji je kôd broj 20. Najpre ćemo odrediti sekvencu brojeva koja se funkcijom τ slika u 20. Lako se može videti da je to sekvenca $[1, 2]$. Dekodiranjem kodova 1 i 2 dobijamo instrukcije $S(0)$ i $T(1, 1)$. Dakle, imamo program $S(0), T(1, 1)$.

Napomena 3.25. Na sličan, doduše tehnički složeniji, način je moguće kodirati i μ-rekurzivne funkcije. S obzirom da je rekurzivna funkcija jednoznačno zadata svojim konstruktivnim zapisom, koji se može predstaviti svojim apstraktnim stablom, logičan pristup je da se rekurzivne funkcije kodiraju po principima kodiranja stabala iz prethodnog odeljka. Ono što komplikuje situaciju je to što mnoga stabla neće predstavljati ispravne rekurzivne konstrukcije, s obzirom da će operatori u njima biti primenjeni na funkcije pogrešne arnosti. Otuda će mnogi kodovi odgovarati neispravnim konstrukcijama. Jedan način da se sa tim problemom izborimo je da kažemo da svaka neispravna rekurzivna konstrukcija predstavlja parcijalnu funkciju *undef* (tj. funkciju za koju je $undef(x) = -$,; za svako x). Drugi način je da se definiše posebna funkcija kodiranja za svaku arnost. Za detalje pogledati literaturu.

3.4.2 Enumeracija URM programa

Imajući u vidu prethodno opisano kodiranje URM programa, sledi da se svi URM programi mogu efektivno poređati u niz:

$$P_0, P_1, \ldots,$$

Dakle, i-ti program u nizu označavaćemo sa P_i, pri čemu ćemo broj i nazivati *kôdom* programa. Uz fiksirano kodiranje, svaki program će imati jedinstven kôd i obrnuto – svaki prirodan broj biće kôd nekog programa.

Kao što znamo, svaki URM program definiše po jednu parcijalnu funkciju za svaku arnost k. Funkciju arnosti k koju definiše program P_i označavaćemo sa $\phi_i^{(k)}$ (ukoliko je arnost 1, tada ćemo kraće pisati samo ϕ_i). Pritom, setimo se da postoje različiti URM programi koji izračunavaju istu parcijalnu funkciju. Otuda se može dogoditi da je $\phi_i^{(k)} = \phi_j^{(k)}$ za $i \neq j$. Drugim rečima, u nizu funkcija:

$$\phi_0^{(k)}, \phi_1^{(k)}, \ldots$$

može biti jednakih funkcija, izračunatih na različite načine.

Mogućnost enumeracije svih URM programa ima neke vrlo zanimljive posledice. Na primer, posmatrajmo funkciju definisanu na sledeći način:

$$f(x) \quad = \quad \begin{cases} \phi_x(x) + 1, & \text{ako je } \phi_x(x) \text{ definisano} \\ 0, & \text{inače} \end{cases}$$

Ova funkcija je totalna, jer je definisana za svako x. Međutim, ako bi ova funkcija bila izračunljiva, tada bi postojao URM program P_e koji je izračunava, tj. važilo bi:

$$f(x) = \phi_e(x)$$

za svako x. Posebno, to bi značilo da je $\phi_e(e) = \phi_e(e) + 1$, s obzirom da je f totalna, pa je $f(e) = \phi_e(e)$ definisano. Ovo nije moguće. Otuda sledi da f nije izračunljiva. Dakle, upravo smo dokazali sledeću teoremu.

Teorema 3.6. *Postoji totalna funkcija koja nije URM izračunljiva.*

Imajući u vidu ekvivalentnost URM programa i μ-rekurzivnih funkcija (teoreme 3.4 i 3.5), odavde sledi i da postoji totalna funkcija koja nije μ-rekurzivna. Time smo dokazali i teoremu 3.3.

Napomena 3.26. Postupak primenjen u prethodnom dokazu je poznat i kao *postupak dijagonalizacije*. Mnoge teoreme u teorijskom računarstvu i matematici se dokazuju na taj način (npr. neprebrojvost skupa realnih brojeva se tako dokazuje). Takođe, ovaj postupak je ,,odgovoran" i za otkrivanje mnogih paradoksa u teoriji skupova u 19. veku.

3.4.3 Univerzalni programi i funkcije

Posmatrajmo funkciju:

$$U^{(k)}(e, x_1, \ldots, x_k) = \phi_e^{(k)}(x_1, \ldots, x_k)$$

Intuitivno, ova funkcija najpre dekodira svoj prvi argument i odredi program P_e. Zatim izršava program P_e nad ulazima $x_1, \ldots, x_k$. Ovakvu funkciju ćemo nazivati *univerzalna funkcija*, s obzirom da ona može oponašati bilo koju URM izračunljivu funkciju, pod pretpostavkom da joj je na raspolaganju kôd odgovarajućeg programa.

Teorema 3.7. *Univerzalna funkcija $U^{(k)}$ je μ-rekurzivna.*

Dokaz. (skica) Svaki URM program koristi konačan broj registara $R_1, \ldots, R_m$. Otuda se *stanje* programa u svakom trenutku može predstaviti sekvencom $[i, r_1, r_2, \ldots, r_m]$, gde je i indeks sledeće instrukcije, a r_j je vrednost registra R_j. Ovu sekvencu možemo kodirati funkcijom τ koja je primitivno rekurzivna. Neka je sa c označen kôd trenutnog stanja programa čiji je kôd e. Razmotrimo sledeće funkcije:

- $c_0 = init(e, x_1, \ldots, x_k)$ vraća kôd inicijalnog stanja $c_0 = \tau([1, x_1, \ldots, x_k, 0, 0, \ldots, 0])$ (prvih k registara se inicijalizuje argmentima $x_1, \ldots, x_k$, dok se preostalih $m - k$ registara inicijalizuju nulama). Vrednost m (ukupan broj registara koje program koristi) se može odrediti dekodiranjem kôda programa e. Ova funkcija se može predstaviti primitivnom rekurzijom
- $i = next(c) = \tau^-(c, 0)$ vraća indeks i naredne instrukcije u stanju c. Kako je τ^- primitivno rekurzivna, funkcija $next$ je takođe takva
- $r_i = r(c, i) = \tau^-(c, i)$ vraća vrednost i-tog registra u trenutnom stanju c
- $s = inst(e, i) = \tau^-(e, i - 1)$ vraća kôd i-te instrukcije programa P_e (ako postoji, inače je nedefinisana).
- $c' = set(c, i, x) = \tau^{\rightarrow}(c, i, x)$ postavlja vrednost registra R_i na vrednost x u stanju c i vraća kôd tako izmenjenog stanja c'. Kako je $\tau^{\rightarrow}$ primitivno rekurzivna, takva je i funkcija set
- $c' = set\_next(c, j) = \tau^{\rightarrow}(c, 0, j)$ postavlja indeks naredne instrukcije u stanju c na j i vraća tako izmenjeno stanje

- $c' = exec(e, c)$ izvršava sledeću instrukciju programa e nad stanjem c i vraća novo stanje programa. Intuitivno, ova funkcija:
  - izdvaja indeks i naredne instrukcije (funkcijom $next(c)$)
  - izdvaja kôd i-te instrukcije programa e ($s = inst(e, i)$)
  - dekodira s i na osnovu tipa instrukcije i njenih parametara izdvaja vrednosti odgovarajućih registara iz stanja c (r funkcijom) i zamenjuje ih izmenjenim vrednostima (set funkcijom). Grananje po različitim tipovima instrukcija se može opisati funkcijom $cond$, a na osnovu ostatka pri deljenju sa 4 (funkcija mod).
  - u tekućem stanju zamenjuje indeks i tekuće instrukcije indeksom $i+1$, ili odgovarajućim indeksom na koji se skače u slučaju instrukcije tipa J (set_next funkcijom).

  Na ovaj način se određuje kôd izmenjenog stanja c' izvršavanjem tekuće instrukcije. Primetimo da je funkcija $exec$ dobijena supstitucijom većeg broja funkcija koje su sve primitivno rekurzivne. Otuda je i $exec$ primitivno rekurzivna.

- $c' = exec\_all(e, c, j)$ izvršava narednih j koraka programa P_e polazeći od stanja c i vraća stanje c' dobijeno nakon toga. Ova funkcija će j puta primeniti funkciju $exec$ na c:

$$\begin{aligned} exec\_all(e, c, 0) &= c \\ exec\_all(e, c, j+1) &= exec(e, exec\_all(e, c, j)) \end{aligned}$$

  Dakle, i ova funkcija je primitivno rekurzivna.

Sada funkcija:

$$halts(e, x_1, \ldots, x_k) = \\ \mu z[\pi_1(z) = exec\_all(e, init(e, x_1, \ldots, x_k), \pi_2(z)) \wedge next(\pi_1(z)) > len(e)]$$

primenjuje program P_e na početno stanje $c_0 = \tau([1, x_1, \ldots, x_k])$ i određuje minimalni kôd z para (c, j) (tj. $z = \pi(c, j)$), takav da je c završno stanje (tj. indeks naredne instrukcije u stanju c je veći od dužine programa $len(e)$), a j je broj koraka potreban da se u to stanje dođe, polazeći od početnog stanja $c_0 = \tau([1, x_1, \ldots, x_k])$.

Vrednost koju program P_e vraća za ulaz $(x_1, \ldots, x_k)$ (tj. vrednost funkcije $U^{(k)}(e, x_1, \ldots, x_k)$) je sda:

$$U^{(k)}(e, x_1, \ldots, x_k) = \tau^-(\pi_1(halts(e, x_1, \ldots, x_k)), 1)$$

Drugim rečima, potrebno je iz kôda $z = \pi(c, j)$ izdvojiti završno stanje c, a zatim iz završnog stanja izdvojiti vrednost registra R_1. Primetimo da će ova funkcija biti nedefinisana za one $x_1, \ldots, x_k$ za koje se program P_e nikada ne zaustavlja. □

Kako je, na osnovu teoreme 3.4, svaka μ-rekurzivna funkcija URM izračunljiva, imamo sledeću posledicu.

Posledica 3.1. *Univerzalna funkcija $U^{(k)}$ je URM izračunljiva.*

Napomena 3.27. URM-izračunljivost funkcije $U^{(k)}$ je od izuzetnog teorijskog i praktičnog značaja. Program koji izračunava funkciju $U^{(k)}$ može simulirati rad proizvoljnog algoritma nad datim podacima. Ovo je teorijski osnov za pojam *interpretacije*. Naime, u teoriji programskih jezika, *interpretator* je program koji simulira rad drugih programa za zadati ulaz. Na primer, interpretator programskog jezika *Python* na ulazu ima *Python* program, kao i ulazne podatke nad kojima taj program treba izvršiti. Interpretator tumači (dekodira) naredbe *Python* programa i izvršava njihov efekat nad datim podacima na računaru. Dakle, interpretator je program koji može izvršavati bilo koji algoritam napisan u programskom jeziku *Python*, tj. ponaša se upravo kao univerzalna funkcija.

Kao što znamo, interpretacija je na realnim računarima prisutna i na hardverskom nivou – moderni procesori su zapravo hardverski interpretatori za programe napisane na odgovarajućem *mašinskom jeziku*. Kontrolna jedinica procesora dohvata jednu po jednu instrukciju programa i simulira njen efekat uz pomoć aritmetičko logičke jedinice i registara procesora. Dakle, procesor može izvršavati bilo koji algoritam kodiran na mašinskom jeziku. Otuda je univerzalna funkcije $U^{(k)}$ teorijski osnov i za *računare sa uskladištenim programom* – program (tj. njegov kôd) se nalazi u memoriji i dekodira se od strane hardverskog interpretatora da bi se izvršavao. Sa druge strane, *računari sa fiksiranim programom* odgovaraju konkretnom, fiksiranom URM programu P – samo podaci se nalaze u memoriji, dok je program koji se nad njima izvršava fiksiran.

Primetimo da ranije navedena teorema 3.5 čiji smo dokaz ostavili za kasnije sada lako sledi kao posledica teoreme 3.7.

Posledica 3.2. *Svaka URM izračunljiva funkcija je μ-rekurzivna.*

Dokaz. Neka je f proizvoljna URM izračunljiva funkcija arnosti k. To znači da postoji URM program P_e koji izračunava funkciju f. Odatle je:

$$f(x_1, \ldots, x_k) = U^{(k)}(e, x_1, \ldots, x_k) = U^{(k)}(C_k^e(x_1, \ldots, x_k), x_1, \ldots, x_k)$$

za svako $(x_1, \ldots, x_k)$, tj. važi:

$$f = Sub(U^{(k)}, C_k^e, P_k^1, \ldots, P_k^k)$$

Dakle, f je μ-rekurzivna. □

Posledica 3.3 (Klinijeva teorema o normalnoj formi)**.** *Svaka μ-rekurzivna funkcija se može predstaviti u obliku koji koristi samo jednu primenu operatora μ, tj. u obliku:*

$$f(x_1, \ldots, x_k) = P(\mu z[Q(x_1, \ldots, x_k, z) = 0])$$

gde su P i Q primitivno rekurzivne funkcije.

Dokaz. Iz dokaza teoreme 3.7 vidimo da je univerzalna funkcija $U^{(k)}$ formulisana tako da koristi samo jedan operator minimizacije. Svaka μ-rekurzivna funkcija f je URM izračunljiva (teorema 3.4), pa postoji program P_e koji je izračunava. Kao u dokazu prethodne posledice, važi da je $f = Sub(U^{(k)}, C_k^e, P_k^1, \ldots, P_k^k)$ što je upravo μ-rekurzivna formulacija funkcije f koja sadrži samo jednu primenu operatora μ (unutar formulacije funkcije $U^{(k)}$). □

3.4.4 *s-m-n* teorema

Teorema 3.8. *Za svako m i n postoji totalna izračunljiva funkcija s_n^m arnosti $m+1$ takva da važi:*

$$\phi_e^{(m+n)}(x_1, \ldots, x_{m+n}) = \phi^{(n)}_{s_n^m(e,x_1,\ldots,x_m)}(x_{m+1}, \ldots, x_{m+n})$$

za svako $e, x_1, \ldots, x_{m+n}$.

Dokaz. Da bismo dokazali ovu teoremu, možemo razmišljati u terminima URM programa. Naime, pretpostavimo da imamo program P_e koji izračunava funkciju $\phi_e^{(m+n)}$. Neka su nam vrednosti $x_1, \ldots, x_m$ fiksirane. Potrebno je konstruisati program koji izračunava funkciju g koja nastaje od ϕ_e^{m+n} fiksiranjem vrednosti prvih m argumenata. Takva funkcija bi kao argumente imala $x_{m+1}, \ldots, x_{m+n}$ i oni bi, po pretpostavci, trebalo da budu smešteni u prvih n registara $R_1, \ldots, R_n$. Odgovarajući program P' bi trebalo da najpre premesti ove argumente u registre $R_{m+1}, \ldots, R_{m+n}$, a da zatim u $R_1, \ldots, R_m$ upiše vrednosti $x_1, \ldots, x_n$. Nakon toga bi trebalo da nastavi sa izvršavanjem programa P_e. Dakle, program P' bi izgledao ovako:

$$\begin{array}{l}
T(n, m+n) \\
T(n-1, m+n-1) \\
\ldots \\
T(1, m+1) \\
Z(1) \\
\left.\begin{array}{l} S(1) \\ \ldots \\ S(1) \end{array}\right\} x_1 \text{ puta} \\
Z(2) \\
\left.\begin{array}{l} S(2) \\ \ldots \\ S(2) \end{array}\right\} x_2 \text{ puta} \\
\ldots \\
Z(m) \\
\left.\begin{array}{l} S(m) \\ \ldots \\ S(m) \end{array}\right\} x_m \text{ puta} \\
P_e
\end{array}$$

Kôd ovog programa možemo efektivno odrediti na osnovu e i $x_1, \ldots, x_m$: najpre dekodiramo e i dobijemo program P_e, zatim ga transformišemo na gore opisani način i onda dobijeni program ponovo kodiramo. Dobijeni rezultat je upravo ono što vraća funkcije $s_n^m(e, x_1, \ldots, x_m)$. Iz intuitivne izračunljivosti opisanog postupka sledi i formalna izračunljivost funkcije s_n^m, prema Čerčovoj tezi. □

Napomena 3.28. Prethodna teorema poznata je i kao *s-m-n* teorema. Intuitivno, ona tvrdi da kada u nekoj izračunljivoj funkciji arnosti $m+n$ fiksiramo prvih m argumenata, dobijamo funkciju po preostalih n argumenata koja je takođe izračunljiva, pri čemu je moguće efektivno odrediti program koji je izračunava (tj. moguće je efektivno izračunati njegov kôd). Taj kôd se može odrediti totalnom efektivno izračunljivom funkcijom s_n^m na osnovu vrednosti prvih m argumenata.

Na primer, posmatrajmo program:

$$\begin{array}{l} fun \quad f(x,y) \\ \{ \\ \quad return \quad x+y; \\ \} \end{array}$$

U pitanju je program koji izračunava zbir svoja dva argumenta. Ako bismo fiksirali da je $x = 3$, dobili bismo program:

$$\begin{array}{l} fun \quad g(y) \\ \{ \\ \quad return \quad 3+y; \\ \} \end{array}$$

Dakle, ako u programu koji izračunava funkciju f fiksiramo prvi argument, dobijamo novi program koji izračunava funkciju g, i taj program možemo efektivno odrediti na osnovu izabrane vrednosti argumenta x.

Dakle, suština je da je moguće na algoritamski način odrediti kako se transformiše proizvoljni program kada mu fiksiramo neke od argumenata.

Napomena 3.29. Mnogi funkcionalni programski jezici dopuštaju *parcijalnu primenu* funkcija:

$$f(x,y) = f(x)(y)$$

Dakle, da bismo izračunali funkciju f arnosti 2 za ulaz (x,y), možemo najpre *parcijalno primeniti* funkciju f nad prvim argumentom, čime dobijamo *novu funkciju* $f(x)$ arnosti 1, koju dalje primenjujemo na preostali argument y. Pritom, *s-m-n* teorema nam daje teorijsku podlogu za ovu parcijalnu primenu – ona tvrdi da postoji algoritam kojim se na osnovu algoritma za izračunavanje funkcije f i vrednosti prvog argumenta konstruiše algoritam za izračunavanje funkcije $f(x)$.

Napomena 3.30. Ako u *s-m-n* teoremi fiksiramo e, dobijamo konkretnu funkciju $f = \phi_e^{(m+n)}$, za koju *s-m-n* teorema tvrdi da važi:

$$f(x_1, \ldots, x_{m+n}) = \phi^{(n)}_{s'(x_1,\ldots,x_m)}(x_{m+1}, \ldots, x_{m+n})$$

pri čemu je s' neka totalna izračunljiva funkcija arnosti m (konkretno, $s'(x_1, \ldots, x_m) = s_n^m(e, x_1, \ldots, x_m)$ za to fiksirano e). Često ćemo za konkretne funkcije upravo ovako izražavati *s-m-n* teoremu.

Teorema 3.9. *Neka je data totalna izračunljiva funkcija f arnosti* 1. *Tada postoji neko $n \in \mathbb{N}$ za koje važi:*

$$\phi_{f(n)} = \phi_n$$

Dokaz. Posmatrajmo funkciju g arnosti 2 definisanu na sledeći način:

$$g(x,y) = \phi_{f(\phi_x(x))}(y)$$

Ova funkcija je izračunljiva, jer se može dobiti supstitucijom nad izračunljivim funkcijama $U^{(1)}$ i f:

$$g(x,y) = \phi_{f(\phi_x(x))}(y) = \phi_{f(U^{(1)}(x,x))}(y) = U^{(1)}(f(U^{(1)}(x,x)),y)$$

Samim tim, prema *s-m-n* teoremi, postoji totalna izračunljiva funkcija s arnosti 1 takva da je:

$$g(x,y) = \phi_{s(x)}(y)$$

za svako x. Otuda imamo:

$$\phi_{s(x)}(y) = \phi_{f(\phi_x(x))}(y)$$

Ovo važi za svako x i y. Sa druge strane, kako je s izračunljiva, tada postoji neko e takvo da je $s = \phi_e$. Otuda je:

$$\phi_{\phi_e(x)}(y) = \phi_{f(\phi_x(x))}(y)$$

Ako u gornjoj jednakosti stavimo $x = e$, dobijamo:

$$\phi_{\phi_e(e)}(y) = \phi_{f(\phi_e(e))}(y)$$

za svako y. Otuda za $n = \phi_e(e)$ važi:

$$\phi_n(y) = \phi_{f(n)}(y)$$

za svako y, tj. važi:

$$\phi_n = \phi_{f(n)}$$

□

Napomena 3.31. Opisana teorema je poznata i kao *teorema o fiksnoj tački*. Intuicija je sledeća: funkcija f preslikava jedan kôd programa u drugi: $x \mapsto f(x)$. Otuda preslikavanje $\phi^{-1} \circ f \circ \phi$ transformiše program P_e u program $P_{f(e)}$. Dakle, imamo algoritam koji transformiše jedan program u drugi. Na primer, posmatrajmo sledeću transformaciju:

$$\begin{array}{l} g \mapsto fun \quad (x) \\ \quad \{ \\ \qquad if(x = 0) \quad return \quad 1; \\ \qquad return \quad x \cdot g(x-1); \\ \quad \} \end{array}$$

Ovom transformacijom se funkcija g arnosti 1 transformiše u drugu funkciju arnosti 1 opisanu datim pseudokodom (koji iznutra poziva g, pa samim tim zavisi od g). Teorema o fiksnoj tački tvrdi da će za neku pogodno odabranu izračunljivu funkciju h arnosti 1 važiti:

$$\begin{array}{l} h = fun \quad (x) \\ \quad \{ \\ \qquad if(x = 0) \quad return \quad 1; \\ \qquad return \quad x \cdot h(x-1); \\ \quad \} \end{array}$$

Dakle, kada transformišemo h na opisani način, ponovo dobijamo funkciju h. Drugim rečima, ova transformacija svodi h na samu sebe. Ovo opravdava definisanje funkcija pomoću rekurzije. Preciznije, teorema tvrdi da svaka efektivno

zadata rekurzivna specifikacija definiše izračunljivu funkciju. Ovo ,,efektivno zadata" podrazumeva najopštiju moguću rekurziju, ne samo primitivnu. Zato se ova teorema često naziva i *teoremom o rekurziji*.

Glava 4

Problemi odlučivanja

Problem odlučivanja (engl. *decision problem*) je problem kod koga se za svaku instancu kao odgovor očekuje ,,da" ili ,,ne". Dakle, u pitanju su problemi kod kojih je potrebno odrediti da li ulazna instanca ima neko svojstvo ili zadovoljava neku relaciju.

Primer 4.1. Neki primeri problema odlučivanja:

- Da li je dati broj x paran?
- Da li je dati broj x veći od datog broja y?
- Da li se dati broj x nalazi u datom nizu brojeva a?
- Da li je data iskazna formula F zadovoljiva?
- Da li se dati program P zaustavlja za dati ulaz x?
- Da li postoji put u grafu G koji povezuje dva data čvora u i v?
- Da li u grafu G postoji ciklus?
- ...

Kod svih ovih problema, očekuje se odgovor ,,da" ili ,,ne" na izlazu.

U terminima formalne izračunljivosti, problemi odlučivanja predstavljaju *relacije* nad $\mathbb{N}$. Ako k-torka $(x_1, \ldots, x_k) \in \mathbb{N}^k$ zadovoljava relaciju ρ arnosti k nad $\mathbb{N}$, tada ćemo pisati $\rho(x_1, \ldots, x_k)$, u suprotnom, pisaćemo $\neg\rho(x_1, \ldots, x_k)$. Intuitivno, torke koje zadovoljavaju relaciju odgovaraju instancama problema odlučivanja za koje je odgovor potvrdan, kao i obratno. U nastavku teksta, za termine *relacija* i *problem odlučivanja* smatraćemo da imaju isto značenje, pri čemu ćemo termin *relacija* koristiti pre svega u formalnim razmatranjima, dok će termin *problem odlučivanja* pretežno biti korišćen u intuitivnim objašnjenjima.

4.1 Odlučivost

Definicija 4.1. Neka je data proizvoljna relacija ρ arnosti k nad $\mathbb{N}$. Funkcija:

$$\chi_\rho(x_1, \ldots, x_k) = \begin{cases} 1, & \text{ako } \rho(x_1, \ldots, x_k) \\ 0, & \text{u suprotnom} \end{cases}$$

se naziva *karakteritična funkcija* relacije ρ.

Ključno pitanje je da li je za datu relaciju ρ njena karakteristična funkcija izračunljiva? Ovo nas vodi ka sledećoj definiciji.

Definicija 4.2. Za relaciju ρ kažemo da je *odlučiva* (engl. *decidable*), ako je njena karakteristična funkcija izračunljiva.

Napomena 4.1. Intuitivno, problem je odlučiv ako postoji efektivan postupak (algoritam) kojim se za ulaznu instancu utvrđuje da li je odgovor ,,da" ili ,,ne". Takav algoritam se naziva i *procedura odlučivanja* (engl. *decision procedure*) za dati problem.

Primer 4.2. Primeri odlučivih problema su:

- Da li je dati broj x paran?
- Da li je broj x deljiv datim brojem y?
- Da li je dati broj x veći od datog broja y?
- Da li je dati broj x prost?
- Da li je dati graf G povezan?
- Da li u datom nizu a postoji element jednak datom broju x?
- ...

Za neke od ovih problema smo ranije konstruisali μ-rekurzivne funkcije koje predstavljaju njihove karakteristične funkcije. Za neke druge možemo osmisliti intuitivne postupke koji bi predstavljali odgovarajuće procedure odlučivanja. Na osnovu Čerčove teze, odgovarajuće karakteristične funkcije su i formalno izračunljive.

Napomena 4.2. Složenije relacije možemo konstruisati od postojećih primenom logičkih veznika. Na primer, ako je ρ relacija, tada je i $\neg\rho$ takođe relacija koju zadovoljavaju tačno one torke koje *ne* zadovoljavaju relaciju ρ. Ovu relaciju nazivamo *komplementarnom relacijom* (ili *komplementarnim problemom*) relacije (problema) ρ. Slično, ako su ρ_1 i ρ_2 relacije arnosti k, tada su to i $\rho_1 \wedge \rho_2$ i $\rho_1 \vee \rho_2$, uz značenja koja odgovaraju uobičajenoj semantici konjunkcije i disjunkcije. Najzad, ako je $\rho(y, x_1, \ldots, x_k)$ relacija arnosti $k+1$, tada su $\exists y.\rho(y, x_1, \ldots, x_k)$ i $\forall y.\rho(y, x_1, \ldots, x_k)$ relacije arnosti k, opet sa značenjem koje odgovara standardnoj semantici koju kvantifikatori imaju u logici.

Teorema 4.1. *Ako je relacija ρ odlučiva, tada je i $\neg\rho$ odlučiva. Ako su ρ_1 i ρ_2 odlučive, tada su i $\rho_1 \wedge \rho_2$ i $\rho_1 \vee \rho_2$ odlučive.*

Dokaz. Imamo da je $\chi_{\neg\rho}(x_1, \ldots, x_k) = 1 - \chi_\rho(x_1, \ldots, x_k)$ pa je $\neg\rho$ odlučiva. Slično, imamo da je $\chi_{\rho_1 \wedge \rho_2}(x_1, \ldots, x_k) = \chi_{\rho_1}(x_1, \ldots, x_k) \cdot \chi_{\rho_2}(x_1, \ldots, x_k)$, pa je i $\rho_1 \wedge \rho_2$ odlučiva. Najzad, važi $\chi_{\rho_1 \vee \rho_2}(x_1, \ldots, x_k) = max(\chi_{\rho_1}(x_1, \ldots, x_k), \chi_{\rho_2}(x_1, \ldots, x_k))$, pa je i $\rho_1 \vee \rho_2$ odlučiva. □

Napomena 4.3. Primetimo da iz odlučivosti relacije $\neg\rho$ sledi odlučivost relacije ρ (dakle, važi i obrnuta implikacija). Zaista, važi da je $\rho = \neg\neg\rho$, odakle zaključak sledi iz prethodne teoreme.

Napomena 4.4. Primetimo da ako je $\rho(y, x_1, \ldots, x_k)$ odlučiva relacija, tada $\exists y.\rho(y, x_1, \ldots, x_k)$ i $\forall y.\rho(y, x_1, \ldots, x_k)$ ne moraju biti odlučive relacije. Primer ćemo videti kasnije.

Primer 4.3. Razmotrimo sledeći problem:

- Da li se URM program P_x zaustavlja za ulaz y?

Karakteristična funkcija ovog problema je:

$$f(x,y) \quad = \quad \begin{cases} 1, & \text{ako } \phi_x(y) \neq - \\ 0, & \text{u suprotnom} \end{cases}$$

Ako bi ova funkcija bila izračunljiva, tada bi bila izračunljiva i sledeća funkcija:

$$g(x) \quad = \quad \begin{cases} -, & \text{ako } \phi_x(x) \neq - \\ 0, & \text{u suprotnom} \end{cases}$$

Zaista, $g(x) = cond(eq(f(x,x),1), undef(x), 0)$. Intuitivno, $g(x)$ možemo izračunati tako što najpre izračunamo $f(x,x)$, a zatim, ako je $f(x,x) = 1$, uđemo u beskonačnu petlju. Međutim, ovo znači da postoji neko n takvo da je $g = \phi_n$. Sada je $g(n) = \phi_n(n)$ pa imamo da je $\phi_n(n) = - \Leftrightarrow \phi_n(n) \neq -$. Iz ove kontradikcije sledi da funkcija f nije izračunljiva.

Napomena 4.5. Problem iz prethodnog primera poznat je i kao *problem zaustavljanja* (engl. *halting problem*). On nam govori da postoje fundamentalni problemi vezani za teoriju izračunljivosti („da li se dati program zaustavlja za dati ulaz?") koji nisu odlučivi. Neodlučivost problema zaustavljanja dokazao je Alan Tjuring i tako postavio formalne granice izračunljivosti još i pre nego što su elektronski računari zvanično nastali.

Napomena 4.6. Možemo razmatrati i specijalne slučajeve problema zaustavljanja, poput:

- Da li program P_x staje za ulaz x?
- Da li program P_x staje za fiksirani ulaz c?

Dakle, za razliku od opšteg problema zaustavljanja gde na ulazu imamo dve vrednosti – kôd programa x i ulaz programa y, ovde imamo samo kôd programa x na ulazu. U prvom slučaju se pitamo da li program P_x staje kada mu je na ulazu njegov sopstveni kôd, dok u drugom slučaju imamo unapred fiksirani broj c i pitamo se da li program sa kôdom x staje za taj ulaz. Ponekad specijalni slučaj nekog problema može biti odlučiv, čak i kada opšti problem nije. Ipak, u ovom slučaju se može pokazati da su i ova dva specijalna slučaja problema zaustavljanja takođe neodlučiva. Dokazi se izvode vrlo slično kao i kod opšteg problema zaustavljanja, pa to ostavljamo čitaocu za vežbu.

Primer 4.4. Neka relacija $\rho(x, y, z)$ ima značenje: „program P_x se za ulaz y zaustavlja u najviše z koraka". Ovaj problem je odlučiv. Intuitivno, treba dekodirati kôd x i dobiti program P_x, a zatim izvršiti najviše z njegovih koraka, kako bi se utvrdilo da li se program zaustavlja u tom broju koraka. Sa druge strane, relacija $\exists z.\rho(x, y, z)$ odgovara problemu zaustavljanja („da li postoji z takvo da se program P_x zaustavlja za ulaz y u najviše z koraka?"). Dakle, kao što smo ranije napomenuli, relacija $\exists z.\rho(x, y, z)$ ne mora biti odlučiva čak

i ako $\rho(x, y, z)$ to jeste. Dalje, relacija $\neg\rho(x, y, z)$ je takođe odlučiva, a relacija $\forall y.\neg\rho(x, y, z)$ to nije. Zaista, ako bi ova relacija bila odlučiva, onda bi to bila i relacija $\neg\forall y.\neg\rho(x, y, z)$, koja je ekvivalentna sa $\exists z.\rho(x, y, z)$, za koju smo dokazali da nije odlučiva. Dakle, univerzalni kvantifikator takođe može od odlučive relacije kreirati neodlučivu.

Primer 4.5. Za razliku od obične („neograničene") kvantifikacije, *ograničena* kvantifikacija $\exists y < z.\ \rho(y, x_1, \ldots, x_k)$ i $\forall y < z.\ \rho(y, x_1, \ldots, x_k)$ od odlučivih relacija kreira nove odlučive relacije. Naime, neka je $\chi_\rho(y, x_1, \ldots, x_k)$ karakteristična funkcija relacije ρ. Ako je ona izračunljiva, tada možemo posmatrati funkciju:

$$\begin{array}{lcl} f(0, x_1, \ldots, x_k) & = & 0 \\ f(z+1, x_1, \ldots, x_k) & = & max(\chi_\rho(z+1, x_1, \ldots, x_k), f(z, x_1, \ldots, x_k)) \end{array}$$

Funkcija f je dobijena primitivnom rekurzijom od χ_ρ, a predstavlja karakterističnu funkciju relacije $\exists y < z.\rho(y, x_1, \ldots, x_k)$. Slično, funkcija:

$$\begin{array}{lcl} g(0, x_1, \ldots, x_k) & = & 1 \\ g(z+1, x_1, \ldots, x_k) & = & \chi_\rho(z+1, x_1, \ldots, x_k) \cdot f(z, x_1, \ldots, x_k) \end{array}$$

predstavlja karakterističnu funkciju relacije $\forall y < z.\rho(y, x_1, \ldots, x_k)$. Odavde sledi odlučivost relacija dobijenih ograničenom kvantifikacijom.

Primer 4.6. Posmatrajmo sledeći problem:

- Za dati prirodan broj x, da li je funkcija ϕ_x totalna?

Karakteristična funkcija odgovarajuće relacije je:

$$f(x) = \begin{cases} 1, & \text{ako je } \phi_x \text{ totalna} \\ 0, & \text{u suprotnom} \end{cases}$$

Ako je f izračunljiva, onda je izračunljiva i funkcija:

$$g(x) = \begin{cases} \phi_x(x) + 1, & \text{ako je } \phi_x \text{ totalna} \\ 0, & \text{u suprotnom} \end{cases}$$

Funkcija g je očigledno totalna, a njeno izračunavanje se svodi na određivanje da li je ϕ_x totalna (izračunavanjem funkcije f) i ako jeste, izračunavanje vrednosti $\phi_x(x)+1$. Kako je g izračunljiva, važi da je $g = \phi_n$ za neko n. Kako je g totalna, važi $g(n) = \phi_n(n) + 1$. Dakle, imamo daje $\phi_n(n) = \phi_n(n) + 1$, što nije moguće. Iz ove kontradikcije sledi da f nije izračunljiva, pa dati problem nije odlučiv. Dakle, ne možemo za proizvoljan dati program odlučiti da li se zaustavlja za svaki ulaz.

Teorema 4.2 (Rajsova teorema)**.** *Neka je $\mathcal{F}$ proizvoljni pravi neprazni potskup skupa svih izračunljivih funkcija arnosti* 1*. Tada je sledeći problem neodlučiv:*

- *Za dato x, da li funkcija ϕ_x pripada $\mathcal{F}$?*

Dokaz. Iz činjenice da je $\mathcal{F}$ pravi neprazni podskup skupa svih izračunljivih funkcija arnosti 1 sledi da postoje dve izračunljive funkcije ϕ_p i ϕ_q takve da je $\phi_p \in \mathcal{F}$ i $\phi_q \notin \mathcal{F}$.

Karakteristična funkcija ovog problema je:

$$f(x) = \begin{cases} 1, & \text{ako } \phi_x \in \mathcal{F} \\ 0, & \text{u suprotnom} \end{cases}$$

Ako bi ova funkcija bila izračunljiva, tada bi to bila i funkcija:

$$g(x) = \begin{cases} q, & \text{ako } \phi_x \in \mathcal{F} \\ p, & \text{u suprotnom} \end{cases}$$

Funkcija g je totalna, pa na osnovu teoreme o fiksnoj tački postoji neko n takvo da je $\phi_{g(n)} = \phi_n$. Postavlja se pitanje da li je $\phi_n \in \mathcal{F}$?

- ako je $\phi_n \in \mathcal{F}$, tada je $g(n) = q$, pa imamo da je $\phi_n = \phi_{g(n)} = \phi_q \notin \mathcal{F}$. Kontradikcija.
- ako je $\phi_n \notin \mathcal{F}$, tada je $g(n) = p$, pa imamo da je $\phi_n = \phi_{g(n)} = \phi_p \in \mathcal{F}$. Opet kontradikcija.

Dakle, u oba slučaja imamo kontradikciju. Otuda sledi da je polazna pretpostavka pogrešna, tj. funkcija f nije izračunljiva. □

Napomena 4.7. Rajsova teorema je jedna od najznačajnijih ,,negativnih" teorema teorijskog računarstva. Ona tvrdi da ni za jedno ne-trivijalno *semantičko svojstvo* ne postoji efektivan način da se proveri da li dati program ima to svojstvo ili ne. Pod semantičkim svojstvom podrazumevamo nešto što se odnosi na ponašanje programa u fazi izvršenja (tj. odnosi se na funkciju koju program izračunava). Nasuprot tome, *sintaksna svojstva* programa tiču se strukture samog programa. Primeri sintaksnih svojstava bili bi:

- Da li program sadrži *while* petlju?
- Da li program ima više od 1000 linija koda?
- Da li program koristi rekurziju?
- itd.

Problemi ispitivanja ovakvih sintaksnih svojstava programa su odlučivi (npr. treba proći kroz program i ispitati da li sadrži ključnu reč *while*). Sa druge strane, primeri semantičkih svojstava bili bi:

- Da li se program zaustavlja za dati ulaz c?
- Da li se program uvek zaustavlja?
- Da li program izračunava funkciju $f(x) = x \cdot x$?
- Da li za svako x i y program izračunava vrednost koja se nalazi u intervalu $[x, y]$?
- Da li program za neki ulaz može prijaviti *grešku pristupa memoriji* (engl. *segmentation fault*)?
- itd.

Prema Rajsovoj teoremi, svi ovi problemi su neodlučivi. Na primer, da bismo dokazali da ne možemo algoritamski utvrditi da li dati program izračunava vrednost $x \cdot x$, možemo posmatrati familiju funkcija $\mathcal{F} = \{f\}$, pri čemu je $f(x) = x \cdot x$ (dakle, familija koja sadrži samo tu jednu funkciju). Na osnovu Rajsove teoreme, problem $\phi_x \in \mathcal{F}$ je neodlučiv, što znači da nije moguće za proizvoljan program ispitati da li izračunava funkciju koja pripada $\mathcal{F}$, tj. da li izračunava funkciju f.

Uopšte, Rajsova teorema isključuje mogućnost da možemo da efektivno proverimo da li je implementacija nekog programa u skladu sa *specifikacijom*, odnosno da ispunjava zahteve u vezi onoga šta bi *trebalo* da radi. Na primer, nije moguće automatizovati pregledanje studentskih radova tako da se egzaktno utvrdi da li studentski programi rade ono što se traži u zadatku (moguće je testirati na ograničenom skupu ulaza, tzv. *test primera*, ali to ne garantuje potpunu korektnost).

Primer 4.7. Neka je dat program P_e. Problem:

- Da li je za dato x program P_x ekvivalentan sa P_e, tj. da li je $\phi_x = \phi_e$?

Prema Rajsovoj teoremi, ovaj problem je neodlučiv. Za to je dovoljno da posmatramo familiju $\mathcal{F} = \{\phi_e\}$. Sada je $\phi_x \in \mathcal{F}$ akko $\phi_x = \phi_e$.

Napomena 4.8. Intuicija iza prethodnog primera je sledeća: ako nastavnik napravi *referentnu implementaciju* P_e koja predstavlja rešenje ispitnog zadatka, ne postoji način da automatizuje proveru da li su studentski radovi ekvivalentni sa tom referentnom implementacijom.

Primer 4.8. Za date brojeve x i y, problem:

- da li je $\phi_x = \phi_y$?

je neodlučiv. Dakle, nije moguće algoritamski ispitati ekvivalentnost dva programa. Neodlučivost ovog problema sledi iz prethodnog primera, zato što je problem „$\phi_x = \phi_e$?" (za unapred zadato fiksirano e) specijalni slučaj ovog problema (ako je specijalni slučaj neodlučiv, onda je opšti problem tim pre neodlučiv).

4.2 Rekurzivni skupovi

Problem odlučivanja možemo identifikovati sa skupom njegovih instanci za koje je odgovor potvrdan. Na primer, problem „Da li je broj paran?" se može identifikovati sa skupom parnih brojeva, dok se problem „Da li je dati graf povezan?" identifikuje sa skupom svih povezanih grafova. Ovo je u skladu sa uobičajenom definicijom pojma relacije u matematici – relacija ρ arnosti k nad $\mathbb{N}$ je proizvoljni *podskup* $\rho \subseteq \mathbb{N}^k$. Dakle, kada razmatramo da li neka k-torka zadovoljava neku relaciju, mi zapravo ispitujemo da li ona pripada nekom skupu k-torki koje ta relacija sadrži.

Važi i obratno: svaki skup $A \subseteq \mathbb{N}^k$ određuje jednu relaciju, tj. problem odlučivanja:

- Da li važi $(x_1, \ldots, x_k) \in A$?

S tim u vezi, imamo sledeću definiciju.

Definicija 4.3. Za skup $A \subseteq \mathbb{N}^k$ kažemo da je *rekurzivan* ako je relacija $(x_1, \ldots, x_k) \in A$ odlučiva, tj. ako je karakteristična funkcija skupa A:

$$\chi_A(x_1, \ldots, x_k) \quad = \quad \begin{cases} 1, & \text{ako } (x_1, \ldots, x_k) \in A \\ 0, & \text{u suprotnom} \end{cases}$$

izračunljiva.

Napomena 4.9. Intuitivno, skup je rekurzivan ako postoji efektivan metod (algoritam) kojim možemo da ispitamo da li proizvoljni element pripada tom skupu ili ne. Na primer, skup parnih brojeva je rekurzivan podskup skupa prirodnih brojeva, jer možemo efektivno proveriti da li neki prirodan broj pripada tom skupu ili ne.

Napomena 4.10. Većina ranije navedenih teorijskih rezultata se mogu formulisati u terminologiji rekurzivnih skupova. Na primer, možemo posmatrati skup $D = \{x \in \mathbb{N} \mid \phi_x(x) \neq -\}$ svih kôdova izračunljivih funkcija takvih da je $\phi_x(x)$ definisano. Ovaj skup nije rekurzivan, zato što problem „$\phi_x(x) \neq -$" nije odlučiv.

Napomena 4.11. Svaki konačan skup $A \subseteq \mathbb{N}^k$ je rekurzivan. Zaista, ako je $A = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ (gde je $\mathbf{x}_i = (x_{i1}, \ldots, x_{ik})$), tada je $\mathbf{x} \in A$ ekvivalentno sa $\mathbf{x} = \mathbf{x}_1 \vee \mathbf{x} = \mathbf{x}_2 \vee \ldots \vee \mathbf{x} = \mathbf{x}_n$. Jasno je da je vrednost ovog izraza izračunljiva, pa je skup rekurzivan.

4.3 Poluodlučivost

Definicija 4.4. *Polukarakteristična funkcija* relacije ρ arnosti k nad $\mathbb{N}$ je funkcija:

$$\overline{\chi}_\rho(x_1, \ldots, x_k) \quad = \quad \begin{cases} 1, & \text{ako } \rho(x_1, \ldots, x_k) \\ -, & \text{u suprotnom} \end{cases}$$

Za relaciju ρ kažemo da je *poluodlučiva* ako je njena polukarakteristična funkcija izračunljiva.

Napomena 4.12. Intuitivno, da bi neki problem bio poluodlučiv, potrebno je postoji efektivan postupak (algoritam) koji za sve instance za koje je odgovor „da" to potvrđuje u konačnom broju koraka, dok se za instance za koje je odgovor „ne" ne zaustavlja. Ovakav postupak se naziva i *procedura poluodlučivanja.*

Napomena 4.13. Jasno je da ako je relacija ρ odlučiva, ona je i poluodlučiva. Naime, ako je karakteristična funkcija χ_ρ relacije ρ izračunljiva, onda je i polukarakteristična funkcije $\overline{\chi}_\rho$ takođe izračunljiva:

$$\overline{\chi}_\rho((x_1, \ldots, x_k) = cond(eq(\chi_\rho(x_1, \ldots, x_k), 1), 1, undef(0))$$

Obrnuto ne mora da važi, o čemu govore primeri u nastavku.

Primer 4.9. Ranije smo videli da je problem zaustavljanja neodlučiv. Međutim, problem zaustavljanja *jeste* poluodlučiv. Intuitivno, njegovu polukarakterističnu funkciju:

$$h(x, y) \quad = \quad \begin{cases} 1, & \text{ako } \phi_x(y) \neq - \\ -, & \text{u suprotnom} \end{cases}$$

možemo izračunati na sledeći način:

$$h(x, y) = C_1^1(U^{(1)}(x, y))$$

Ova funkcija će biti definisana i imati vrednost 1 akko se program P_x zaustavlja za ulaz y.

Primer 4.10. Neka je $\rho(y, x_1, \ldots, x_k)$ odlučiva relacija arnosti $k + 1$. Ranije smo videli da relacija $\exists y.\rho(y, x_1, \ldots, x_k)$ ne mora biti odlučiva. Ipak, ova relacija će svakako biti poluodlučiva. Naime, ako je χ_ρ karakteristična funkcija funkcije ρ (koja je izračunljiva, po pretpostavci), tada je funkcija $C_1^1(\mu y[\chi_\rho(y, x_1, \ldots, x_k) = 1])$ polukarakteristična funkcija relacije $\exists y.\rho(y, x_1, \ldots, x_k)$. Ova funkcija je takođe izračunljiva, odakle sledi poluodlučivost relacije $\exists y.\rho(y, x_1, \ldots, x_k)$. Intuitivno, u petlji ćemo uvećavati y dok ne pronađemo neku vrednost za koju važi relacija ρ. Ako takvo y ne postoji, petlja će se beskonačno izvršavati.

Primer 4.11. Dokažimo da važi i obrnuto: ako je relacija $\sigma(x_1, \ldots, x_k)$ poluodlučiva, tada postoji neka odlučiva relacija $\rho(y, x_1, \ldots, x_k)$ takva da je $\sigma(x_1, \ldots, x_k) \Leftrightarrow \exists y.\rho(y, x_1, \ldots, x_k)$. Naime, neka je P program koji izračunava polukarakterističnu funkciju relacije σ. Neka relacija $\rho(y, x_1, \ldots, x_k)$ ima značenje „program P za ulaz $x_1, \ldots, x_k$ se zaustavlja u najviše y koraka". Intuitivno je jasno da je ova relacija odlučiva. Pritom, relacija $\sigma(x_1, \ldots, x_k)$ je ekvivalentna sa $\exists y.\rho(y, x_1, \ldots, x_k)$, s obzirom da se program P zaustavlja u konačno mnogo koraka ako i samo ako važi $\sigma(x_1, \ldots, x_k)$.

Primer 4.12. Može se pokazati da je dovoljno da relacija $\rho(y, x_1, \ldots, x_k)$ bude *poluodlučiva*, relacija $\exists y.\rho(y, x_1, \ldots, x_k)$ će takođe biti poluodlučiva. Naime, ako je $\rho(y, x_1, \ldots, x_k)$ poluodlučiva, tada prema prethodnom primeru postoji neka odlučiva relacija $\rho'(t, y, x_1, \ldots, x_k))$ takva da je $\rho(y, x_1, \ldots, x_k) \Leftrightarrow \exists t.\rho'(t, y, x_1, \ldots, x_k)$. Sada je relacija $\exists y.\rho(y, x_1, \ldots, x_k)$ ekvivalentna sa $\exists y.\exists t.\rho'(t, y, x_1, \ldots, x_k)$. Posmatrajmo sada funkciju:

$$f(x_1, \ldots, x_k) = C_1^1(\mu u[\chi_{\rho'}(\pi_1(u), \pi_2(u), x_1, \ldots, x_k) = 1])$$

pri čemu je $\chi_{\rho'}$ karakteristična funkcija relacije ρ' (ova funkcija je izračunljiva, jer je ρ' odlučiva), a π_1 i π_2 su funkcije dekodiranja uređenih parova (odeljak 3.3.1). Ova funkcija je definisana samo za one $x_1, \ldots, x_k$ za koje postoji neko u, pa samim tim i neko $t = \pi_1(u)$ i $y = \pi_2(u)$ za koje je $\rho'(t, y, x_1, \ldots, x_k)$, tj. ako postoji y takvo da je $\rho(y, x_1, \ldots, x_k)$. Za te vrednosti, funkcija f ima vrednost 1. Otuda, f je upravo polukarakteristična funkcija relacije $\exists y.\rho(y, x_1, \ldots, x_k)$, pa je ona poluodlučiva.

Intuitivno, da bismo ispitali da li postoji y takvo da važi $\rho(y, x_1, \ldots, x_k)$, možemo da paralelno pokrenemo proceduru poluodlučivanja P za relaciju ρ za svako $y \in \mathbb{N}$. Ako postoji takvo y, pre ili kasnije će se odgovarajuće izvršavanje procedure P završiti, pa će odgovor biti potvrdan. Paralelno izvršavanje programa P za sve vrednosti y možemo simulirati sekvencijalno, tako što enumerišemo parove $(\pi_1(u), \pi_2(u))$, tj. $(0, 0), (1, 0), (0, 1), (2, 0), (1, 1), (0, 2), \ldots$, pri čemu par (i, j) označava izvršavanje j-tog koraka procedure P za ulaz $y = i$. Na ovaj način će se za svako $y \in \mathbb{N}$ svi koraci procedure P izvršiti pre ili kasnije.

Teorema 4.3 (Postova teorema)**.** *Ako su relacije ρ i $\neg\rho$ poluodlučive, onda je ρ odlučiva.*

Dokaz. Neka su P_a i P_b programi koji izračunavaju polukarakterističke funkcije relacija ρ i $\neg\rho$, respektivno. Ako istovremeno pokrenemo oba ova programa za isti ulaz $(x_1, \ldots, x_k)$, pre ili kasnije će se zaustaviti jedan od njih, jer svakako važi ili $\rho(x_1, \ldots, x_k)$ ili $\neg\rho(x_1, \ldots, x_k)$. Na osnovu toga koji se od programa zaustavi odlučićemo da li jeste ili nije $\rho(x_1, \ldots, x_k)$. Formalno, neka je $T(e, x_1, \ldots, x_k, z)$ relacija sa značenjem ,,program P_e se za ulaz $x_1, \ldots, x_k$ zaustavlja u najviše z koraka". Kao što znamo, ova relacija je odlučiva. Funkcija:

$$h(x_1, \ldots, x_k) \quad = \quad \mu u \left[\begin{array}{l} (mod(u,2) = 1 \wedge T(a, x_1, \ldots, x_k, div(u,2))) \vee \\ (mod(u,2) = 0 \wedge T(b, x_1, \ldots, x_k, div(u,2))) \end{array} \right]$$

je totalna i vraća najmanje $u = 2 \cdot z + k$, takvo da je z broj koraka potreban da se zaustavi jedan od programa P_a ili P_b, a $k = 1$ ako se zaustavio P_a, odnosno $k = 0$ u suprotnom. Sada će karakteristična funkcija relacije ρ biti:

$$\chi_\rho(x_1, \ldots, x_k) = mod(h(x_1, \ldots, x_k), 2)$$

Intuitivno, programi P_a i P_b će naizmenično izvršavati svoje korake, čime se simulira njihovo istovremeno izvršavanje. Na kraju se utvrđuje koji od njih je završio svoj rad, čime se odlučuje o relaciji ρ. □

Primer 4.13. Problem ,,$\phi_x(y) = -$?" (tj. problem koji pita ,,Da li se program P_x *ne zaustavlja* za ulaz y ?") nije poluodlučiv. Naime, kako znamo da je problem ,,$\phi_x(y) \neq -$?" (problem zaustavljanja) poluodlučiv, iz poluodlučivosti njegovog komplementarnog problema ,,$\phi_x(y) = -$?" bi, prema Postovoj teoremi, sledela odlučivost halting problema, što znamo da nije tačno. Otuda problem ,,$\phi_x(y) = -$?" nije poluodlučiv.

Primer 4.14. Neka je $\rho(y, x_1, \ldots, x_k)$ odlučiva relacija arnosti $k + 1$. Znamo da relacija $\forall y.\rho(y, x_1, \ldots, x_k)$ ne mora biti odlučiva. Pitanje je da li mora biti poluodlučiva? Ova relacija je ekvivalentna sa $\neg\exists y.\neg\rho(y, x_1, \ldots, x_k)$. Relacija $\neg\rho(y, x_1, \ldots, x_k)$ je takođe odlučiva, pa je relacija $\exists y.\neg\rho(y, x_1, \ldots, x_k)$ poluodlučiva, prema primeru 4.10. Ako bi i relacija $\neg\exists y.\neg\rho(y, x_1, \ldots, x_k)$ bila poluodlučiva, tada bi relacija $\exists y.\neg\rho(y, x_1, \ldots, x_k)$ morala biti odlučiva, prema Postovoj teoremi. Ipak, znamo da to ne mora biti slučaj. Otuda, relacija $\forall y.\rho(y, x_1, \ldots, x_k)$ ne mora biti ni poluodlučiva u opštem slučaju.

Primer 4.15. Razmotrimo problem ,,Da li je ϕ_x totalna?". Ovaj problem nije odlučiv (primer 4.6). Ispitajmo da li je ovaj problem poluodlučiv. Posmatrajmo sledeću funkciju:

$$f(x, y) \quad = \quad \left\{ \begin{array}{ll} 1, & \text{ako } P_x \text{ za ulaz x radi duže od } y \text{ koraka} \\ -, & \text{u suprotnom} \end{array} \right.$$

Ova funkcija je izračunljiva. Intuitivno, potrebno je dekodirati program P_x, a zatim za ulaz x simulirati najviše prvih y koraka njegovog rada. Ako se ne završi u tom broju koraka, vraćamo 1. U suprotnom, ulazimo u beskonačnu petlju.

Kako je f izračunljiva, prema *s-m-n* teoremi imamo da postoji totalna izračunljiva funkcija s arnosti 1 takva da važi:

$$f(x, y) = \phi_{s(x)}(y)$$

za svako $x, y \in \mathbb{N}$. Za svako fiksirano x, razmotrimo sada sledeća dva slučaja:

- P_x se ne zaustavlja za ulaz x. Tada će funkcija $f(x, y)$ za to x imati vrednost 1 za svako y, pa će funkcija $\phi_{s(x)}$ biti totalna.
- P_x se zaustavlja za ulaz x u y_0 koraka. Tada će funkcija $f(x, y)$ za to x imati vrednost 1 za $y < y_0$, a biće nedefinisana za $y \geq y_0$. Drugim rečina, $\phi_{s(x)}$ neće biti totalna.

Dakle, važi da je $\phi_{s(x)}$ totalna ako i samo ako $\phi_x(x) = -$. Sada bi iz poluodlučivosti problema totalnosti sledila i poluodlučivost problema ,,$\phi_x(x) = -$?". Međutim, za taj problem znamo da nije poluodlučiv (primer 4.13). Odavde sledi da problem totalnosti funkcije nije ni poluodlučiv.

Primer 4.16. Može se pokazati da ni komplementarni problem problema totalnosti (tj. problem ,,Da li funkcije ϕ_x *nije* totalna?") nije poluodlučiv. Za to je dovoljno posmatrati funkciju:

$$f(x,y) \quad = \quad \begin{cases} 1, & \text{ako } y \neq 0 \\ \phi_x(x), & \text{ako } y = 0 \end{cases}$$

Ova funkcija je izračunljiva, pa po istom principu imamo totalnu funkciju s arnosti 1 takvu da je:

$$f(x, y) = \phi_{s(x)}(y)$$

na osnovu *s-m-n* teoreme. Sada $\phi_{s(x)}$ *nije* totalna akko je $\phi_x(x) = -$, za šta znamo da nije ni poluodlučivo.

Napomena 4.14. Prethodna dva primera nam pokazuju da postoji problem takav da ni on ni njegov komplement nisu poluodlučivi. Ovakvi problemi su u izvesnom smislu najteži, jer nije moguće efektivno odrediti ni da li jeste ni da li nije ispunjeno dato svojstvo.

4.4 Polurekurzivni skupovi

Definicija 4.5. Skup $A \subseteq \mathbb{N}^k$ je *polurekurzivan* (ili *rekurzivno nabrojiv*) ako je njegova polukarakteristična funkcija:

$$\overline{\chi}_A(x_1, \ldots, x_k) \quad = \quad \begin{cases} 1, & \text{ako } (x_1, \ldots, x_k) \in A \\ -, & \text{u suprotnom} \end{cases}$$

izračunljiva.

Napomena 4.15. Iz prethodne definicije je jasno da je skup A polurekurzivan akko je problem ,,$(x_1, \ldots, x_k) \in A$?" poluodlučiv. Otuda se svi rezultati iz prethodnog poglavlja mogu primeniti kada se razmatra polurekurzivnost. Na primer, iz Postove teoreme sledi da ako su skupovi A i njegov komplement $\overline{A}$ polurekurzivni, tada je A i rekurzivan.

Primer 4.17. Iz prethodne definicije i ranijih primera sledi da je skup $\{x \in \mathbb{N} \mid \phi_x(x) \neq -\}$ polurekurzivan, dok skupovi $\{x \in \mathbb{N} \mid \phi_x(x) = -\}$ i $\{x \in \mathbb{N} \mid \phi_x$ je totalna$\}$ to nisu.

Jasno je da će svaki polurekurzivni skup biti domen neke izračunljive funkcije – na primer, upravo svoje polukarakteristične funkcije. Važi i obratno, o čemu govori sledeća teorema.

Teorema 4.4. *Neka je data proizvoljna izračunljiva funkcija f arnosti k. Tada je njen domen $dom(f)$ polurekurzivan skup.*

Dokaz. Funkcija:

$$g(x_1, \ldots, x_k) = C_1^1(f(x_1, \ldots, x_k))$$

ima vrednost 1 akko je $(x_1, \ldots, x_k) \in dom(f)$, dok za $(x_1, \ldots, x_k) \notin dom(f)$ nije definisana. Otuda je g upravo polukarakteristična funkcija skupa $dom(f)$. Kako je g izračunljiva, $dom(f)$ je polurekurzivan. □

Iz prethodne teoreme sledi da se polurekurzivni skupovi poklapaju sa domenima izračunljivih funkcija. Na primer, u terminologiji URM programa, neki skup A je polurekurzivan akko postoji URM program koji se zaustavlja upravo za ulaze koji su elementi od A, dok se za ostale ulaze ne zaustavlja.

Teorema 4.5. *Neprazan skup $A \subseteq \mathbb{N}$ je polurekurzivan akko postoji totalna izračunljiva funkcija f arnosti 1 takva da je $f(\mathbb{N}) = A$ (tj. slika skupa $\mathbb{N}$ je ceo skup A).*

Dokaz. Pretpostavimo da postoji takva funkcija f. Sada se polukarakteristična funkcija skupa A može izraziti kao:

$$\overline{\chi}_A(x) = C_1^1(\mu y[f(y) = x])$$

Dakle, $\overline{\chi}_A$ je izračunljiva, pa je A polurekurzivan. Intuitivno, procedura poluodlučivanja se sastoji u nabrajanju elemenata skupa A funkcijom f dok se ne naiđe na element x.

Obrnuto, pretpostavimo da je A neprazan i polurekurzivan. Neka je P program koji izračunava polukarakterističnu funkciju skupa A i neka je a neki fiksirani element od A. Sada možemo posmatrati sledeću funkciju:

$$g(x, y) = \begin{cases} x, & \text{ako se } P \text{ zaustavlja u najviše } y \text{ koraka} \\ a, & \text{u suprotnom} \end{cases}$$

Ova funkcija je izračunljiva i totalna. Pritom, važi $g(\mathbb{N}^2) = A$. Naime, za svako $x \in A$, program P će se završiti u konačnom broju koraka, pa će za neko y važiti $g(x, y) = x$. Sada možemo uzeti da je:

$$f(u) = g(\pi_1(u), \pi_2(u))$$

gde su π_1 i π_2 funkcije dekodiranja za uređene parove prirodnih brojeva (odeljak 3.3.1). Zaista, ova funkcija će biti izračunljiva i totalna i važiće $f(\mathbb{N}) = A$. □

Napomena 4.16. Intuitivno, prethodna teorema nam kaže da je skup polurekurzivan akko se njegovi elementi mogu *efektivno* nabrojati, tj. poređati u niz:

$$f(0), f(1), f(2), \ldots$$

Na primer, možemo efektivno nabrojati sve x takve da je $\phi_x(x) \neq -$. Sa druge strane, nije moguće efektivno nabrojati sve x takve da je ϕ_x totalna. Zbog toga se polurekurzivni skupovi često nazivaju i *rekurzivno nabrojivi skupovi*.

Dodatno, sam dokaz teoreme prikazuje zanimljiv intuitivni postupak kako se ova enumeracija može izvršiti. Potrebno je paralelno pokrenuti program P za

sve ulaze x. Kako se koji program završi, taj ulaz dodajemo u niz. Na ovaj način će svi elementi skupa A pre ili kasnije biti dodati u niz. Paralelno izvršavanje programa P za različite ulaze možemo sekvencijalno simulirati na isti način kao u primeru 4.12, enumeracijom parova (x, y), gde par (x, y) podrazumeva izvršavanje y-tog koraka programa P za ulaz x.

4.5 Odlučivost u logici

U ovom poglavlju razmatramo najznačajnije teorijske rezultate kada je u pitanju odlučivost u iskaznoj logici i logici prvog reda. Svi rezultati biće predstavljeni u terminima intuitivne izračunljivosti, zbog čitljivosti. Ipak, treba imati u vidu da se formule, kao hijerarhijske strukture, uvek mogu kodirati brojevima. Isto važi i za sve konačne skupove formula. Samim tim, svi problemi odlučivanja i odgovarajuće (polu)karakteristične funkcije se mogu predstaviti i u formalnom obliku, kao parcijalne funkcije. Njihovu formalna izračunljivost će biti posledica Čerčove teze.

Teorema 4.6. *Problem zadovoljivosti iskazne formule je odlučiv.*

Dokaz. Odlučivost proističe iz konačnosti skupa valuacija koje treba razmotriti. Jedna moguća procedura odlučivanja bila bi metod istinitosnih tablica, kao što je razmatrano u glavi 2. □

Posledica 4.1. *Problem tautologičnosti iskazne formule je odlučiv.*

Dokaz. Problem tautologičnosti formule A je ekvivalentan problemu *nezadovoljivosti* formule $\neg A$, što je komplementaran problem problemu *zadovoljivosti* formule $\neg A$. Kako je komplementaran problem odlučivog problema odlučiv, sledi da je i problem tautologičnosti formule odlučiv. □

Napomena 4.17. Isti metod istinitosnih tablica se može primeniti i za odlučivanje o tautologičnosti iskazne formule. Ipak, u praksi se obično tautologičnost svodi na ispitivanje zadovoljivosti negacije, s obzirom da su za problem zadovoljivosti razvijeni efikasniji algoritmi.

Posledica 4.2. *SAT problem je odlučiv.*

Dokaz. SAT problem je specijalni slučaj zadovoljivosti iskaznih formula u KNF-u, pa je odlučiv. □

Napomena 4.18. U praksi se obično opšti problem zadovoljivosti iskazne formule (pa i problem tautologičnosti) svodi na SAT problem, transformacijom polazne formule (ili njene negacije, u slučaju tautologičnosti) na KNF formulu. Ovo je zbog toga što najefikasniji postojeći algoritmi za ispitivanje zadovoljivosti rade nad formulama u KNF-u.

Posledica 4.3. *Zadovoljivost konačnog skupa iskaznih formula je odlučiv.*

Dokaz. Neka je $\Delta = \{A_1, \ldots, A_n\}$ proizvoljan konačan skup iskaznih formula. Skup Δ je zadovoljiv ako postoji valuacija v koja istovremeno zadovoljava sve njegove formule. Kako je Δ konačan skup, moguće je konstruisati konjunkciju svih njegovih formula, tj. formulu $A_1 \wedge \ldots \wedge A_n$. Sada valuacija v zadovoljava skup Δ akko zadovoljava ovu konjunkciju. Otuda se ispitivanje zadovoljivosti

skupa Δ svodi na ispitivanje zadovoljivosti jedne formule, što je odlučivo na osnovu teoreme 4.6. □

Posledica 4.4. *Ako je Δ konačan skup iskaznih formula i A proizvoljna iskazna formula, tada je problem $\Delta \vDash A$ odlučiv.*

Dokaz. Važi $\Delta \vDash A$ akko je skup $\Delta \cup \{\neg A\}$ nezadovoljiv. Kako je $\Delta \cup \{\neg A\}$ konačan skup, ispitivanje njegove (ne)zadovoljivosti je, odlučivo, prema posledici 4.3. □

Situacija je znatno komplikovanija za beskonačne skupove iskaznih formula. O tome govori sledeća teorema, čiji dokaz izostavljamo, zbog složenosti. Pritom, napominjemo da u nastavku ovog poglavlja podrazumevamo da su svi beskonačni skupovi formula koje razmatramo rekurzivni, tj. da je uvek moguće efektivno odrediti da li proizvoljna formula pripada tom skupu ili ne.

Teorema 4.7. *Problem zadovoljivosti beskonačnog skupa iskaznih formula je neodlučiv.*

Posledično, komplementarni problem *nezadovoljivosti* beskonačnog skupa iskaznih formula je takođe neodlučiv. Ispostavlja se da je ovaj problem poluodlučiv, o čemu govori sledeća teorema.

Teorema 4.8. *Problem* nezadovoljivosti *beskonačnog skupa iskaznih formula Δ (tj. pitanje „Da li je dati beskonačni skup iskaznih formula Δ nezadovoljiv?") je poluodlučiv.*

Dokaz. Na osnovu teoreme o kompaktnosti, Δ je nezadovoljiv akko postoji njegov konačan nezadovoljiv potskup. Za svaki konačan potskup Δ' pitanje njegove (ne)zadovoljivosti je odlučivo. Pritom, kako je Δ po pretpostavci rekurzivan, pa samim tim i rekurzivno nabrojiv, moguće je efektivno nabrajati sve njegove elemente, pa samim tim i sve njegove konačne potskupove. Otuda je problem nezadovoljivosti skupa Δ poluodlučiv, jer je dobijen egzistencijalnom kvantifikacijom odlučivog problema (na osnovu primera 4.10). □

Posledica 4.5. *Problem $\Delta \vDash A$, gde je Δ beskonačan skup iskaznih formula, a A je proizvoljna iskazna formula, je poluodlučiv.*

Razmotrimo sada fundamentalne semantičke probleme u logici prvog reda.

Teorema 4.9. *Problem valjanosti formule prvog reda je neodlučiv.*

Dokaz. Dokaz ćemo izvesti tako što ćemo svesti problem zaustavljanja (za koji znamo da je neodlučiv) na problem valjanosti formule prvog reda. Neka je dat proizvoljan URM program $P = I_1, \ldots, I_m$, i neka je n indeks najvećeg registra koji se koristi u instrukcijama programa P.. *Stanje* programa P je dato sekvencom $[k, r_1, r_2, \ldots, r_n]$, gde su $r_1, r_2, \ldots, r_n$ redom vrednosti registara $R_1, R_2, \ldots, R_n$, a k je indeks naredne instrukcije koju treba izvršiti. Bez ograničenja opštosti, možemo pretpostaviti da se program P može završiti jedino tako što indeks naredne instrukcije postane jednak $m+1$, tj. kada dođe u stanje oblika $[m+1, r_1, r_2, \ldots, r_n]$.

Razmatraćemo signaturu $\mathcal{L}$ koja sadrži simbol konstante 0, funkcijski simbol S arnosti 1, predikatski simbol $=$ arnosti 2, kao i predikatski simbol R arnosti $n+1$. Termovi $0, S(0), S(S(0)), \ldots$ predstavljaju *numerale* koje ćemo kraće

označavati sa $\overline{0}, \overline{1}, \overline{2}, \ldots$, a koji intuitivno predstavljaju prirodne brojeve. Simbol $=$ intuitivno predstavlja jednakost nad brojevima. Ovakav smisao ovih simbola se obezbeđuje formulom A, koja predstavlja konjunkciju sledećih formula:

- $\forall x.x = x$
- $\forall xy.x = y \Rightarrow y = x$
- $\forall xyz.x = y \wedge y = z \Rightarrow x = z$
- $\forall xy.x = y \Rightarrow S(x) = S(y)$
- $\forall x_1 \ldots x_n y u_1 \ldots u_n z.x_1 = u_1 \wedge \ldots \wedge x_n = u_n \wedge y = z \Rightarrow (R(x_1, \ldots, x_n, y) \Leftrightarrow R(u_1, \ldots, u_n, z))$
- $\forall x.\neg(0 = S(x))$
- $\forall xy.S(x) = S(y) \Rightarrow x = y$

Formula A obezbeđuje da relacija kojom se interpretira $=$ bude refleksivna, simetrična, tranzitivna i *kongruentna* (saglasna) sa S i R, kao i da se numerali interpretiraju različitim elementima domena.

Takođe, želimo da značenje atoma $R(x_1, \ldots, x_n, y)$ bude „stanje $[y, x_1, \ldots, x_n]$ je dostižno prilikom izvršavanja programa P za ulaz a". Činjenica da je početno stanje $[1, a, 0, 0, \ldots, 0]$ dostižna je opisana atomičkom formulom $R(\overline{a}, \overline{0}, \ldots, \overline{0}, \overline{1})$. Označimo ovu formulu sa T_0.

Dalje, za svaku instrukciju I_k $(i = 1, \ldots, m)$ programa P formiramo formulu T_k na sledeći način:

- ako je $I_k = Z(r)$ $(r \leq n)$, tada je T_k:

$$\forall x_1 \ldots x_n.R(x_1, \ldots, x_r, \ldots, x_n, \overline{k}) \Rightarrow R(x_1, \ldots, 0, \ldots, x_n, S(\overline{k}))$$

- ako je $I_k = S(r)$ $(r \leq n)$, tada je T_k:

$$\forall x_1 \ldots x_n.R(x_1, \ldots, x_r, \ldots, x_n, \overline{k}) \Rightarrow R(x_1, \ldots, S(x_r), \ldots, x_n, S(\overline{k}))$$

- ako je $I_k = T(p, q)$ $(p, q \leq n)$, tada je T_k:

$$\forall x_1 \ldots x_n.R(x_1, \ldots, x_p, \ldots, x_q, \ldots, x_n, \overline{k}) \Rightarrow R(x_1, \ldots, x_p, \ldots, x_p, \ldots, x_n, S(\overline{k}))$$

- ako je $I_k = J(p, q, r)$, $(p, q \leq n)$, tada je T_k:

$$\begin{aligned} \forall x_1 \ldots x_n. \quad & R(x_1, \ldots, x_p, \ldots, x_q, \ldots, x_n, \overline{k}) \Rightarrow \\ & (x_p = x_q \Rightarrow R(x_1, \ldots, x_p, \ldots, x_q, \ldots, x_n, \overline{r})) \wedge \\ & (\neg(x_p = x_q) \Rightarrow R(x_1, \ldots, x_p, \ldots, x_q, \ldots, x_n, S(\overline{k}))) \end{aligned}$$

Intuitivno, ove formule izvode zaključke o dostižnosti stanja programa P na osnovu njegovih instrukcija. Označimo sa $T_{P,a}$ formulu:

$$A \wedge T_0 \wedge T_1 \wedge \ldots \wedge T_m$$

Ova formula opisuje rad programa P za ulaz a, tj. njegova dostižna stanja. Preciznije, neka je $\mathcal{D}$ proizvoljna struktura nad jezikom $\mathcal{L}$ koja zadovoljava formulu

$T_{P,a}$. Tada se, zbog formule A, svi numerali $\overline{0}, \overline{1}, \overline{2}, \ldots$ interpretiraju različitim elementima domena. Bez ograničenja opštosti, možemo pretpostaviti da domen strukture $\mathcal{D}$ sadrži skup $\mathbb{N}$, kao i da se numerali $\overline{0}, \overline{1}, \overline{2}, \ldots$ interpretiraju odgovarajućim prirodnim brojevima $0, 1, 2, \ldots$. Takođe, ako je ρ relacija arnosti $n+1$ kojom se interpretira simbol R, tada za svako dostižno stanje $[y, x_1, \ldots, x_n]$ važi $\rho(x_1, \ldots, x_n, y)$. Ovo se može dokazati indukcijom po rednom broju koraka izvršavanja programa, a na osnovu formula $T_0, T_1, \ldots, T_m$. Primetimo da formula $T_{P,a}$ dopušta da i za neka nedostižna stanja $[y, x_1, \ldots, x_n]$ važi da je $\rho(x_1, \ldots, x_n, y)$. Dakle, formula $T_{P,a}$ definiše *minimalnu* relaciju ρ kojom se može interpretirati simbol R nad fiksiranim domenom. Neka je $\mathcal{D}_0$ jedna takva minimalna struktura koja zadovoljava $T_{P,a}$, tj. struktura u kojoj $\rho(x_1, \ldots, x_n, y)$ važi *samo* za dostižna stanja.

Problem zaustavljanja programa P za ulaz a je sada ekvivalentan sa problemom valjanosti formule:

$$T_{P,a} \Rightarrow (\exists x_1 \ldots x_n . R(x_1, \ldots, x_n, S(m)))$$

Zaista, ako je ova formula valjana, tada i formula $\exists x_1 \ldots x_n . R(x_1, \ldots, x_n, S(m))$ mora biti tačna u svakoj strukturi $\mathcal{D}$ u kojoj je tačna i formula $T_{P,a}$. Specijalno, ona mora biti tačna u strukturi $\mathcal{D}_0$. To znači da postoji dostižno stanje URM programa u kome je indeks naredne instrukcije jednak $m+1$. Dakle, program P se zaustavlja za ulaz a.

Obrnuto, pretpostavimo da se program P zaustavlja za ulaz a. Tada postoji dostižno stanje oblika $[m+1, x_1, \ldots, x_n]$, što znači da u svakoj strukturi $\mathcal{D}$ koja zadovoljava $T_{P,a}$ formula $R(x_1, \ldots, x_n, S(m))$ mora biti tačna za neko $x_1, \ldots, x_n$. Otuda u $\mathcal{D}$ važi $\exists x_1 \ldots x_n . R(x_1, \ldots, x_n, S(m))$. Samim tim, gornja formula je valjana.

Dakle, ako bi problem valjanosti formule prvog reda bio odlučiv, tada bi i problem zaustavljanja bio odlučiv. Zaista, za proizvoljan program P i ulaz a mogli bismo da formiramo gore opisanu formulu i da postojećom procedurom odlučivanja ispitamo njenu valjanost. Ipak, kako znamo od ranije da problem zaustavljanja nije odlučiv, sledi da ni problem valjanosti u logici prvog reda nije odlučiv. □

Teorema 4.10. *Problem* nezadovoljivosti *formule prvog reda (tj. pitanje ,,da li je formula F prvog reda* nezadovoljiva*?") je neodlučiv, ali jeste poluodlučiv.*

Dokaz. Opisani problem je ekvivalentan problemu valjanosti: formula F je nezadovoljiva akko je $\neg F$ valjana. Ako bi problem nezadovoljivosti bio odlučiv, tada bismo mogli ispitati valjanost proizvoljne formule F tako što utvrdimo da li je njena negacija nezadovoljiva. Otuda, problem nezadovoljivosti ne može biti odlučiv.

Sa druge strane, može se pokazati da se ispitivanje (ne)zadovoljivosti formule prvog reda može svesti na problem ispitivanja (ne)zadovoljivosti beskonačnog skupa iskaznih formula (dokazivanje te činjenice izlazi van okvira ovog teksta). Kako je problem nezadovoljivosti beskonačnog skupa iskaznih formula poluodlučiv, sledi da je i problem nezadovoljivosti formule prvog reda poluodlučiv. □

Posledica 4.6. *Problem zadovoljivosti formule prvog reda (tj. pitanje ,,da li je formula F prvog reda* zadovoljiva*?") nije ni poluodlučiv.*

Dokaz. Ako bi problem zadovoljivosti bio poluodlučiv, onda bi, na osnovu Postove teoreme, bio i odlučiv, s obzirom da je njegov komplementarni problem (problem *nezadovoljivosti*) poluodlučiv. Otuda bi i problem valjanosti bio odlučiv (jer je F valjana akko $\neg F$ nije zadovoljiva), što znamo da nije slučaj. Kontradikcija. □

Teorema 4.11. *Problem $\Delta \vDash F$ u logici prvog reda nije odlučiv, ali jeste poluodlučiv.*

Dokaz. Neodlučivost sledi iz neodlučivosti specijalnog slučaja za $\Delta = \varnothing$ (tada je ovaj problem ekvivalentan sa valjanošću formule F).

Sa druge strane, problem $\Delta \vDash F$ je ekvivalentan sa problemom nezadovoljivosti skupa $\Delta \cup \{\neg F\}$. U slučaju konačnog skupa $\Delta = \{A_1, \ldots, A_n\}$, ovo se svodi na nezadovoljivost formule $A_1 \ldots A_n \wedge \neg F$, što je poluodlučivo. U slučaju beskonačnog skupa Δ, na osnovu teoreme kompaktnosti važi da je $\Delta \vDash F$ akko postoji konačan potskup $\Delta' \subseteq \Delta$ takav da $\Delta' \vDash F$. Pritom, iz rekurzivnosti skupa Δ sledi da je moguće efektivno nabrajati sve njegove konačne potskupove. Problem $\Delta' \vDash F$ je poluodlučiv, a na osnovu primera 4.12, egzistencijalnom kvantifikacijom poluodlučivog problema opet dobijamo poluodlučiv problem. Otuda je problem $\Delta \vDash F$ poluodlučiv i za beskonačni skup Δ. □

Glava 5

Deduktivni sistemi

Pod *dedukcijom* podrazumevamo izvođenje zaključaka na osnovu datih *pretpostavki*, korišćenjem jasno definisanih *pravila izvođenja*. U matematičkoj logici, dedukcija predstavlja mehanizam formalnog rezonovanja, tj. dokazivanja tvrđenja. Pored logike, deduktivni sistemi se koriste i u drugim oblastima gde je potrebno formalno i rigorozno izvoditi zaključke o nekom sistemu i njegovim svojstvima. Sa nekim takvim primenama deduktivnih sistema ćemo se upoznati kasnije.

Svaki deduktivni sistem je zadat skupom pravila izvođenja, koja su oblika:

$$\frac{P_1 \quad P_2 \quad \dots \quad P_n}{Q}$$

gde su $P_1, \dots, P_n$ *premise* pravila, dok je Q *zaključak* pravila. Premise i zaključak pravila predstavljaju *tvrđenja* – činjenice koje su predmet izvođenja u deduktivnom sistemu. Smisamo gornjeg pravila izvođenja je: ,,ako mogu izvesti $P_1, \dots, P_n$, tada mogu izvesti i Q".

U kontekstu logike, tvrđenja se najčešće predstavljaju formulama, a izvođenje formula u deduktivnom sistemu predstavlja njihovo *dokazivanje*. U tom smislu, značenje gornjeg pravila izvođenja je: ,,ako mogu da dokažem formule $P_1, \dots, P_n$, tada mogu da dokažem i Q". Na primer, posmatrajmo pravilo izvođenja:

$$\frac{P \Rightarrow Q \quad P}{Q}$$

Ovo pravilo (poznato i kao *modus ponens*) kaže da, ako možemo da dokažemo neku formulu P, kao i formulu $P \Rightarrow Q$, tada možemo da dokažemo i formulu Q.

Pored pravila, deduktivni sistem po pravilu sadrži i skup *aksioma*, tj. tvrđenja za koje pretpostavljamo da važe i da se ne moraju izvoditi. Aksiome se mogu formulisati kao pravila izvođenja sa praznim skupom premisa, npr.:

$$\frac{}{Q \vee \neg Q}$$

Ovo pravilo kaže da mogu da dokažem formulu $Q \vee \neg Q$ bez ikakvih prethodnih pretpostavki, tj. formulu $Q \vee \neg Q$ ne moram ni da dokazujem, već je uvek mogu smatrati već dokazanom.

Primetimo da su aksiome i pravila izvođenja zadata šematski, a njihove konkretne instance se mogu dobiti instanciranjem simbola konkretnim objektima koji čine tvrđenje (u logici su to formule). Na primer, konkretna instanca pravila *modus ponens* može biti:

$$\frac{p \vee q \Rightarrow \neg r \quad p \vee q}{\neg r}$$

pri čemu je P instancirano sa $p \vee q$, a Q sa $\neg r$, gde su p, q, r konkretna iskazna slova. Slično, instanca aksioma $Q \vee \neg Q$ može biti formula $(p \Rightarrow r) \vee \neg(p \Rightarrow r)$.

Pod *izvođenjem* u deduktivnom sistemu podrazumevamo niz tvrđenja $A_1, A_2, \ldots A_n$ takav da je svako A_i u nizu ili instanca aksiome, ili je izvedeno primenom nekog pravila izvođenja koristeći neka tvrđenja koje prethode A_i u nizu kao pretpostavke. Za tvrdjenje A_n kojim se izvođenje završava kažemo da je izvedeno datim izvođenjem.

U kontekstu logike, izvođenja nazivamo i *dokazima*, a tvrđenja za koja postoji dokaz nazivamo *teoremama* datog deduktivnog sistema. Na primer, ako pored aksiome $Q \vee \neg Q$ postoji i aksioma $P \Rightarrow (Q \Rightarrow P)$, uz pravilo izvođenja *modus ponens*, možemo imati sledeći dokaz:

1. $p \vee \neg p$
2. $p \vee \neg p \Rightarrow (q \Rightarrow p \vee \neg p)$
3. $q \Rightarrow p \vee \neg p$
4. $(q \Rightarrow p \vee \neg p) \Rightarrow (p \Rightarrow (q \Rightarrow p \vee \neg p))$
5. $p \Rightarrow (q \Rightarrow p \vee \neg p)$

Prve dve formule u nizu su instance aksioma, dok je treća izvedena iz prve dve pravilom *modus ponens*. Četvrta je ponovo instanca aksiome, dok je peta izvedena iz treće i četvrte primenom pravila *modus ponens*. Otuda je formula $p \Rightarrow (q \Rightarrow p \vee \neg p)$ jedna teorema ovog jednostavnog deduktivnog sistema.

Zbog preglednosti, izvođenja (dokazi) se često umesto niza predstavljaju *stablom*, kako bi se jasno videlo koje je tvrđenje izvedeno iz kojih pretpostavki, kao i koje je pravilo primenjeno. U listovima tog stabla se nalaze aksiome, dok se u unutrašnjim čvorovima nalaze izvedena tvrđenja. Pritom, deca unutrašnjeg čvora odgovaraju pretpostavkama koje su primenjene prilikom izvođenja odgovarajućeg tvrđenja (sam unutrašnji čvor obično sadrži i informaciju o primenjenom pravilu). U korenu stabla se nalazi tvrđenje koje je izvedeno tim izvođenjem. Na primer, gornji dokaz se u obliku stabla može predstaviti na sledeći način:

$$\dfrac{\dfrac{p \vee \neg p \qquad p \vee \neg p \Rightarrow (q \Rightarrow p \vee \neg p)}{q \Rightarrow p \vee \neg p}\, mp \qquad (q \Rightarrow p \vee \neg p) \Rightarrow (p \Rightarrow (q \Rightarrow p \vee \neg p))}{p \Rightarrow (q \Rightarrow p \vee \neg p)}\, mp$$

Činjenicu da se neko tvrđenje P može izvesti u deduktivnom sistemu R zapisivaćemo kao:

$$\vdash_R P$$

pri čemu se indeks R može izostaviti ako je jasno o kom se deduktivnom sistemu radi (tj. možemo pisati samo $\vdash P$). U kontekstu logike, $\vdash P$ značiće da je P teorema tog deduktivnog sistema.

Ponekad ćemo razmatrati izvođenja u kojima se, kao listovi u stablu, pored aksioma mogu nalaziti i dodatne *pretpostavke* iz nekog zadatog skupa tvrđenja Δ. U tom slučaju, kažemo da takvo izvođenje izvodi (dokazuje) neko tvrđenje P *iz pretpostavki* Δ i to zapisujemo kao:

$$\Delta \vdash P$$

Kažemo i da je P *deduktivna posledica* skupa Δ u datom deduktivnom sistemu.

Dedukcija predstavlja formalni, simbolički sistem manipulisanja diskretnim objektima i kao takva je lišena bilo kakvog značenja osim onog koji sama proizvodi. Ipak, svrha dedukcije je obično da formalno opravda neke zaključke o kojima mi imamo nekakvu predstavu u stvarnosti. Drugim rečima, mi deduktivnom sistemu želimo da pridružimo nekakav smisao, tj. neku semantiku. Konkretno, u slučaju logike, očekujemo da deduktivni sistem izvodi formule koje su logički valjane. Za takav deduktivni sistem kažemo da je *saglasan* (engl. *sound*). Saglasnost deduktivnog sistema je neophodan uslov da bi taj deduktivni sistem bio od bilo kakvog praktičnog značaja u formalizaciji logičkog rasuđivanja. Pored saglasnosti, poželjno je da logički deduktivni sistem ima i svojstvo *potpunosti* – tj. da omogućava da se svaka valjana formula može dokazati kao teorema u tom deduktivnom sistemu. Dakle, imamo:

- *saglasnost*: iz $\vdash F$ sledi $\models F$
- *potpunost*: iz $\models F$ sledi $\vdash F$
- *saglasnost* i *potpunost*: $\vdash F$ akko $\models F$

5.1 Prirodna dedukcija

U ovom poglavlju razmatramo logički deduktivni sistem poznat pod nazivom *prirodna dedukcija* (engl. *natural deduction*). Ovaj sistem se može koristiti kako u iskaznoj, tako i u logici prvog reda (a uz određena uopštenja, i u logikama višeg reda). Naziv *prirodna dedukcija* potiče od činjenice da njegova pravila izvođenja prate uobičajene načine izvođenja zaključaka koje matematičari koriste prilikom dokazivanja teorema. Prirodnu dedukciju je razvio nemački matematičar Gerhard Gencen 1933. godine.

Prilikom izvođenja dokaza u prirodnoj dedukciji, moguće je u određenim delovima dokaza privremeno uvesti dodatne pretpostavke koje se u nekom trenutku mogu *osloboditi* primenom odgovarajućih pravila. Oslobođena pretpostavka se na dalje ne može koristiti u narednim koracima dokaza. To znači da tokom izvođenja dokaza moramo da vodimo evidenciju o tome koje su pretpostavke na snazi u svakom koraku. Zbog toga ćemo tvrđenja koja čine dokaz predstavljati u obliku $\Gamma \vdash A$, pri čemu je Γ konačan skup rečenica, sa značenjem ,,Formula A se može dokazati iz pretpostavki Γ" (ili ,,Formulu A treba dokazati iz pretpostavki Γ" u zavisnosti u kom smeru posmatramo dokaz). Zapis $\Gamma \vdash A$ nazivamo *kontekst* ili *sekvent*. Iz konteksta za svaku formulu vidimo koje su nam pretpostavke na raspolaganju za izvođenje dokaza te formule.

U prirodnoj dedukciji, za svaki od logičkih veznika imamo dva pravila: *pravilo uvođenja* (označeno sa I) i *pravilo eliminacije* (označeno sa E). U nastavku razmatramo ova pravila za svaki od veznika.

U slučaju negacije, imamo sledeća pravila:

$$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\,\neg I \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash \neg A}{\Gamma \vdash \bot}\,\neg E$$

Pravilo uvođenja negacije ($\neg I$) kaže da ako se formula $\bot$ može dokazati iz skupa pretpostavki Γ, A (gde je Γ neki konačan skup pretpostavki, možda i prazan), tada je formulu $\neg A$ moguće dokazati iz pretpostavki Γ. Ovim se dokazuje formula $\neg A$, tj. uvodi se veznik $\neg$ u dokazanu formulu.

Primetimo da se ovo pravilo može tumačiti i ,,unazad": da bismo dokazali $\neg A$ iz pretpostavki Γ, dovoljno je dokazati formulu $\bot$ iz pretpostavki Γ, A. Sva pravila prirodne dedukcije ćemo moći ovako dualno da tumačimo. Ova dva tumačenja proističu iz načina na koji posmatramo dokaze. Jedan način je da dokaz (predstavljen u vidu stabla) posmatramo od listova ka korenu: mi polazimo od pretpostavki i pravilima izvodimo nove činjenice. Tada je tumačenje tih pravila ,,ako sam dokazao premise, mogu dokazati i zaključak pravila". Ovaj pogled na dokaz je krajnje intuitivan ako želimo da tumačimo postojeći dokaz. Drugi pogled na dokaz je od korena ka listovima: mi polazimo od tvrđenja koje je potrebno dokazati (tzv. *cilj* dokaza) i primenom pravila ,,unazad" dolazimo do premisa koje je potrebno dokazati kako bi zaključak važio. Time se dokazivanje glavnog cilja svodi na dokazivanje *potciljeva* koji predstavljaju premise za primenu tog pravila. U takvom pogledu na dokaz, pravila se tumače u stilu ,,da bih dokazao cilj, dovoljno je dokazati ove potciljeve". Ovakav pogled na dokaze i pravila je pogodan prilikom konstrukcije dokaza i tipično se koristi u automatskom dokazivanju teorema.

Pravilo eliminacije negacije ($\neg E$) kaže da ako iz pretpostavki Γ dokažemo A i $\neg A$, tada iz Γ možemo dokazati i $\bot$. Ili, inverzno: da bismo dokazali $\bot$ iz Γ, dovoljno je dokazati A i $\neg A$ iz Γ, za neku, proizvoljno odabranu formulu A. Ovim pravilom se negacija koja se nalazi u premisi $\neg A$ eliminiše, tj. ne postoji u zaključku.

Za konjunkciju imamo sledeća pravila:

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\,\wedge I \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\,\wedge E1 \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\,\wedge E2$$

Pravilo uvođenja konjunkcije ($\wedge I$) kaže da ako iz Γ možemo dokazati formule A i B, tada iz Γ možemo dokazati i $A \wedge B$. Inverzno: da bismo dokazali $A \wedge B$ iz Γ, dovoljno je iz istog skupa pretpostavki dokazati A i B. Ovim se dokazivanje cilja $A \wedge B$ svodi na dokazivanje dva potcilja, A i B.

Pravilo eliminacije konjunkcije dolazi u dve varijante: eliminacija po prvom konjunktu ($\wedge E1$) i po drugom konjunktu ($\wedge E2$). U prvom slučaju, pravilo kaže: ako možemo da dokažemo $A \wedge B$ iz pretpostavki Γ, tada je moguće dokazati i formulu A iz Γ. Ili inverzno: da bismo dokazali A iz Γ, dovoljno je dokazati $A \wedge B$ iz istih pretpostavki. Druga varijanta pravila je analogna, samo za drugi konjunkt B.

Za disjunkciju imamo sledeća pravila:

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\,\vee I1 \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\,\vee I2$$

$$\frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}\,\vee E$$

Ovoga puta, pravilo uvođenja disjunkcije dolazi u dva oblika, po prvom disjunktu ($\vee I1$) i po drugom disjunktu ($\vee I2$). Ovo pravilo kaže da ako možemo da iz Γ dokažemo jedan disjunkt (A ili B), tada je iz istog skupa pretpostavki moguće dokazati i disjunkciju $A \vee B$. Ili obrnuto: da bismo dokazali $A \vee B$ iz Γ, dovoljno je ili dokazati A iz Γ, ili B iz Γ.

Pravilo eliminacije disjunkcije ($\vee E$) izgleda znatno kompleksnije. Ono kaže da, ako je moguće (1) iz pretpostavki Γ dokazati formulu $A \vee B$, (2) iz pretpostavki Γ, A dokazati neku formulu C i (3) iz pretpostavki Γ, B dokazati tu istu formulu C, tada je iz pretpostavki Γ moguće dokazati formulu C. Intuitivno, ovo pravilo predstavlja grananje po slučajevima koje matematičari često primenjuju u dokazima („ako sam dokazao da važi $A \vee B$ tada mogu razmatrati dva odvojena slučaja: da važi A i da važi B i u svakom od ta dva slučaja da posebno dokažem tvrđenje C"). Tumačenje ovog pravila „unazad" bilo bi: da bih dokazao tvrđenje C iz pretpostavki Γ, dovoljno je dokazati $A \vee B$ iz Γ za neke proizvoljne formule A i B, a zatim dokazati C iz Γ, A i dokazati C iz Γ, B. Primetimo da ovo pravilo vrši oslobađanje pretpostavki – prilikom dokazivanja druge i treće premise dozvoljeno je koristiti A, odnosno B, kao pretpostavku, ali te pretpostavke se zatim oslobađaju i nisu dostupne u daljem toku dokaza.

Za implikaciju imamo sledeća pravila:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \Rightarrow I \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash A \Rightarrow B}{\Gamma \vdash B} \Rightarrow E$$

Pravilo uvođenja implikacije ($\Rightarrow I$) ima sledeći intuitivni smisao: ako iz pretpostavki Γ, A možemo dokazati neku formulu B, tada formulu $A \Rightarrow B$ možemo dokazati iz pretpostavki Γ. Dakle, ovde A imamo kao lokalnu pretpostavku koju koristimo da dokažemo B, a zatim je primenom ovog pravila oslobađamo (jer je ugrađujemo u levu stranu uvedene implikacije). Inverzno tumačenje pravila glasi: da bismo dokazali implikaciju $A \Rightarrow B$ iz nekog skupa pretpostavki Γ, dovoljno je dokazati B pod dodatnom pretpostavkom A.

Pravilo eliminacije implikacije ($\Rightarrow E$) je u stvari naše staro dobro pravilo *modus ponens*: ako iz Γ dokažemo A i $A \Rightarrow B$, tada možemo dokazati B iz istog skupa pretpostavki. Ili inverzno: da bismo dokazali neku formulu B, dovoljno je, za proizvoljno odabranu formulu A, dokazati A, kao i formulu $A \Rightarrow B$.

Za logičke konstante imamo sledeća pravila:

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A} \bot E \qquad\qquad \frac{}{\Gamma \vdash \top} \top I$$

Intuitivno, pravilo $\bot E$ kaže da ako je iz nekog skupa Γ moguće dokazati $\bot$, tada je iz Γ moguće dokazati bilo koju formulu A. Sa druge strane, pravilo $\top I$ kaže da je $\top$ moguće dokazati iz bilo kog skupa pretpostavki Γ. Kako ovo pravilo nema premisa, kontekst $\Gamma \vdash \top$ smatramo *aksiomom* prirodne dedukcije. Tumačeno unatraške, dokazivanje cilja oblika $\Gamma \vdash \top$ ne proizvodi nikakve potciljeve, pa se tu ta grana dokaza završava. Drugim rečima, kontekst $\Gamma \vdash \top$ se može nalaziti u listovima dokaza, budući da je aksioma.

Najzad, imamo pravilo izvođenja:

$$\frac{}{\Gamma, A \vdash A}\ \text{ass}$$

Ovo pravilo takođe uvodi kao aksiomu kontekst oblika $\Gamma, A \vdash A$. Intuitivno, uvek je moguće dokazati formulu A iz skupa pretpostavki koji sadrži tu formulu. Samim tim, ovakvi konteksti se mogu nalaziti u listovima dokaza. Tumačeno ,,unazad": ako se formula koja se dokazuje nalazi u skupu pretpostavki, onda je dokaz završen.

Za formulu A kažemo da je *teorema* prirodne dedukcije (u oznaci $\vdash A$) ako je u prirodnoj dedukciji moguće dokazati kontekst $\vdash A$. Slično, za formulu A kažemo da je *deduktivna posledica* skupa formula Γ (u oznaci $\Gamma \vdash A$), ako je moguće dokazati kontekst $\Gamma \vdash A$.

Primer 5.1. Dokažimo da je formula $\neg(A \vee B) \Rightarrow \neg A \wedge \neg B$ teorema prirodne dedukcije. Imamo sledeći dokaz konteksta $\vdash \neg(A \vee B) \Rightarrow \neg A \wedge \neg B$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg(A \vee B), A \vdash \neg(A \vee B)} \quad \dfrac{\dfrac{}{\neg(A \vee B), A \vdash A}}{\neg(A \vee B), A \vdash A \vee B} \vee I1}{\neg(A \vee B), A \vdash \bot} \neg E}{\neg(A \vee B) \vdash \neg A} \neg I \quad \dfrac{\dfrac{\dfrac{}{\neg(A \vee B), B \vdash \neg(A \vee B)} \quad \dfrac{\dfrac{}{\neg(A \vee B), B \vdash B}}{\neg(A \vee B), B \vdash A \vee B} \vee I2}{\neg(A \vee B), B \vdash \bot} \neg E}{\neg(A \vee B) \vdash \neg B} \neg I}{\neg(A \vee B) \vdash \neg A \wedge \neg B} \wedge I}{\vdash \neg(A \vee B) \Rightarrow \neg A \wedge \neg B} \Rightarrow I$$

Ovaj dokaz se konstruiše od korena ka listovima, tako što se određuju potciljevi koje je potrebno dokazati da bi se odgovarajuće pravilo moglo primeniti. Na primer, da bismo dokazali polazni kontekst $\vdash \neg(A \vee B) \Rightarrow \neg A \wedge \neg B$, na osnovu pravila $\Rightarrow I$, dovoljno je dokazati kontekst $\neg(A \vee B) \vdash \neg A \wedge \neg B$. Otuda se ovaj čvor dodaje u stablo kao dete korenog ɣora, a dokazivanje se nastavlja dokazivanjem ovog potcilja. Da bismo njega dokazali, dovoljno je, u skladu sa pravilom $\wedge I$, dokazati dva potcilja: $\neg(A \vee B) \vdash \neg A$ i $\neg(A \vee B) \vdash \neg B$. Odgovarajući čvorovi se dodaju u stablo dokaza kao deca ɣora $\neg(A \vee B) \vdash \neg A \wedge \neg B$, a dokazivanje se nastavlja dokazivanjem ovih potciljeva u proizvoljnom redosledu. Dokazivanje svakog potcilja se završava kada se dođe do aksiome.

Napomena 5.1. Kao što je objašnjeno u prethodnom primeru, dokazi se tipično konstruišu ,,unazad", počev od korena stabla. U tom smislu, pravila uvođenja veznika zapravo omogućavaju ,,eliminaciju" tog veznika, jer veznik obrnutom primenom pravila nestaje iz potciljeva. Pritom, formule koje se pojavljuju u dobijenim potciljevima su uvek *potformule* formula koje su se pojavljivale u polaznom kontekstu. Na primer, kada pravilo uvođenja konjunkcije primenimo unazad, tada umesto formule $A \wedge B$ dobijamo formule A i B, koje su potformule polazne formule $A \wedge B$. Ovo *svojstvo potformule* je veoma važno prilikom konstrukcije dokaza – u potciljevima baratamo samo sa potformulama koje se javljaju u polaznom cilju, pa nije potrebno ,,izmišljati" nikakve dodatne formule. Sa druge strane, pravila eliminacije nemaju svojstvo potformule. Na primer, primena pravila eliminacije negacije ,,unazad" u cilju dokaza formule $\bot$ zahteva da se ,,izmisli" neka nova formula A takva da se dalje dokazivanje svodi na potciljeve A i $\neg A$. U praksi, ta formula A se ne bira proizvoljno, već se bira tako da ju je moguće dokazati. Tipično, biramo neku formulu koja se već nalazi među pretpostavkama, poput formule $\neg(A \vee B)$ u prethodnom primeru.

Primer 5.2. Dokažimo da je formula $A \Rightarrow C$ deduktivna posledica skupa formula $A \Rightarrow B, B \Rightarrow C$. Dakle, potrebno je dokazati kontekst $A \Rightarrow B, B \Rightarrow C \vdash A \Rightarrow C$. Imamo sledeći dokaz u prirodnoj dedukciji:

$$\dfrac{\dfrac{\dfrac{A \Rightarrow B, B \Rightarrow C, A \vdash A \qquad A \Rightarrow B, B \Rightarrow C, A \vdash A \Rightarrow B}{A \Rightarrow B, B \Rightarrow C, A \vdash B} \Rightarrow E \qquad A \Rightarrow B, B \Rightarrow C, A \vdash B \Rightarrow C}{A \Rightarrow B, B \Rightarrow C, A \vdash C} \Rightarrow E}{A \Rightarrow B, B \Rightarrow C \vdash A \Rightarrow C} \Rightarrow I$$

Dakle, da bismo dokazali $A \Rightarrow C$, dokazujemo formulu C pod dodatnom pretpostavkom A. Da bismo to dokazali, dovoljno je dokazati formulu B, s obzirom da formulu $B \Rightarrow C$ već imamo među pretpostavkama, pa je moguće izvesti C eliminacijom implikacije. Najzad, dokazivanje formule B se opet vrši primenom eliminacije implikacije, s obzirom da formule A i $A \Rightarrow B$ postoje među pretpostavkama.

Opisani sistem prirodne dedukcije barata sa iskaznim formulama, pa se može koristiti za rezonovanje u iskaznoj logici. Važi sledeća teorema.

Teorema 5.1. *Sistem prirodne dedukcije je* saglasan *za iskaznu logiku: ako je* $\Gamma \vdash A$*, tada je* $\Gamma \vDash A$*. Specijalno, ako je* A *teorema prirodne dedukcije, tada je* A *tautologija.*

Dokaz. Dokaz se izvodi tako što se pokaže saglasnost svakog pravila izvođenja. Da bismo dokazali saglasnost pravila izvođenja, potrebno je dokazati da ako saglasnost važi za premise, tada važi i za zaključak. Na primer, za pravilo $\wedge I$, pretpostavimo da saglasnost važi za premise $\Gamma \vdash A$ i $\Gamma \vdash B$. To znači da je $\Gamma \vDash A$ i $\Gamma \vDash B$. Sada u svakoj valuaciji u kojoj važe sve formule iz Γ važe i formule A i B, pa važi i formula $A \wedge B$. Otuda je $\Gamma \vDash A \wedge B$. Dakle, saglasnost važi i za zaključak. Slično se dokazuje i za ostala pravila. □

Iz prethodne teoreme sledi da možemo utvrditi tautologičnost neke iskazne formule tako što konstruišemo njen formalni dokaz u prirodnoj dedukciji. Ovo je sasvim drugačiji pristup od, npr. utvrđivanja tautologičnosti metodom istinitosnih tablica. Naime, metod istinitosnih tablica je semantički metod: njime utvrđujemo tautologičnost tako što razmatramo sve moguće modele, tj. pretražujemo prostor valuacija. Sa druge strane, u sintaksno-deduktivnom pristupu, mi konstruišemo eksplicitan dokaz koji potvrđuje univerzalnu tačnost formule. Ovaj dokaz je značajan zato što nam pruža razumljiv argument zbog čega je neka formula valjana.

Prirodno se postavlja i pitanje potpunosti prirodne dedukcije za iskaznu logiku. Na žalost, ispostavi se da gore navedeni skup pravila ne obezbeđuje potpunost. Na primer, formulu $A \vee \neg A$ nije moguće dokazati pomoću do sada prikazanih pravila prirodne dedukcije. Sa druge strane, formula $A \vee \neg A$ je svakako tautologija. Dakle, nije moguće dokazati svaku tautologiju, pa potpunost ne važi.

Da bismo obezbedili potpunost, dovoljno je dodati još jedno pravilo u naš deduktivni sistem:

$$\dfrac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash A}\ \mathit{contr}$$

Ovo pravilo je poznato i kao *pravilo svođenja na kontradikciju* – ili što matematičari vole da kažu ,,pretpostavimo suprotno": da bismo dokazali A iz pretpostavki Γ, dovoljno je da pretpostavimo da važi $\neg A$ i da iz te dodatne pretpostavke izvedemo kontradikciju, tj. dokažemo $\bot$.

Primer 5.3. Želimo da dokažemo da je formula $\neg(A \wedge B) \Rightarrow \neg A \vee \neg B$ teorema prirodne dedukcije. Da bismo to dokazali, potrebno je da konstruišemo dokaz u čijem korenu se nalazi kontekst $\vdash \neg(A \wedge B) \Rightarrow \neg A \vee \neg B$. Zbog nedostatka prostora, ovaj dokaz ćemo prikazati iz delova. Najpre, polazeći od polaznog konteksta u korenu, imamo sledeći deo dokaza:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\neg(A \wedge B), \neg(\neg A \vee \neg B) \vdash \neg(A \wedge B)} \quad \dfrac{\vdots}{\neg(A \wedge B), \neg(\neg A \vee \neg B) \vdash A \wedge B}}{\neg(A \wedge B), \neg(\neg A \vee \neg B) \vdash \bot}\,\neg E}{\neg(A \wedge B) \vdash \neg A \vee \neg B}\,contr}{\vdash \neg(A \wedge B) \Rightarrow \neg A \vee \neg B}\,\Rightarrow I$$

Ovde smo primenom pravila uvođenja implikacije naš cilj sveli na potcilj $\neg(A \wedge B) \vdash \neg A \vee \neg B$. Zatim smo primenom pravila *contr* dokazivanje formule $\neg A \vee \neg B$ sveli na dokazivanje formule $\bot$ iz dodatne pretpostavke $\neg(\neg A \vee \neg B)$. Kako je formula $\neg(A \wedge B)$ među pretpostavka, na osnovu pravila $\neg E$, dovoljno je dokazati formulu $A \wedge B$ iz tekućeg skupa pretpostavki. Odgovarajući deo dokaza navodimo u nastavku.

$$\dfrac{\dfrac{\dfrac{\vdots}{\neg(A \wedge B), \neg(\neg A \vee \neg B) \vdash A} \quad \dfrac{\vdots}{\neg(A \wedge B), \neg(\neg A \vee \neg B) \vdash B}}{\neg(A \wedge B), \neg(\neg A \vee \neg B) \vdash A \wedge B}\,\wedge I}{\vdots}$$

Dakle, primenjujemo pravilo uvođenja konjunkcije kojim dokaz formule $A \wedge B$ svodimo na dokazivanje potciljeva A i B. Ovi delovi dokaza su dati u nastavku.

$$\dfrac{\dfrac{\dfrac{\neg(A \wedge B), \neg(\neg A \vee \neg B), \neg A \vdash \neg(\neg A \vee \neg B)} \quad \dfrac{\neg(A \wedge B), \neg(\neg A \vee \neg B), \neg A \vdash \neg A}{\neg(A \wedge B), \neg(\neg A \vee \neg B), \neg A \vdash \neg A \vee \neg B}\,\vee I1}{\neg(A \wedge B), \neg(\neg A \vee \neg B), \neg A \vdash \bot}\,\neg E}{\neg(A \wedge B), \neg(\neg A \vee \neg B) \vdash A}\,contr$$

$$\vdots$$

$$\dfrac{\dfrac{\dfrac{\neg(A \wedge B), \neg(\neg A \vee \neg B), \neg B \vdash \neg(\neg A \vee \neg B)} \quad \dfrac{\neg(A \wedge B), \neg(\neg A \vee \neg B), \neg B \vdash \neg B}{\neg(A \wedge B), \neg(\neg A \vee \neg B), \neg B \vdash \neg A \vee \neg B}\,\vee I2}{\neg(A \wedge B), \neg(\neg A \vee \neg B), \neg B \vdash \bot}\,\neg E}{\neg(A \wedge B), \neg(\neg A \vee \neg B) \vdash B}\,contr$$

$$\vdots$$

Dakle, formulu A dokazujemo tako što primenjujemo pravilo *contr*, čime se dokaz svodi na dokazivanje formule $\bot$ pod dodatnom pretpostavkom $\neg A$ („pretpostavimo suprotno"). Sada se, na osnovu pravila eliminacije negacije, dokaz svodi na dokazivanje formule $\neg A \vee \neg B$, s obzirom da $\neg(\neg A \vee \neg B)$ imamo među pretpostavkama. Dokaz formule B je analogan.

Napomena 5.2. Prikazani sistem prirodne dedukcije *bez* pravila *contr* predstavlja sistem *ituicionističke iskazne logike*. Intuicionistička logika je nastala krajem 19. veka kao jedan od pokušaja da se matematika ,,očisti" od paradoksa koji su se u njoj javili u tom periodu. *Intuicionizam* predstavlja filozofski pravac u matematici koji insistira na tome da se u matematici može dokazati samo ono za šta je moguće pružiti *konstruktivistički* dokaz, tj. dokaz koji direktno konstruiše tu formulu. Zbog toga intuicionisti odbacuju svođenje na kontradikciju kao metod dokazivanja: time što smo dokazali da $\neg A$ ne može da važi i dalje nam ne daje za pravo da tvrdimo da važi A, jer mi nismo konstruisali dokaz za A. U suštini, intuicionisti odbacuju *zakon isključenja trećeg* – $A \vee \neg A$ i zahtevaju da se eksplicitno konstruiše dokaz ili za A ili za $\neg A$.

Ako pravilima prirodne dedukcije pridružimo i pravilo *contr*, tada dobijamo *klasičnu iskaznu logiku*. U klasičnoj iskaznoj logici je dozvoljeno koristiti svođenje na kontradikciju kao legitiman metod dokazivanja. Samim tim, u klasičnoj logici je moguće dokazati i $A \vee \neg A$, o čemu govori sledeći primer.

Primer 5.4. Dokažimo da je formula $A \vee \neg A$ teorema prirodne dedukcije za *klasičnu* logiku. Imamo sledeći dokaz (ponovo prikazan iz delova, zbog nedostatka prostora):

$$\dfrac{\dfrac{\dfrac{\vdots}{\neg(A \vee \neg A) \vdash A} \qquad \dfrac{\vdots}{\neg(A \vee \neg A) \vdash \neg A}}{\dfrac{\neg(A \vee \neg A) \vdash \bot}{\vdash A \vee \neg A}\, contr}\, \neg E}$$

Prvi deo dokaza predstavlja koren u kome se nalazi kontekst $\vdash A \vee \neg A$ koji dokazujemo. Pravilom kontradikcije ovaj dokaz svodimo na dokazivanje kontradikcije iz $\neg(A \vee \neg A)$. Ovo se dokazuje eliminacijom negacije, po formuli A. To znači da je u nastavku potrebno dokazati dva potcilja: A i $\neg A$ iz iste pretpostavke $\neg(A \vee \neg A)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\neg(A \vee \neg A), \neg A \vdash \neg A}{\neg(A \vee \neg A), \neg A \vdash A \vee \neg A}\, \vee I2 \qquad \neg(A \vee \neg A), \neg A \vdash \neg(A \vee \neg A)}{\neg(A \vee \neg A), \neg A \vdash \bot}\, \neg E}{\neg(A \vee \neg A) \vdash A}\, contr}{\vdots}$$

Drugi deo dokaza predstavlja dokaz formule A. Ovde ponovo primenjujemo pravilo *contr*, pri čemu dobijamo dodatnu pretpostavku $\neg A$ iz koje treba dokazati kontradikciju. Ovo dalje radimo eliminacijom negacije po formuli $\neg(A \vee \neg A)$ koju imamo među pretpostavkama. Preostali potcilj $A \vee \neg A$ je sada lako dokazati jer imamo $\neg A$ među pretpostavkama, pa možemo primeniti pravilo uvođenja disjunkcije.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\neg(A \vee \neg A), A \vdash A}{\neg(A \vee \neg A), A \vdash A \vee \neg A}\, \vee I1 \qquad \neg(A \vee \neg A), A \vdash \neg(A \vee \neg A)}{\neg(A \vee \neg A), A \vdash \bot}\, \neg E}{\neg(A \vee \neg A) \vdash \neg A}\, \neg I}{\vdots}$$

Treći deo dokaza predstavlja dokaz formule $\neg A$. Ovde umesto pravila *ccontr* primenjujemo pravilo uvođenja negacije unazad, čime se među pretpostavkama pojavljuje dodatno i formula A. Dalji dokaz je sličan kao i u drugom delu dokaza.

Teorema 5.2. *Sistem prirodne dedukcije za klasičnu iskaznu logiku ima svojstvo* potpunosti*: ako je* $\Gamma \vDash A$*, tada je i* $\Gamma \vdash A$*. Specijalno, ako je formula* A *tautologija, tada je moguće dokazati formulu* A *u sistemu prirodne dedukcije.*

Napomena 5.3. Dokaz gornje teoreme izostavljamo zbog složenosti.

Teorema 5.3. *Neka je* Γ *konačan skup iskaznih formula, a* A *proizvoljna iskazna formula. Problem* $\Gamma \vdash A$ *je odlučiv.*

Dokaz. Da bismo utvrdili da li važi $\Gamma \vdash A$, dovoljno je da utvrdimo da li je $\Gamma \vDash A$, što je odlučivo na osnovu posledice 4.4. □

Posledica 5.1. *Problem* $\vdash A$*, tj. pitanje ,,Da li je* A *teorema prirodne dedukcije za klasičnu iskaznu logiku?" je odlučivo.*

Napomena 5.4. Iz dokaza prethodne teoreme sledi da mi možemo ispitivati dokazivost nekog tvrđenja u prirodnoj dedukciji (tj. utvrđivati da li dokaz postoji ili ne) bez konstruicije samog dokaza. Dovoljno je semantičkim metodama utvrditi tautologičnost odgovarajuće formule. Ako se ispostavi da je formula tautologija, tada znamo da dokaz postoji, iako ga nismo efektivno konstruisali. Ipak, efektivna konstrukcija dokaza je često poželjna u praksi, jer ona omogućava jednostavnu verifikaciju dobijenog odgovora (složenim procedurama odlučivanja često ne možemo verovati, jer mogu sadržati greške).

Sistem prirodne dedukcije se može proširiti tako da se može primeniti i na formule u logici prvog reda. Za ovo je dovoljno dodati odgovarajuća pravila uvođenja i eliminacije za kvantifikatore.

Pravila za univerzalni kvantifikator su:

$$\frac{\Gamma \vdash A[x \mapsto y]}{\Gamma \vdash \forall x.A}\,\forall I \qquad \frac{\Gamma \vdash \forall x.A}{\Gamma \vdash A[x \mapsto t]}\,\forall E$$

Pravilo uvođenja univerzalnog kvantifikatora ($\forall I$) u premisi sadrži formulu $A[x \mapsto y]$ koja nastaje tako što se promenljiva x u formuli A zameni sa y. Pritom, važno je da promenljiva y bude takva da se ne pojavljuje kao slobodna promenljiva ni u A ni u Γ. Intuitivno, to znači ,,neko y o kome nemamo nikakvih pretpostavki", tj. ,,proizvoljno fiksirano y". Sada pravilo možemo razumeti na sledeći način: ako iz pretpostavki Γ možemo da dokažemo da formula A važi za proizvoljno fiksirano y, tada A važi za svako x pod pretpostavkama Γ. Obratno, možemo reći: da bismo dokazali $\forall x.A$ iz Γ, dovoljno je da dokažemo da A važi kada umesto x stavimo neko proizvoljno fiksirano y.

Pravilo eliminacije univerzalnog kvantifikatora ($\forall E$) kaže da ako je moguće dokazati $\forall x.A$ iz nekih pretpostavki Γ, tada je moguće dokazati i bilo koju instancu formule A, tj. formulu $A[x \mapsto t]$ koja nastaje kada se x zameni proizvoljnim termom t. Pritom je jedino važno da term t ostane potpuno slobodan u $A[x \mapsto t]$, tj. ni jedna promenljiva koja se javlja u t ne sme biti vezana nekim kvantifikatorom u A. Inverzno tumačenje pravila je: da bismo dokazali neku instancu formule A po promenljivoj x, dovoljno je dokazati da važi $\forall x.A$. Tipično, ovo pravilo primenjujemo kada imamo $\forall x.A$ među pretpostavkama.

Pravila za egzistencijalni kvantifikator su sledeća:

$$\frac{\Gamma \vdash A[x \mapsto t]}{\Gamma \vdash \exists x.A}\,\exists I \qquad \frac{\Gamma \vdash \exists x.A \qquad \Gamma, A[x \mapsto y] \vdash B}{\Gamma \vdash B}\,\exists E$$

Pravilo uvođenja egzistencijalnog kvantifikatora ($\exists I$) kaže da ako je moguće dokazati neku instancu (po x) formule A iz pretpostavki Γ, tada je moguće dokazati formulu $\exists x.A$ (tj. tada postoji neko x za koje važi A). Ili obratno, da bismo dokazali $\exists x.A$ iz pretpostavki Γ, dovoljno je dokazati neku instancu formule A iz istih pretpostavki. I ovde važi uslov da term t mora biti slobodan u $A[x \mapsto t]$, tj. ni jedna njegova promenljiva ne sme biti vezana kvantifikatorom u A.

Prva premisa pravila eliminacije egzistencijalnog kvantifikatora ($\exists E$) je da se iz pretpostavki Γ može dokazati $\exists x.A$. U drugoj premisi, među pretpostavkama imamo formulu $A[x \mapsto y]$ u kojoj umesto x stoji y koje je opet ,,proizvoljno fiksirano y", tj. y koje se ne pojavljuje ni u jednoj od pretpostavki iz Γ, kao ni u formulama A i B. Smisao pravila je sledeći: ako smo dokazali $\exists x.A$ i ako možemo da, pod pretpostavkom da A važi za neko proizvoljno fiksirano y, dokažemo formulu B, tada je iz Γ moguće dokazati B. Intuitivno, to ,,proizvoljno fiksirano y" je upravo ono x za koje važi A, a za koje smo dokazali da postoji. Posmatranjem pravila unazad, možemo ga razumeti ovako: da bismo dokazali formulu B, dovoljno je dokazati $\exists x.A$ za neku formulu A, a onda pod pretpostavkom da za neko proizvoljno y važi A dokazati formulu B.

U logici prvog reda takođe možemo razmatrati sistem prirodne dedukcije sa i bez pravila *contr*, čime dobijamo klasičnu i intuicionističku logiku prvog reda. Kao i u iskaznoj logici, u logici prvog reda važi sledeća teorema u *saglasnosti*.

Teorema 5.4. *Sistem prirodne dedukcije za klasičnu logiku prvog reda je* saglasan*, tj. ako je* $\Gamma \vdash A$*, tada je i* $\Gamma \vDash A$*. Specijalno, ako je formula* A *teorema prirodne dedukcije za klasičnu logiku, onda je* A *valjana formula.*

Dokaz. Kao i u slučaju iskazne logike, potrebno je dokazati saglasnost svakog od pravila. Saglasnost pravila za iskazne veznike je već dokazana ranije, pa je potrebno još dokazati saglasnot pravila za kvantifikatore. Ilustracije radi, razmotrimo pravilo uvođenja univerzalnog kvantifikatora. Pretpostavimo da je premisa saglasna, tj. da iz $\Gamma \vdash A[x \mapsto y]$ sledi $\Gamma \vDash A[x \mapsto y]$. Ovo znači da za svaku strukturu $\mathcal{D}$ i valuaciju v koje zadovoljavaju sve formule iz Γ važi da je $\mathcal{D}_v(A[x \mapsto y]) = 1$. Pritom, kako se y ne pojavljuje kao slobodna promenljiva u Γ, sledi da vrednost formula iz Γ u strukturi $\mathcal{D}$ ne zavisi od vrednosti valuacije za promenljivu y. To znači da su formule iz Γ tačne u strukturi $\mathcal{D}$ i za svaku valuaciju v_y koja se od v razlikuje samo na promenljivoj y. Otuda će i formula $A[x \mapsto y]$ biti tačna za svaku takvu valuaciju v_y, kao logička posledica formula iz Γ. Samim tim će formula $\forall y.A[x \mapsto y]$ biti tačna u strukturi $\mathcal{D}$ i valuaciji v. Odavde sledi da će i formula $\forall x.A$ biti tačna u strukturi $\mathcal{D}$ i valuaciji v, s obzirom da su formule $\forall x.A$ i $\forall y.A[x \mapsto y]$ logički ekvivalentne (druga je dobijena od prve preimenovanjem vezane promenljive, tj. α-konverzijom koja je korektno izvedena, s obzirom da se y ne pojavljuje ni u A kao slobodna). Time smo dokazali da važi $\Gamma \vDash \forall x.A$, tj. da je i zaključak takođe saglaasn. Čitalac za vežbu može dokazati saglasnost ostalih pravila za kvantifikatore. □

Sledeća teorema, koju navodimo bez dokaza, govori o poptunosti sistema prirodne dedukcije za klasičnu logiku prvog reda.

Teorema 5.5. *Sistem prirodne dedukcije za klasičnu logiku prvog reda je* potpun*, tj. važi da iz* $\Gamma \vDash A$ *sledi da je* $\Gamma \vdash A$*. Specijalno, svaka valjana formula prvog reda je i teorema sistema prirodne dedukcije za klasičnu logiku prvog reda.*

Napomena 5.5. Prethodna teorema se pripisuje nemačkom matematičaru Kurtu Gedelu i poznata je pod nazivom *Gedelova teorema o potpunosti*. Ova teorema, iako veoma važna, često se nalazi u senci druge, mnogo poznatije *Gedelove teoreme o nepotpunosti aritmetike* koju izlažemo kasnije.

Primer 5.5. Dokažimo da je formula $(\exists x.\forall y.p(x,y)) \Rightarrow (\forall y.\exists x.p(x,y))$ teorema prirodne dedukcije za logiku prvog reda. Imamo sledeći dokaz, sa kontekstom $\vdash (\exists x.\forall y.p(x,y)) \Rightarrow (\forall y.\exists x.p(x,y))$ u korenu:

$$\dfrac{\dfrac{\exists x.\forall y.p(x,y) \vdash \exists x.\forall y.p(x,y) \qquad \dfrac{\dfrac{\dfrac{\forall y.p(x',y) \vdash \forall y.p(x',y)}{\forall y.p(x',y) \vdash p(x',y')}\,\forall E}{\forall y.p(x',y) \vdash \exists x.p(x,y')}\,\exists I}{\forall y.p(x',y) \vdash \forall y.\exists x.p(x,y)}\,\forall I}{\exists x.\forall y.p(x,y) \vdash \forall y.\exists x.p(x,y)}\,\exists E}{\vdash (\exists x.\forall y.p(x,y)) \Rightarrow (\forall y.\exists x.p(x,y))}\Rightarrow I$$

Dakle, polazeći od korena stabla, najpre primenjujemo pravilo uvođenja implikacije, čime dobijamo podcilj $\exists x.\forall y.p(x,y) \vdash \forall y.\exists x.p(x,y)$, tj. dokazujemo formulu $\forall y.\exists x.p(x,y)$ pod pretpostavkom $\exists x.\forall y.p(x,y)$. Kako imamo formulu sa vodećim egzistencijalnim kvantifikatorom među pretpostavkama, prirodno je da primenimo pravilo eliminacije egzistencijalnog kvantifikatora, čime nam preostaje da dokažemo formulu $\forall y.\exists x.p(x,y)$ pod pretpostavkom da formula $\forall y.p(x',y)$ važi za neko proizvoljno fiksirano x'. Ovo se dalje svodi na dokazivanje formule $\exists x.p(x,y')$ za proizvoljno fiksirano y' (pravilo uvođenja univerzalnog kvantifikatora). Da bismo dokazali formulu $\exists x.p(x,y')$, dovoljno je dokazati formulu $p(x,y')$ za neku instancu po x. Mi ćemo odabrati baš instancu $x \mapsto x'$, s obzirom da imamo $\forall y.p(x',y)$ među pretpostavkama, odakle jednostavno sledi $p(x',y')$ (eliminacija univerzalnog kvantifikatora).

Primer 5.6. Dokažimo da je formula $(\neg\exists x.p(x)) \Rightarrow (\forall y.\neg p(y))$ teorema prirodne dedukcije za logiku prvog reda.

$$\dfrac{\dfrac{\dfrac{\dfrac{\neg\exists x.p(x), p(z) \vdash p(z)}{\neg\exists x.p(x), p(z) \vdash \exists x.p(x)}\,\exists I \qquad \neg\exists x.p(x), p(z) \vdash \neg\exists x.p(x)}{\neg\exists x.p(x), p(z) \vdash \bot}\,\neg E}{\neg\exists x.p(x) \vdash \neg p(z)}\,\neg I}{\neg\exists x.p(x) \vdash \forall y.\neg p(y)}\,\forall I}{\vdash (\neg\exists x.p(x)) \Rightarrow (\forall y.\neg p(y))}\Rightarrow I$$

Dakle, najpre pravilom uvođenja implikacije, dokaz teoreme svodimo na potcilj $\neg\exists x.p(x) \vdash \forall y.\neg p(y)$. Da bismo dokazali $\forall y.\neg p(y)$, dovoljno je dokazati da važi $\neg p(z)$ za neko proizvoljno fiksirano z (uvođenje univerzalnog kvantifikatora). Dalje, da bismo dokazali $\neg p(z)$, dovoljno je dokazati kontradikciju ($\bot$) pod pretpostavkom $p(z)$ (uvođenje negacije). Kako među pretpostavkama imamo formulu $\neg\exists x.p(x)$, prirodno je da pokušamo primenu pravila eliminacije negacije po ovoj formuli. Time se dokaz svodi na potcilj $\exists x.p(x)$, za šta je dovoljno dokazati da $p(x)$ važi za neki proizvoljan term. Ovo trivijalno sledi, s obzirom da $p(z)$ imamo među pretpostavkama.

Teorema 5.6. *Problem $\Delta \vdash F$ u sistemu prirodne dedukcije za klasičnu logiku prvog reda je neodlučiv, ali jeste poluodlučiv.*

Dokaz. Problem $\Delta \vdash F$ je ekvivalentan sa $\Delta \models F$, pa teorema sledi iz teoreme 4.11. □

Posledica 5.2. *Problem $\vdash F$, tj. pitanje „Da li je F teorema u sistemu prirodne dedukcije za klasičnu logiku prvog reda?" je neodlučiv, ali je poluodlučiv.*

Napomena 5.6. Iz dokaza prethodne teoreme sledi da mi možemo utvrditi da je neka formula valjana (semantičkim metodama) i na osnovu toga zaključiti da postoji dokaz u prirodnoj dedukciji, iako ga nismo efektivno konstruisali. Ipak, u logici prvog reda, čak i više nego u iskaznoj logici, postoji potreba za efektivnom konstrukcijom dokaza, kako bismo mogli da verifikujemo dobijeni rezultat (proverom ispravnosti dokaza).

Jedan način, ne preterano efikasan, je da se sistematski nabrajaju svi ispravni dokazi (kao hijerarhijske strukture, odeljak 3.3.3). Ako je formula F teorema, pre ili kasnije ćemo doći do dokaza u čijem se korenu nalazi F. Ukoliko F nije teorema, takav dokaz ne postoji, pa se ovaj postupak nikada neće završiti. Otuda je opisani postupak jedna procedura poluodlučivanja.

U računarstvu se razmatraju i znatno efikasniji postupci pronalaženja i konstrukcije dokaza.

Glava 6

Izračunljivost i formalizacija matematike

U ovoj glavi podrazumevaćemo da radimo u sistemu prirodne dedukcije za klasičnu logiku prvog reda. Samim tim, oznaka $\vdash$ će označavati dokazivost u sistemu prirodne dedukcije. Ipak, sve navedeno će važiti u svakom deduktivnom sistemu za logiku prvog reda koji ima svojstvo saglasnosti i potpunosti.

6.1 Logičke teorije

U ovom poglavlju uvodimo pojam *logičke teorije prvog reda* i razmatramo osnovna meta-svojstva logičkih teorija.

Definicija 6.1. *Teorija prvog reda* $\mathcal{T}$ zadata skupom *aksioma* $\mathcal{A}_\mathcal{T}$ je skup svih rečenica koje se mogu dokazati iz $\mathcal{A}_\mathcal{T}$ u datom deduktivnom sistemu, tj. $\mathcal{T} = \{F \mid \mathcal{A}_\mathcal{T} \vdash F\}$. Ako je $F \in \mathcal{T}$, tada kažemo da je F *teorema* teorije $\mathcal{T}$ i označavamo sa $\vdash_\mathcal{T} F$.

Napomena 6.1. Skup aksioma $\mathcal{A}_\mathcal{T}$ teorije $\mathcal{T}$ može biti konačan ili prebrojivo beskonačan.

Napomena 6.2. Jasno je da je svaka aksioma teorije $\mathcal{T}$ ujedno i njena teorema, tj. $\mathcal{A}_\mathcal{T} \subseteq \mathcal{T}$. Pored toga, teorija sadrži i sve rečenice koje se iz aksioma mogu dokazati. Drugim rečima, teorija $\mathcal{T}$ je *deduktivno zatvorenje* svog skupa aksioma.

Napomena 6.3. Skup aksioma $\mathcal{A}_\mathcal{T}$ teorije $\mathcal{T}$ ne treba mešati sa skupom aksioma deduktivnog sistema. Aksiome deduktivnog sistema određuju koje se formule mogu dokazati u deduktivnom sistemu bez ikakvih dodatnih pretpostavki, tj. koje su formule *teoreme deduktivnog sistema*. Sa druge strane, aksiome teorije su, posmatrano iz ugla deduktivnog sistema, dodatne pretpostavke koje možemo koristiti prilikom izvođenja dokaza. Samim tim, *teoreme teorije* $\mathcal{T}$ će, pored teorema deduktivnog sistema, biti i mnoge druge formule koje se ne mogu izvesti samo iz aksioma deduktivnog sistema, ali se mogu izvesti uz pomoć aksioma teorije kao dodatnih pretpostavki.

Napomena 6.4. Specijalno, skup aksioma $\mathcal{A}_\mathcal{T}$ može biti i prazan. Tada je $\vdash_\mathcal{T} F$ ekvivalentno sa $\vdash F$, tj. skup teorema teorije $\mathcal{T}$ se poklapa sa skupom teorema

deduktivnog sistema. Ovakva ,,prazna" teorija se često naziva i *čist predikatski račun prvog reda*.

Definicija 6.2. Rečenica F je *deduktivna posledica* skupa rečenica Γ u teoriji $\mathcal{T}$ (u oznaci $\Gamma \vdash_{\mathcal{T}} F$) ako važi $\mathcal{A}_{\mathcal{T}}, \Gamma \vdash F$.

Napomena 6.5. Deduktivna posledica u teoriji je uopštenje pojma deduktivne posledice u deduktivnom sistemu, gde sada možemo, pored formula iz Γ, koristiti i aksiome teorije $\mathcal{T}$ kao dodatne pretpostavke prilikom izvođenja dokaza.

Definicija 6.3. Teorija $\mathcal{T}$ zadata skupom aksioma $\mathcal{A}_{\mathcal{T}}$ je *konzistentna* (ili *neprotivrečna*) ako ne postoji rečenica A takva da je i $A \in \mathcal{T}$ i $\neg A \in \mathcal{T}$. U suprotnom, teorija je *nekonzistentna*.

Napomena 6.6. Ako bi postojala rečenica takva da je $A \in \mathcal{T}$ i $\neg A \in \mathcal{T}$, to bi značilo da je moguće konstruisati dokaze formula A i $\neg A$ koji koriste aksiome $\mathcal{A}_{\mathcal{T}}$ kao pretpostavke. Međutim, iz ova dva dokaza bi se mogao konstruisati dokaz formule $\bot$, tj. izvesti kontradikcija (npr. u prirodnoj dedukciji bismo to uradili primenom pravila eliminacije negacije iz A i $\neg A$). Slično, ako se iz aksioma teorije $\mathcal{T}$ može izvesti $\bot$, tada se dalje može izvesti bilo koja formula A (npr. u prirodnoj dedukciji primenom pravila $\bot E$), tj. sve formule bi bile dokazive u teoriji $\mathcal{T}$. Specijalno, za proizvoljnu formulu A mogli bismo da dokažemo i A i $\neg A$. Dakle, teorija je nekonzistentna akko važi $\vdash_{\mathcal{T}} \bot$. Ponekad se i ovo uzima za definiciju nekonzistentnosti.

Definicija 6.4. *Model* teorije $\mathcal{T}$ zadate skupom aksioma $\mathcal{A}_{\mathcal{T}}$ je bilo koja $\mathcal{L}$-struktura $\mathcal{D}$ koja zadovoljava sve aksiome teorije $\mathcal{A}_{\mathcal{T}}$.

Narednu poznatu teoremu navodimo bez dokaza.

Teorema 6.1. *Teorija $\mathcal{T}$ je konzistentna akko ima bar jedan model.*

Definicija 6.5. Formula F je *zadovoljiva u teoriji* $\mathcal{T}$ ako je tačna u bar jednom modelu teorije $\mathcal{T}$. Formula F je *valjana u teoriji* $\mathcal{T}$ (u oznaci $\vDash_{\mathcal{T}} F$) ako je tačna u svim modelima teorije $\mathcal{T}$. Formula F je *logička posledica skupa formula* Γ *u teoriji* $\mathcal{T}$ (u oznaci $\Gamma \vDash_{\mathcal{T}} F$) ako je $\mathcal{A}(\mathcal{T}), \Gamma \vDash F$. Formule F i G su *logički ekvivalentne u teoriji* $\mathcal{T}$ (u oznaci $F \equiv_{\mathcal{T}} G$) ako je $F \vDash_{\mathcal{T}} G$ i $G \vDash_{\mathcal{T}} F$, tj. ako imaju iste vrednosti u svim modelima teorije $\mathcal{T}$.

Teorema 6.2. *Važi $\Gamma \vdash_{\mathcal{T}} F$ akko $\Gamma \vDash_{\mathcal{T}} F$. Specijalno, F je teorema teorije $\mathcal{T}$ akko je logički valjana u teoriji $\mathcal{T}$, tj. $\vdash_{\mathcal{T}} F$ akko $\vDash_{\mathcal{T}} F$.*

Prethodna teorema sledi iz saglasnosti i potpunosti deduktivnog sistema, kao i gornjih definicija.

Na kraju navodimo još jednu teoremu bez dokaza koja govori o izražajnosti logike prvog reda, u smislu (ne)mogućnosti da se aksiomama fiksira kardinalnost domena modela.

Teorema 6.3 (Skolem-Lovenhajmova teorema)**.** *Neka je $\mathcal{T}$ proizvoljna teorija prvog reda. Ako je $\mathcal{T}$ konzistentna, tada ona ima model bilo koje beskonačne kardinalnosti $\mathcal{K}$.*

Posledica 6.1. *Svaka konzistentna teorija prvog reda ima prebrojiv model.*

Posledica 6.2. *Teorija $\mathcal{T}$ ne može biti* kategorična*, tj. ne može fiksirati jedinstven model, do na izomorfizam.*

Dakle, u logici prvog reda nije moguće fiksirati interpretaciju aksiomatski zadate logičke teorije na jednu, unapred određenu strukturu. Uvek će postojati suštinski različite, međusobno neizomorfne strukture (čak različitih kardinalnosti) koje zadovoljavaju date aksiome. Najviše što se može postići je da te strukture makar budu *elementarno ekvivalentne*. Za dve strukture kažemo da su elementarno ekvivalentne ako je proizvoljna rečenica A tačna u jednoj strukturi akko je tačna u drugoj. To znači da se te strukture, iako neizomorfne, logički ponašaju isto, tj. slažu se po pitanju *istinitosti* tvrđenja koja se na jeziku $\mathcal{L}$ mogu izraziti.

Definicija 6.6. Teorija $\mathcal{T}$ nad jezikom $\mathcal{L}$ je *potpuna* akko za svaku rečenicu A važi da ili je $\vdash_{\mathcal{T}} A$ ili je $\vdash_{\mathcal{T}} \neg A$.

Napomena 6.7. Pojam *potpunosti* u smislu potpune teorije ne treba mešati sa pojmom potpunosti deduktivnog sistema. Potpunost deduktivnog sistema označava da je u tom deduktivnom sistemu moguće dokazati svaku valjanu formulu. Dakle, ovaj pojam predstavlja spoj između dedukcije i semantike. Sa druge strane, potpunost teorije je čisto deduktivni pojam koji govori o deduktivnoj moći teorije, tj. deduktivnog sistema ,,ojačanog" skupom aksioma teorije.

Primer 6.1. Čist predikatski račun nije potpuna teorija, s obzirom da npr. ni formula $\forall x.p(x)$ ni formula $\neg(\forall x.p(x))$ nisu dokazive (jer nisu valjane, a pretpostavljamo da je deduktivni sistem saglasan, tj. može dokazati samo valjane formule).

Dodavanjem aksioma teorije se povećava broj rečenica koje se mogu dokazati u deduktivnom sistemu. Pitanje je da li je dati skup aksioma dovoljan da za svaku rečenicu A možemo dokazati ili A ili $\neg A$ (ali nikako ne oba, jer bi tada teorija bila nekonzistentna)? Dakle, želimo da upotpunimo deduktivnu moć teorije, a da pritom ostanemo konzistentni. O važnost potpunosti teorije prvog reda govori sledeća teorema.

Teorema 6.4. *Teorija $\mathcal{T}$ je potpuna akko su joj svi modeli međusobno elementarno ekvivalentni.*

Dokaz. Ako je teorija potpuna, tada se za svaku rečenicu A može dokazati ili A ili $\neg A$. Pretpostavimo, određenosti radi, da je A dokazivo u teoriji. Na osnovu teoreme 6.2, sledi da ono što je dokazivo mora biti tačno u svim modelima teorije. Dakle, rečenica A je tačna u svim modelima teorije, a rečenica $\neg A$ je netačna u svim modelima teorije. Dakle, svaka rečenica ima istu interpretaciju u svim modelima teorije, pa su svi modeli elementarno ekvivalentni.

Obratno, ako su svi modeli elementarno ekvivalentni, onda za proizvoljnu rečenicu A važi ili da je ona tačna u svim modelima teorije, ili da je njena negacija $\neg A$ tačna u svim modelima teorije. Na osnovu teoreme 6.2, tada je ili A ili $\neg A$ dokaziva u teoriji $\mathcal{T}$. Dakle, teorija je potpuna. □

Imajući u vidu ranije napomene, iz prethodne teoreme zaključujemo da je potpunost teorije najviše što možemo očekivati od teorije prvog reda. Na žalost, ispostavlja se da ovo nije uvek moguće, tj. nije moguće svaki skup aksioma dopuniti do potpune teorije (a da se ne naruši konzistentost). O tome ćemo detaljnije govoriti kasnije.

Definicija 6.7. Teorija $\mathcal{T}$ je *odlučiva* ako je problem $\vdash_{\mathcal{T}} F$, gde je F proizvoljna formula, odlučiv.

Primetimo da teorija može biti odlučiva, iako sama logika prvog reda nije odlučiva. Ovo je zato što je problem $\vdash_{\mathcal{T}} F$, odnosno $\mathcal{A}_{\mathcal{T}} \vdash F$ potproblem opšteg problema $\Delta \vdash F$ (jer je $\mathcal{A}_{\mathcal{T}}$ fiksiran, unapred odabran skup aksioma, dok je u opštem slučaju skup Δ proizvoljan skup). Potproblem često može biti jednostavniji od opšteg problema.

Definicija 6.8. Za teoriju $\mathcal{T}$ kažemo da je *aksiomatska* ako je njen skup aksioma $\mathcal{A}_{\mathcal{T}}$ rekurzivan.

Rekurzivnost skupa aksioma znači da za proizvoljnu formulu možemo da utvrdimo da li je aksioma teorije ili ne. Ukoliko je skup aksioma konačan, on je svakako rekurzivan, pa će teorija sa konačnim brojem aksioma uvek biti aksiomatska. Sa druge strane, kod beskonačnih skupova aksioma, neophodno je da postoji mogućnost efektivnog utvrđivanja da li je nešto aksioma ili ne. Tipično, beskonačni skupovi aksioma se zadaju kao *šeme* aksioma, tj. obrasci koji definišu oblik i strukturu formule koja se smatra aksiomom. Otuda je moguće efektivno ispitati da li se data formula uklapa u date obrasce, tj. da li je *instanca* aksiomatske šeme ili ne.

Teorema 6.5. *Ako je teorija $\mathcal{T}$ aksiomatska i potpuna, ona je odlučiva.*

Dokaz. Jedna jednostavna intuitivna procedura odlučivanja bi se sastojala u rekurzivnom nabrajanju svih dokaza sa pretpostavkama iz skupa $\mathcal{A}_{\mathcal{T}}$. Ovo je moguće zato što je skup $\mathcal{A}_{\mathcal{T}}$ rekurzivan, pa je i sam rekurzivno nabrojiv. Kako je teorija potpuna, sledi da ili postoji dokaz rečenice A ili dokaz njene negacije $\neg A$. Otuda ćemo pre ili kasnije naići na jedan od ta dva dokaza, odakle ćemo utvrditi da li je $\vdash A$ ili ne. □

Iz prethodne teoreme zaključujemo da je, u slučaju aksiomatskih teorija, potpunost *jače* svojstvo od odlučivosti – svaka potpuna teorija je i odlučiva. Napomenimo da obratno ne mora da važi.

6.1.1 Peanova aritmetika

U ovom odeljku dajemo primer jedne logičke teorije koja je od fundamentalnog značaja kako za formalno zasnivanje matematike, tako i za teorijsko računarstvo. U pitanju je teorija aritmetike koja logički opisuje uobičajenu, intuitivnu strukturu prirodnih brojeva.

Neka je data signatura $\mathcal{L}$ koja sadrži simbol konstante 0, funkcijski simbol S arnosti 1, funkcijske simbole $+$ i $\cdot$ arnosti 2, kao i predikatski simbol $=$ arnosti 2. *Penaova aritmetika prvog reda* $\mathcal{P}$ je logička teorija zadata sledećim skupom *aksioma*:

1. $\forall x.x = x$
2. $\forall xy.x = y \Rightarrow y = x$
3. $\forall xyz.x = y \wedge y = z \Rightarrow x = z$
4. $\forall xy.x = y \Rightarrow S(x) = S(y)$

5. $\forall x_1 x_2 y_1 y_2 . x_1 = x_2 \wedge y_1 = y_2 \Rightarrow x_1 + y_1 = x_2 + y_2$
6. $\forall x_1 x_2 y_1 y_2 . x_1 = x_2 \wedge y_1 = y_2 \Rightarrow x_1 \cdot y_1 = x_2 \cdot y_2$
7. $\forall x . \neg(0 = S(x))$
8. $\forall xy . S(x) = S(y) \Rightarrow x = y$
9. $\phi(0) \wedge (\forall x . \phi(x) \Rightarrow \phi(S(x))) \Rightarrow (\forall x . \phi(x))$, za svaku formulu $\phi(x)$
10. $\forall x . x + 0 = x$
11. $\forall xy . x + S(y) = S(x + y)$
12. $\forall x . x \cdot 0 = 0$
13. $\forall xy . x \cdot S(y) = x \cdot y + x$

Napomena 6.8. Aksiome 1-6 predstavljaju *aksiome jednakosti*. Ovim se zahteva da za simbol jednakosti ($=$) važi refleksivnost, simetričnost i tranzitivnost, kao i saglasnost (*kongruentnost*) sa funkcijskim simbolima S, $+$ i $\cdot$. Aksiome 7 i 8 zahtevaju da različito konstruisani objekti budu različiti: 0 nije jednaka $S(x)$ ni za jedno x, a ako su x i y različiti, tada su i $S(x)$ i $S(y)$ različiti. Element $S(x)$ nazivaćemo *sledbenik od* x. Aksioma 9 predstavlja tzv. *aksiomatsku šemu indukcije*. Formula $\phi(x)$ je formula koja u sebi sadrži slobodna pojavljivanja promenljive x, pri čemu je $\phi(0)$ kraći zapis za $\phi(x)[x \mapsto 0]$, a $\phi(S(x))$ je kraći zapis za $\phi(x)[x \mapsto S(x)]$. Intuitivno, $\phi(x)$ predstavlja neko *svojstvo* koje neki element x ispunjava. Aksioma 9 kaže da ako 0 ima to svojstvo, i ako iz činjenice da x ima to svojstvo sledi da i $S(x)$ ima to svojstvo, onda svi elementi imaju to svojstvo. Ovo odgovara uobičajenom principu matematičke indukcije u skupu prirodnih brojeva. Aksiome 10 i 11 definišu operaciju sabiranja: $x + 0$ je uvek jednako x, dok se dodavanje sledbenika od y svodi na dodavanje y i određivanje sledbenika tako dobijenog zbira. Slično, aksiome 12 i 13 definišu operaciju množenja: proizvod $x \cdot 0$ je uvek nula, dok se proizvod x sa sledbenikom od y definiše tako što pomnožimo sa y, a zatim dodamo x.

Primetimo da je aksioma 9 zapravo aksiomatska šema iz koje možemo generisati prebrojivo mnogo instanci ove aksiome (za svaku formulu $\phi(x)$ nad jezikom $\mathcal{L}$ sa slobodnom promenljivom x po jednu). Ipak, uvek je moguće efektivno utvrditi da li je data formula instanca aksiome 9 ili ne. Otuda je skup aksioma rekurzivan, pa je Peanova aritmetika aksiomatska teorija.

Napomena 6.9. Uvodimo oznake $\overline{0} = 0, \overline{1} = S(0), \overline{2} = S(1), \overline{3} = S(2), \ldots$, koje nazivamo *numeralima*.

Primer 6.2. Primetimo da iz aksioma jednakosti uvek možemo zaključiti da iz npr. $s_1 = s_2$ sledi $s_1 + t = s_2 + t$ za proizvoljne termove s_1, s_2 i t. Slično važi i za funkcijske simbole S i $\cdot$. Evo dokaza u prirodnoj dedukciji:

$$\dfrac{\dfrac{s_1 = s_2 \vdash_{\mathcal{P}} s_1 = s_2 \qquad \dfrac{s_1 = s_2 \vdash_{\mathcal{P}} \forall x . x = x}{s_1 = s_2 \vdash_{\mathcal{P}} t = t}\,\forall E}{s_1 = s_2 \vdash_{\mathcal{P}} s_1 = s_2 \wedge t = t}\,\wedge I \qquad \dfrac{s_1 = s_2 \vdash_{\mathcal{P}} \forall x_1 x_2 y_1 y_2 . x_1 = x_2 \wedge y_1 = y_2 \Rightarrow x_1 + y_1 = x_2 + y_2}{s_1 = s_2 \vdash_{\mathcal{P}} s_1 = s_2 \wedge t = t \Rightarrow s_1 + t = s_2 + t}\,\forall E}{s_1 = s_2 \vdash_{\mathcal{P}} s_1 + t = s_2 + t}\,\Rightarrow E$$

pri čemu $\vdash_{\mathcal{P}}$ podrazumeva da se među pretpostavkama sa leve strane pored $s_1 = s_2$ nalaze i sve aksiome Peanove aritmetike. Dakle, „jednako se može zameniti jednakim". Slično, može se dokazati da iz $s = t$ i $t = r$ sledi $s = r$:

$$\dfrac{\dfrac{\dfrac{s=t, t=r \vdash_{\mathcal{P}} s=t \qquad s=t, t=r \vdash_{\mathcal{P}} t=r}{s=t, t=r \vdash_{\mathcal{P}} s=t \wedge t=r}\wedge I \qquad \dfrac{s=t, t=r \vdash_{\mathcal{P}} \forall xyz.x=y \wedge y=z \Rightarrow x=z}{s=t, t=r \vdash_{\mathcal{P}} s=t \wedge t=r \Rightarrow s=r}\forall E}{s=t, t=r \vdash_{\mathcal{P}} s=r}\Rightarrow E$$

Ova dva tvrđenja nam opravdavaju tzv. *zamenske dokaze*: da bismo dokazali da važi neka jednakost $s = t$, možemo da krenemo od s i da u njemu sukcesivno zamenjujemo neke podtermove njima jednakim termovima sve dok ne stignemo do terma t.

Primer 6.3. Dokažemo da je $\overline{2} + \overline{2} = \overline{4}$, tj. da važi $S(S(0)) + S(S(0)) = S(S(S(S(0))))$. Zaista, imamo zamenski dokaz: $S(S(0))+S(S(0)) = S(S(S(0))+S(0)) = S(S(S(S(0)) + 0)) = S(S(S(S(0))))$. Pritom, kao opravdanja zamena u prethodnom nizu koristili smo instance aksioma $\forall xy.x + S(y) = S(x + y)$ i $\forall x.x + 0 = x$.

Napomena 6.10. Izraz $s \neq t$ gde su s i t proizvoljni termovi je kraći zapis za $\neg(s = t)$.

Primer 6.4. Dokažimo da važi $\forall x.x \neq S(x)$. Za dokaz ovog tvrđenja koristićemo aksiomu indukcije, tj. aksiomu 9. Gornje tvrđenje možemo zapisati u obliku $\forall x.\phi(x)$, gde je $\phi(x)$ formula $x \neq S(x)$. Važi $\phi(0)$, odnosno $0 \neq S(0)$, na osnovu aksiome 7, eliminacijom univerzalnog kvantifikatora za $x \mapsto 0$. Pretpostavimo da za neko proizvoljno x važi $\phi(x)$, odnosno $x \neq S(x)$. Ako bi važilo $S(x) = S(S(x))$, primenom aksiome 8, eliminacijom univerzalnog kvantifikatora za $y \mapsto S(x)$, a zatim eliminacijom implikacije mogli bismo da dokažemo $x = S(x)$, odakle iz $x \neq S(x)$ (induktivna pretpostavka) sledi kontradikcija. Odavde možemo uvođenjem negacije da izvedemo $S(x) \neq S(S(x))$, tj. $\phi(S(x))$. Kako smo za proizvoljno x pod pretpostavkom $\phi(x)$ dokazali $\phi(S(x))$, uvođenjem implikacije i univerzalnog kvantifikatora dobijamo $\forall x.\phi(x) \Rightarrow \phi(S(x))$. Sada uvođenjem konjunkcije imamo $\phi(0) \wedge (\phi(x) \Rightarrow \phi(S(x)))$, pa iz aksiome 9, eliminacijom implikacije sledei $\forall x.\phi(x)$, tj. $\forall x.x \neq S(x)$.

Primer 6.5. Dokažimo da važi $\forall x.0+x = x$. Ovo ćemo dokazati pomoću aksiome 9 (aksioma indukcije). Neka je formula $\phi(x)$ upravo formula $0 + x = x$. Svakako važi $\phi(0)$, tj. $0 + 0 = 0$, na osnovu aksiome $\forall x.x + 0 = x$. Pretpostavimo da važi $\phi(x)$ za neko x, tj. važi $0 + x = x$. Treba dokazati da važi formula $\phi(S(x))$, tj. da je $0 + S(x) = S(x)$. Kako je $0 + S(x) = S(0 + x) = S(x)$ (jer je $0 + x = x$ na osnovu induktivne pretpostavke), sleei da važi $\forall x.\phi(x) \Rightarrow \phi(S(x))$. Prema aksiomi 9, imamo da je $\forall x.0 + x = x$.

Primer 6.6. Posledica prethodnog primera je da važi $\forall x.0 + x = x + 0$. Naime, za proizvoljno fiksirano y važi $y + 0 = y$ (pravilom eliminacije univerzalnog kvantifikatora iz aksiome 10), kao i $0 + y = y$ na osnovu prethodnog primera. Na osnovu simetričnosti imamo da je $y = y + 0$, a na osnovu tranzitivnosti da je $0 + y = y + 0$. Kako je y proizvoljno i fiksirano, možemo primeniti pravilo uvođenja univerzalnog kvantifikatora, tj. važi $\forall x.0 + x = x + 0$.

Primer 6.7. Dokažimo da važi $\forall xy.S(y) + x = S(y + x)$. Zapišimo ovo tvrđenje u obliku $\forall x.\phi(x)$, gde je $\phi(x)$ formula $\forall y.S(y) + x = S(y + x)$. Dokažimo ovo tvrđenje indukcijom, tj. pomoću aksiome 9. Formula $\phi(0)$, tj. $\forall y.S(y) + 0 =$

$S(y+0)$ važi, jer za proizvoljno fiksirano y važi $S(y)+0 = S(y)$ (na osnovu aksiome 10, eliminacijom univerzalnog kvantifikatora), a iz $y+0 = y$ sledi $S(y+0) = S(y)$, pa na osnovu simetričnosti i tranzitivnosti jednakosti važi $S(y)+0 = S(y+0)$. Otuda, uvođenjem univerzalnog kvantifikatora važi $\forall y.S(y)+0 = S(y+0)$. Dalje, pretpostavimo da važi $\phi(x)$ za neko x, tj. da važi $\forall y.S(y)+x = S(y+x)$. Imamo da za proizvoljno fiksirano y važi $S(y)+S(x) = S(S(y)+x) = S(S(y+x))$ na osnovu induktivne pretpostavke. Dalje je $S(S(y+x)) = S(y+S(x))$, jer je $S(y+x) = y+S(x)$ (na osnovu aksiome 11, eliminacijom univerzalnog kvantifikatora i primenom simetričnosti). Dakle, imamo da je $S(y)+S(x) = S(y+S(x))$ za proizvoljno fiksirano y. Otuda, uvođenjem univerzalnog kvantifikatora imamo $\forall y.S(y)+S(x) = S(y+S(x))$, tj. važi $\phi(S(x))$. Dakle, za proizvoljno x, iz pretpostavke $\phi(x)$ smo dokazali $\phi(S(x))$, odakle uvođenjem implikacije i univerzalnog kvantifikatora dokazujemo $\forall x.\phi(x) \Rightarrow \phi(S(x))$. Na osnovu aksiome 9, sada zaključujemo da važi $\forall x.\phi(x)$, tj. $\forall xy.S(y)+x = S(y+x)$.

Primer 6.8. Dokažimo da važi $\forall xy.x+y = y+x$. Ovo je isto što i $\forall x.\forall y.x+y = y+x$, odnosno $\forall x.\phi(x)$, gde je $\phi(x)$ formula $\forall y.x+y = y+x$, sa slobodnom promenljivom x. U dokazu se ponovo služimo indukcijom. $\phi(0)$ je isto što i $\forall y.0+y = y+0$, što je dokazano u primeru 6.6. Pretpostavimo da za neko proizvoljno x važi $\phi(x)$, tj. važi $\forall y.x+y = y+x$. Za proizvoljno y važi $S(x)+y = S(x+y)$ na osnovu primera 6.7. Prema induktivnoj pretpostavci i na osnovu simetričnosti važi $y+x = x+y$, pa je $S(x)+y = S(x+y) = S(y+x) = y+S(x)$ (poslednja jednakost sledi iz aksiome 11, primenom simetričnosti). Kako je y proizvoljno, uvođenjem univerzalnog kvantifikatora imamo $\forall y.S(x)+y = y+S(x)$, tj. važi $\phi(S(x))$. Kako smo za proizvoljno x iz $\phi(x)$ dokazali $\phi(S(x))$, važi $\forall x.\phi(x) \Rightarrow \phi(S(x))$, pa na osnovu aksiome 9 imamo da važi $\forall x.\phi(x)$, tj. važi $\forall xy.x+y = y+x$.

Uočimo strukturu čiji je domen skup $\mathbb{N} = \{0, 1, \ldots\}$ *prirodnih brojeva*. Pretpostavimo da ova struktura simbole iz signature interpretira na sledeći način:

- 0 se interpretira kao prirodan broj *nula* (tj. kao 0).
- $=$ se interpretira kao *relacija jednakosti* nad $\mathbb{N}$, tj. kao relacija $= = \{(x,x) \mid x \in \mathbb{N}\}$.
- S se interpretira kao funkcija *sledbenika* u skupu $\mathbb{N}$, tj. kao funkcija $s : \mathbb{N} \to \mathbb{N}$ definisana sa $s(x) = x+1$ za svako x.
- $+$ se interpretira kao operacija *sabiranja* $+$ nad skupom prirodnih brojeva
- $\cdot$ se interpretira kao operacija *množenja* $\cdot$ nad skupom prirodnih brojeva.

Može se pokazati da ova struktura zadovoljava sve aksiome teorije $\mathcal{P}$. Nazivamo je i *standardni model aritmetike* i označavamo ga takođe sa $\mathbb{N}$. U njemu će se numeral $\overline{n}$ interpretirati kao prirodan broj n. Iz postojanja ovog modela sledi sledeća teorema.

Teorema 6.6. *Peanova aritmetika $\mathcal{P}$ je konzistentna.*

Standardni model $\mathbb{N}$ je, naravno, prebrojiv. Kao posledica Skolem-Lovenhajmove teoreme, postojaće i *nestandardni modeli* Peanove aritmetike čiji će domeni biti neprebrojivo beskonačni.

Napomena 6.11. U prethodnom razmatranju, konzistentnost Peanove aritmetike smo zaključili na osnovu postojanja standardnog modela $\mathbb{N}$. Ovaj model se konstruiše u okviru *teorije skupova*: broj 0 se identifikuje sa praznim skupom $\varnothing$, a sledbenik broja n se identifikuje sa skupom $n \cup \{n\}$ (tj. broj 1 se identifikuje sa skupom $\{0\}$, broj 2 sa skupom $\{0, 1\}$, broj 3 sa skupom $\{0, 1, 2\}$ i sl.). Uopšte, dokazivanje konzistentnosti teorije $\mathcal{T}$ na osnovu postojanja modela uvek podrazumeva da je taj model konstruisan u okviru neke druge teorije $\mathcal{T}'$, za koju pretpostavljamo da je konzistentna. Dakle, konzistentnost teorije $\mathcal{T}$ je relativna, do na konzistentnost teorije $\mathcal{T}'$. Da bismo dokazali apsolutnu konzistentnost teorije $\mathcal{T}$, potrebno bi bilo da možemo da dokažemo konzistentost teorije $\mathcal{T}$ u okviru nje same. O ovom problemu u slučaju Peanove aritmetike govorićemo u narednom poglavlju.

6.2 Gedelove teoreme

6.2.1 Izrazivost izračunljivih funkcija na jeziku aritmetike

Neka je f proizvoljna parcijalna funkcija arnosti k. Pod *grafom* funkcije f podrazumevamo relaciju $\rho_f \subseteq \mathbb{N}^k \times \mathbb{N}$ definisanu sa $\rho_f = \{((x_1, \ldots, x_k), y) \mid f(x_1, \ldots, x_k) = y\}$. Za funkciju f kažemo da je *izraziva* na jeziku Peanove aritmetike $\mathcal{P}$ ako postoji formula $R_f(x_1, \ldots, x_k, y)$ sa slobodnim promenljivama $x_1, \ldots, x_k, y$ nad signaturom teorije $\mathcal{P}$ takva da svaki izbor brojeva $n_1, \ldots, n_k, m \in \mathbb{N}$ važi da je $R_f(\overline{n}_1, \ldots, \overline{n}_k, \overline{m})$ tačna u strukturi $\mathbb{N}$ akko $(n_1, \ldots, n_k, m) \in \rho_f$.

Primer 6.9. Funkcija $z = mod(x, y)$ koja izračunava ostatak pri deljenju x sa y se može izraziti na sledeći način:

$$\exists q.x = y \cdot q + z \wedge z < y$$

Ova formula ima slobodne promenljive x, y, z i u standardnom modelu aritmetike je tačna akko je $z = mod(x, y)$.

Napomena 6.12. U prethodnom primeru, koristili smo simbol $<$ koji, formalno, nije deo signature teorije $\mathcal{P}$. Ipak, relaciju $z < y$ je lako izraziti na jeziku Peanove aritmetike:

$$\exists k.z + S(k) = y$$

Ova formula je tačna u $\mathbb{N}$ akko je $z < y$ tačno u $\mathbb{N}$. Na sličan način se mogu izraziti i drugi relacioni operatori ($\leq$, $>$, $\geq$). U nastavku ćemo, zbog jednostavnosti, uvek koristiti ove relacione simbole kao da su deo jezika, uz napomenu da ih uvek možemo eliminisati na opisan način i svesti formulu na „čistu Peanovu aritmetiku".

Primer 6.10. Slično prethodnom primeru, funkciju $z = div(x, y)$ možemo izraziti kao:

$$\exists r.x = y \cdot z + r \wedge r < y$$

Ova formula je tačna za neko x, y, z akko je $z = div(x, y)$.

Lema 6.1. *Neka su date funkcije h arnosti m i $g_1, \ldots, g_m$ arnosti k. Ako su ove funkcije izrazive na jeziku aritmetike, tada je to i funkcija $f = Sub(h, g_1, \ldots, g_k)$.*

Dokaz. Neka je $H(u_1, \ldots, u_m, y)$ formula koja izražava funkciju $y = h(u_1, \ldots, u_m)$ i neka su $G_i(x_1, \ldots, x_k, z)$ formule koje izražavaju funkcije $z = g_i(x_1, \ldots, x_k)$. Formula:

$$\exists u_1 \ldots u_m . H(u_1, \ldots, u_m, y) \wedge G_1(x_1, \ldots, x_k, u_1) \wedge \ldots \wedge G_m(x_1, \ldots, x_k, u_m)$$

izražava funkciju f. □

Primer 6.11. Funkcija definisana sa:

$$\beta(x_1, x_2, x_3) = mod(x_1, 1 + (x_3 + 1) \cdot x_2)$$

je izraziva na jeziku aritmetike, s obzirom da se može dobiti supstitucijom izrazive funkcije $1 + (x_3 + 1) \cdot x_2$ u funkciji *mod* za koju smo pokazali da je izraziva. Dakle, postoji formula $R_\beta(x_1, x_2, x_3, y)$ na jeziku aritmetike koja je tačna akko je $y = \beta(x_1, x_2, x_3)$.

Funkcija iz prethodnog primera je u literaturi poznata kao *Gedelova β funkcija*. Jedno njeno zanimljivo svojstvo opisano je sledećom lemom.

Lema 6.2. *Neka je data proizvoljna sekvenca prirodnih brojeva $a_0, \ldots, a_n$. Tada postoje s i t takvi da važi:*

$$\beta(s, t, i) = a_i$$

za svako $i \in \{0, \ldots, n\}$.

Dokaz. Dokaz se zasniva na činjenici da postoji t takvo da su brojevi $1 + t, 1 + 2 \cdot t, 1 + 3 \cdot t, \ldots, 1 + (n + 1) \cdot t$ uzajamno prosti (što ovde nećemo dokazivati). Otuda, na osnovu *Kineske teoreme o ostacima* moguće je pronaći s takvo da važi:

$$\begin{array}{l} s \ mod \ (1 + t) = a_0 \\ s \ mod \ (1 + 2 \cdot t) = a_1 \\ \ldots \\ s \ mod \ (1 + (n + 1) \cdot t) = a_n \end{array}$$

□

Prethodna lema je značajna zato što omogućava da *kodiramo* proizvoljne sekvence brojeva $a_0, \ldots, a_n$ parom brojeva s i t, pri čemu je to kodiranje izrazivo na jeziku aritmetike. Ovo omogućava da se dokaže sledeća lema.

Lema 6.3. *Neka su date funkcije g arnosti k i h arnosti $k + 2$. Ako su g i h izrazive na jeziku aritmetike, tada je i funkcija $f = Rec(g, h)$ takva.*

Dokaz. Neka je $G(x_1, \ldots, x_k, y)$ formula koja izražava da je $y = g(x_1, \ldots, x_k)$, a neka je formula $H(y, x_1, \ldots, x_k, u, z)$ formula koja izražava da je $z = h(y, x_1, \ldots, x_k, u)$. Imamo da je:

$$\begin{array}{lcl} f(0, x_1, \ldots, x_k) & = & g(x_1, \ldots, x_k) \\ f(y + 1, x_1, \ldots, x_k) & = & h(y, x_1, \ldots, x_k, f(y, x_1, \ldots, x_k)) \end{array}$$

Da bismo izrazili da je $z = f(y, x_1, \ldots, x_k)$, posmatrajmo sekvencu vrednosti:

$$\begin{array}{l} a_0 = f(0, x_1, \ldots, x_k) \\ a_1 = f(1, x_1, \ldots, x_k) \\ \ldots \\ a_y = f(y, x_1, \ldots, x_k) \end{array}$$

Prema prethodnoj lemi, postoje s i t takvi da je: $\beta(s,t,i) = a_i$ za svako $i \leq y$. Neka je data formula:

$$\exists st. \quad \left(\begin{array}{ll} G(x_1, \ldots, x_k, \beta(s,t,0)) & \wedge \\ \forall i.i < y \Rightarrow H(i, x_1, \ldots, x_k, \beta(s,t,i), \beta(s,t,S(i))) & \wedge \\ z = \beta(s,t,y) & \end{array} \right)$$

Ova formula izražava da je $z = f(y, x_1, \ldots, x_k)$. □

Napomena 6.13. U dokazu prethodne leme smo u formuli koristili funkcijski simbol β koji ne postoji u signaturi aritmetike. Ovo smo uradili da bismo pojednostavili notaciju. To nije „varanje", zato što smo uvek mogli da preformulišemo formulu tako da se koristi formula $R_\beta(s,t,i,u)$ kojom se izražava da je $u = \beta(s,t,i)$, umesto terma $\beta(s,t,i)$. Uopšte, za proizvoljnu izrazivu funkciju f, formulu:

$$A(x_1, \ldots, x_n, f(x_1, \ldots, x_n))$$

uvek možemo izraziti i kao:

$$\exists u.R_f(x_1, \ldots, x_n, u) \wedge A(x_1, \ldots, x_n, u)$$

gde je $R_f(x_1, \ldots, x_n, u)$ formula koja izražava da je $u = f(x_1, \ldots, x_n)$. Na ovaj način se možemo osloboditi funkcijskog simbola f i izraziti formulu na jeziku „čiste" aritmetike. Mi ćemo u praksi uglavnom koristiti funkcijsku notaciju zbog jednostavnosti, ali ćemo uvek imati u vidu da se ta notacija može lako transformisati na opisani način.

Iz prethodnih lema sledi važna teorema.

Teorema 6.7. *Svaka primitivno rekurzivna funkcija je izraziva na jeziku Peanove aritmetike.*

Da bismo izrazili proizvoljnu μ-rekurzivnu funkciju na jeziku aritmetike, potrebno je još izraziti i operator minimizacije. O ovome govori sledeća lema.

Lema 6.4. *Ako je funkcije $g(y, x_1, \ldots, x_k)$ izraziva na jeziku Peanove aritmetike, tada je takva i funkcija:*

$$f(x_1, \ldots, x_k) = \mu y[g(y, x_1, \ldots, x_k) = 0]$$

Dokaz. Pretpostavimo da je $R_g(y, x_1, \ldots, x_k, z)$ aritmetička formula koja izražava da je $z = g(y, x_1, \ldots, x_k)$. Razmotrimo formulu:

$$R_g(y, x_1, \ldots, x_k, 0) \wedge (\forall y'.y' < y \Rightarrow (\exists u.R_g(y', x_1, \ldots, x_k, u) \wedge u \neq 0))$$

Ova formula je tačna za neko $x_1, \ldots, x_k, y$ akko je $y = f(x_1, \ldots, x_k)$. □

Sada imamo sledeću teoremu.

Teorema 6.8. *Svaka μ-rekurzivna funkcija se može izraziti na jeziku Peanove aritmetike.*

Na kraju, povežimo izrazivost izračunljivih funkcija sa *dokazivošću* u Peanovoj aritmetici.

Teorema 6.9. *Neka je f bilo koja μ-izračunljiva funkcija arnosti k i neka za neke konkretne vrednosti $n_1, \ldots, n_k, m \in \mathbb{N}$ važi da je $m = f(n_1, \ldots, n_k)$. Tada važi:*

$$\vdash_{\mathcal{P}} R_f(\overline{n}_1, \ldots, \overline{n}_k, \overline{m})$$

gde je $R_f(x_1, \ldots, x_k, y)$ formula koja izražava da je $y = f(x_1, \ldots, x_k)$.

Dokaz. (skica) Intuitivno, formula $R_f(\overline{n}_1, \ldots, \overline{n}_k, \overline{m})$ će sadržati aritmetičke operacije S, $+$ i $\cdot$ primenjene nad konkretnim numeralima, nad kojima je jednostavno dokazati odgovarajuće identitete (setimo se kako smo dokazivali da je $\overline{2} + \overline{2} = \overline{4}$). Sa druge strane, formula može sadržati i egzistencijalne kvantifikatore, pri čemu se takvi delovi formule mogu dokazati uvođenjem konkretnog numerala (pravilo uvođenja egzistencijalnog kvantifikatora, primenjeno unatraške). Najzad, iz dokaza prethodnih lema vidimo da se univerzalni kvantifikatori uvek pojavljuju u ograničenoj formi (npr. $\forall k.k < n \Rightarrow \ldots$, gde je n parametar koji se zamenjuje konkretnim numeralom). To znači da se dokazivanje takvih formula može svesti na dokazivanje konačno mnogo slučajeva. □

Dokaz da je $\vdash_{\mathcal{P}} R_f(\overline{n}_1, \ldots, \overline{n}_k, \overline{m})$ ćemo nazivati *dokaz izračunavanja* $m = f(n_1, \ldots, n_m)$.

Primer 6.12. Dokažimo da je $mod(5, 2) = 1$. Potrebno je dokazati da važi:

$$\exists q.\overline{5} = \overline{2} \cdot q + \overline{1} \wedge (\exists k.\overline{1} + S(k) = \overline{2})$$

Da bismo dokazali ovu egzistencijalno kvantifikovanu formulu, dovoljno je dokazati potformulu za neko konkretno q. Zamenimo q sa $\overline{2}$, tj. dokažimo formulu:

$$\overline{5} = \overline{2} \cdot \overline{2} + \overline{1} \wedge (\exists k.\overline{1} + S(k) = \overline{2})$$

Prvi konjunkt se dokazuje jednostavno, sledećim zamenama zasnovanim na aksiomama Peanove aritmetike:

$$\begin{array}{ll}
S(S(0)) \cdot S(S(0)) + S(0) & = \\
S(S(0)) \cdot S(0) + S(S(0)) + S(0) & = \\
S(S(0)) \cdot 0 + S(S(0)) + S(S(0)) + S(0) & = \\
0 + S(S(0)) + S(S(0)) + S(0) & = \\
S(S(0)) + S(S(0)) + S(0) & = \\
S(S(S(0)) + S(0)) + S(0) & = \\
S(S(S(S(0)) + 0)) + S(0) & = \\
S(S(S(S(0)))) + S(0) & = \\
S(S(S(S(0)))) + 0) & = \\
S(S(S(S(0)))))
\end{array}$$

Za drugi konjunkt možemo ponovo primeniti pravilo uvođenja egzistencijalnog kvantifikatora, dokazujući potformulu za konkretan numeral 0:

$$\overline{1} + S(0) = \overline{2}$$

Ovo se opet lako dokazuje, s obzirom da su u pitanju konkretni numerali:

$$S(0) + S(0) = S(S(0) + 0) = S(S(0))$$

6.2.2 Kodiranje termova, formula i dokaza

Kao i sve hijerarhijske strukture, termovi, formule i dokazi u Peanovoj aritmetici se mogu kodirati prirodnim brojevima. Pritom, funkcije kodiranja su efektivno izračunljive i mogu se izraziti na jeziku aritmetike, kao što smo prikazali u prethodnom odeljku. Specijalno, sami kôdovi se mogu predstaviti odgovarajućim numeralima. Ovo nam daje vrlo zanimljivu mogućnost: da tvrđenja o samoj aritmetici (tzv. *metatvrđenja*, poput ,,formula A je dokaziva", ,,teorija je konzistentna" i sl.) izražavamo u obliku tvrđenja *u samoj aritmetici*.

Formalno, neka je sa $\lceil t \rceil$ označen numeral koji odgovara kôdu terma t nad jezikom Peanove aritmetike. Slično, neka $\lceil A \rceil$ predstavlja kôd formule A, a $\lceil D \rceil$ kôd dokaza D. Na jeziku aritmetike možemo izraziti tvrđenje: ,,dokaz sa kôdom y dokazuje formulu čiji je kôd x". Intuitivno, potrebno je dekodirati y, a zatim odrediti kôd formule u korenu dobijenog dokaza i uporediti ga sa x. Iz ove intuitivne izračunljivosti, prema Čerčovoj tezi sledi i formalna izračunljivost, a samim tim i izrazivost u Peanovoj aritmetici. Označimo formulu koja kodira ovo tvrđenje sa $proof(x, y)$. Sada formula $\exists y.proof(x, y)$ govori da ,,postoji dokaz formule čiji je kôd x", odnosno ,,formula čiji je kôd x je dokaziva". U daljem tekstu, ovu formulu ćemo označavati sa $pr(x)$.

6.2.3 Nepotpunost i neodlučivost aritmetike

U ovom odeljku razmatramo fundamentalna pitanja o Peanovoj aritmetici kao što su pitanja potpunosti i odlučivosti. Najpre ćemo dokazati nekoliko zanimljivih lema koje važe za Peanovu aritmetiku. Ove leme ćemo kasnije koristiti u dokazima glavnih teorema u ovom odeljku.

Lema 6.5. *Za svaku rečenicu A Peanove aritmetike važi da ako je $\vdash_{\mathcal{P}} A$ tada je i $\vdash_{\mathcal{P}} pr(\lceil A \rceil)$.*

Dokaz. Pretpostavimo da je $\vdash_{\mathcal{P}} A$. To znači da postoji dokaz D_A formule A u Peanovoj aritmetici. Potrebno je dokazati formulu $pr((\lceil A \rceil)$, tj. formulu $\exists y.proof(\lceil A \rceil, y)$. Za ovo je dovoljno pronaći neki numeral $\overline{n}$ takav da je moguće dokazati $proof(\lceil A \rceil, \overline{n})$ (nakon toga ćemo samo primeniti pravilo uvođenja egzistencijalnog kvantifikatora). Uzmimo da je $\overline{n} = \lceil D_A \rceil$. Sada se dekodiranjem broja n može efektivno utvrditi da je kôd formule koju dokazuje dokaz kodiran sa n upravo jednak $\lceil A \rceil$. Dokaz ovog izračunavanja je upravo dokaz formule $proof(\lceil A \rceil, \lceil D_A \rceil)$. □

Lema 6.6. *Za svaku rečenicu A Peanove aritmetike važi da ako je $\vdash_{\mathcal{P}} pr(\lceil A \rceil)$, onda je i $\vdash_{\mathcal{P}} A$.*

Dokaz. Iz dokazivosti formule $pr(\lceil A \rceil)$ u teoriji $\mathcal{P}$, tj. formule $\exists y.proof(\lceil A \rceil, y)$, sledi da je ova formula tačna u svakom modelu teorije $\mathcal{P}$, na osnovu saglasnosti deduktivnog sistema. Konkretno, tačna je i u strukturi prirodnih brojeva $\mathbb{N}$. Ovo znači da postoji konkretan broj $n \in \mathbb{N}$ takav da je formula $proof(\lceil A \rceil, \overline{n})$ tačna u $\mathbb{N}$, pri čemu je $\overline{n}$ odgovarajući numeral. Imajući u vidu značenje formule $proof(x, y)$ u strukturi $\mathbb{N}$, sledi da ćemo dekodiranjem broja n dobiti dokaz formule A. Dakle, A je takođe dokaziva u $\mathcal{P}$. □

Lema 6.7. *Za svake dve rečenice A i B nad jezikom Peanove aritmetike važi $\vdash_{\mathcal{P}} pr(\lceil A \Rightarrow B \rceil) \Rightarrow pr(\lceil A \rceil) \Rightarrow pr(\lceil B \rceil)$.*

Dokaz. Pretpostavimo $pr(\ulcorner A \Rightarrow B \urcorner)$ i $pr(\ulcorner A \urcorner)$, tj. da važe formule $\exists y.proof(\ulcorner A \Rightarrow B \urcorner, y)$ i $\exists y.proof(\ulcorner A \urcorner, y)$. Primenimo pravilo eliminacije egzistencijalnog kvantifikatora na formule $\exists y.proof(\ulcorner A \Rightarrow B \urcorner, y)$ i $\exists y.proof(\ulcorner A \urcorner, y)$, tj. pretpostavimo da postoji neko z_1 takvo da važi $proof(\ulcorner A \Rightarrow B \urcorner, z_1)$ i neko z_2 takvo da važi $proof(\ulcorner A \urcorner, z_2)$. Sada se polazeći od z_1 i z_2 kao kôdova dokaza formula $A \Rightarrow B$ i A, respektivno, može efektivno izračunati kôd dokaza z koji bi nastao od ovih dokaza primenom pravila eliminacije implikacije. Dokaz ovog izračunavanja, predstavlja dokaz formule $proof(\ulcorner B \urcorner, z)$, odakle se uvođenjem egzistencijalnog kvantifikatora dokazuje formula $\exists y.proof(\ulcorner B \urcorner, y)$ (tj. formula $pr(\ulcorner B \urcorner)$) iz pretpostavki $pr(\ulcorner A \Rightarrow B \urcorner)$ i $pr(\ulcorner A \urcorner)$. Uvođenjem implikacije oslobađaju se pretpostavke i dobija se $\vdash_{\mathcal{P}} pr(\ulcorner A \Rightarrow B \urcorner) \Rightarrow pr(\ulcorner A \urcorner) \Rightarrow pr(\ulcorner B \urcorner)$. □

Lema 6.8. *Neka je $\Psi(x)$ proizvoljna formula koja sadrži slobodnu promenljivu x. Tada postoji formula Φ takva da je $\vdash_{\mathcal{P}} \Phi \Rightarrow \Psi(\ulcorner \Phi \urcorner)$ i $\vdash_{\mathcal{P}} \Psi(\ulcorner \Phi \urcorner) \Rightarrow \Phi$.*

Dokaz. Neka je sa $sub(m, n)$ označen kôd formule koja se dobija tako što u formuli čiji je kôd m sve slobodne promenljive zamenimo numeralom $\overline{n}$. Funkcija sub je efektivno izračunljiva i može se izraziti na jeziku Peanove aritmetike. Specijalno, za bilo koju formulu $A(x)$ sa jednom slobodnom promenljivom x važi da funkcija $sub(\ulcorner A(x) \urcorner, \overline{n})$ izračunava kôd $\ulcorner A(\overline{n}) \urcorner$. Otuda je u Peanovoj aritmetici dokazivo:

$$\vdash_{\mathcal{P}} sub(\ulcorner A(x) \urcorner, \overline{n}) = \ulcorner A(\overline{n}) \urcorner \tag{6.1}$$

Neka je sa $\Delta(x)$ označena formula $\Psi(sub(x, x))$. Neka je n_0 kôd formule $\Delta(x)$. Neka je $\Phi = \Delta(\overline{n}_0) = \Psi(sub(\overline{n}_0, \overline{n}_0)) = \Psi(sub(\ulcorner \Delta(x) \urcorner, \overline{n}_0))$. S obzirom na 6.1, imamo da je:

$$\vdash_{\mathcal{P}} \Psi(sub(\ulcorner \Delta(x) \urcorner, \overline{n}_0)) \Rightarrow \Psi(\ulcorner \Delta(\overline{n}_0) \urcorner)$$

i

$$\vdash_{\mathcal{P}} \Psi(\ulcorner \Delta(\overline{n}_0) \urcorner) \Rightarrow \Psi(sub(\ulcorner \Delta(x) \urcorner, \overline{n}_0))$$

odnosno:

$$\vdash_{\mathcal{P}} \Phi \Rightarrow \Psi(\ulcorner \Phi \urcorner)$$

i

$$\vdash_{\mathcal{P}} \Psi(\ulcorner \Phi \urcorner) \Rightarrow \Phi$$

□

Poslednja lema je poznata i kao *lema o dijagonalizaciji*. Primenom ove leme na formulu $\neg pr(x)$ dobijamo formulu Φ takvu da važi: $\vdash_{\mathcal{P}} \Phi \Rightarrow \neg pr(\ulcorner \Phi \urcorner)$ i $\vdash_{\mathcal{P}} \neg pr(\ulcorner \Phi \urcorner) \Rightarrow \Phi$. Intuitivno, formula Φ sama za sebe tvrdi da je nedokaziva.

Teorema 6.10 (Gedelova teorema o nepotpunosti aritmetike)**.** *Neka je Φ formula takva da važi $\vdash_{\mathcal{P}} \neg pr(\ulcorner \Phi \urcorner) \Rightarrow \Phi$ kao i $\vdash_{\mathcal{P}} \Phi \Rightarrow \neg pr(\ulcorner \Phi \urcorner)$. Tada ni Φ ni $\neg\Phi$ nije dokazivo u $\mathcal{P}$.*

Dokaz. Ako bi Φ bilo dokazivo, tada bi i $pr(\ulcorner \Phi \urcorner)$ bilo dokazivo, na osnovu leme 6.5. Sa druge strane, iz $\vdash_{\mathcal{P}} \Phi \Rightarrow \neg pr(\ulcorner \Phi \urcorner)$ sledi da bismo mogli da dokažemo i $\neg pr(\ulcorner \Phi \urcorner)$ (primenom eliminacije implikacije). Ovako nešto nije moguće, jer znamo da je $\mathcal{P}$ konzistentna. Dakle, Φ ne može biti dokazivo u $\mathcal{P}$.

Obratno, ako bi $\neg\Phi$ bilo dokazivo, tada bi bilo dokazivo i $pr(\ulcorner \neg\Phi \urcorner)$ (lema 6.5). Kako je $\vdash_{\mathcal{P}} \neg\Phi \Rightarrow (\Phi \Rightarrow \bot)$, sledi da je i $\vdash_{\mathcal{P}} pr(\ulcorner \neg\Phi \Rightarrow (\Phi \Rightarrow \bot) \urcorner)$

(na osnovu leme 6.5), pa samim tim i $\vdash_{\mathcal{P}} pr(\lceil\neg\Phi\rceil) \Rightarrow pr(\lceil\Phi \Rightarrow \bot\rceil)$ (lema 6.7). Sada iz dokazivosti $pr(\lceil\neg\Phi\rceil)$ sledi $\vdash_{\mathcal{P}} pr(\lceil\Phi \Rightarrow \bot\rceil)$. Dalje, na osnovu leme 6.7, imamo da je $\vdash_{\mathcal{P}} pr(\lceil\Phi\rceil) \Rightarrow pr(\lceil\bot\rceil)$, odnosno $pr(\lceil\Phi\rceil) \vdash_{\mathcal{P}} pr(\lceil\bot\rceil)$. Dakle, ako bi formula $pr(\lceil\Phi\rceil)$ bila dokaziva, tada bi i formula $pr(\lceil\bot\rceil)$ bila dokaziva u $\mathcal{P}$, što bi, na osnovu leme 6.6, značilo da je i $\bot$ dokaziva u $\mathcal{P}$, što nije moguće, jer je Peanova aritmetika konzistentna. Otuda je $\nvdash_{\mathcal{P}} pr(\lceil\bot\rceil)$, pa je i $\nvdash_{\mathcal{P}} pr(\lceil\Phi\rceil)$. Sa druge strane, Iz svojstva $\vdash_{\mathcal{P}} \neg pr(\lceil\Phi\rceil) \Rightarrow \Phi$ kontrapozicijom sledi $\vdash_{\mathcal{P}} \neg\Phi \Rightarrow pr(\lceil\Phi\rceil)$. Sada iz $\vdash_{\mathcal{P}} \neg\Phi$ sledi $\vdash_{\mathcal{P}} pr(\lceil\Phi\rceil)$. Iz ove kontradikcije sledi da ne može biti $\vdash_{\mathcal{P}} \neg\Phi$. □

Gedelova teorema nam govori da postoje aritmetička tvrđenja koja se ne mogu ni dokazati ni opovrgnuti. Štaviše, za takvu rečenicu A važi da je ili A ili $\neg A$ tačno u standardnom modelu prirodnih brojeva $\mathbb{N}$. Dakle, postoje tvrđenja koja su tačna u intuitivnoj aritmetici, ali se ne mogu formalno dokazati u Peanovoj aritmetici. Štaviše, umesto Peanove aritmetike mogli smo uzeti bilo koju drugu *konzistentnu* aksiomatizaciju koja definiše ,,dovoljno" aritmetike da se izrazi kodiranje formula i dokaza. Jedan primer takve minimalističke aritmetičke teorije je *Robinsonova aritmetika*[1]. Isto važi i za *teoriju skupova* (npr. *Cermelo-Frenkelova teorija* (ZF)) koja takođe dopušta izražavanje aritmetike na svom jeziku.

Neko bi mogao pomisliti da se ovaj problem jednostavno može rešiti tako što formulu Φ iz teoreme 6.10 dodamo u aksiome i time je učinimo dokazivom. Međutim, tako proširena teorija $\mathcal{P}'$ će takođe zadovoljavati potrebne uslove za primenu Gedelove teoreme, pa će i u njoj postojati neka druga rečenica Φ' koja će biti tačna u strukturi prirodnih brojeva, a nedokaziva. Dakle, nije moguće dopuniti skup aksioma tako da teorija postane potpuna, a da ostane konzistentna.

Kao što znamo iz teoreme 6.5, potpunost aritmetike bi povlačila njenu odlučivost. Međutim, čak i kada teorija nije potpuna, ona može biti odlučiva. To nam daje mogućnost da makar prepoznamo tvrđenja koja su u teoriji ostala ,,nerešena", tj. koja nije moguće ni dokazati ni opovrgnuti (prosto bismo procedurom odlučivanja mogli utvrditi da ni A ni $\neg A$ nije teorema u toj teoriji). Na žalost, nekoliko godina nakon što je Gedel prezentovao svoju teoremu o nepotpunosti aritmetike, dokazano je da aritmetika nije ni odlučiva, o čemu govori sledeća teorema.

Teorema 6.11. *Peanova aritmetika je neodlučiva: ne postoji algoritam koji za proizvoljnu formulu A utvrđuje da li je $\vdash_{\mathcal{P}} A$.*

Dokaz. Neka je dat proizvoljan URM program P_e sa kodom e. Na osnovu Klinijeve teoreme o normalnoj formi, funkciju ϕ_e je moguće izraziti u obliku:

$$\phi_e(x) = P(\mu z[Q(e, x, z) = 0])$$

gde su P i Q primitivno rekurzivne funkcije. Kako je Q primitivno rekurzivna, ona se može izraziti na jeziku aritmetike, tj. postoji formula $R_Q(e, x, z)$ koja je tačna u $\mathbb{N}$ ako i samo ako je $Q(e, x, z) = 0$. Sada se program P_e zaustavlja za ulaz x ako i samo ako je formula $\exists z.R_Q(e, x, z)$ teorema Peanove aritmetike. Zaista, ako je $\exists z.R_Q(e, x, z)$ teorema Peanove aritmetike, tada je ona tačna u

[1] Robinsonova aritmetika je konačno aksiomatizovana, jer ne uključuje aksiomatsku šemu indukcije koja nam za dokaz Gedelovih teorema nije neophodna.

svim modelima teorije $\mathcal{P}$, pa i u $\mathbb{N}$. Otuda postoji neko $n \in \mathbb{N}$ takvo da je formula $R_Q(e, x, \overline{n})$ tačna u $\mathbb{N}$, pa za to n važi $Q(e, x, n) = 0$, odakle sledi da je $\phi_e(x)$ definisano, tj. program P_e se zaustavlja za ulaz x. Obrnuto, ako se program P_e zaustavlja za ulaz x, tada će za neko n važiti $Q(e, x, n) = 0$, odakle će formula $R_Q(e, x, \overline{n})$ biti tačna u $\mathbb{N}$. Kako je Q primitivno rekurzivna, postoji dokaz izračunavanja koji izvodi formulu $R_Q(e, x, \overline{n})$. Uvođenjem egzistencijalnog kvantifikatora, dokazujemo formulu $\exists z.R_Q(e, x, z)$.

Dakle, formula $\exists z.R_Q(e, x, z)$ je teoreme Peanove aritmetike ako i samo ako se P_e zaustavlja. Ovo znači da ako bi Peanova aritmetika bila odlučiva, tada bi i problem zaustavljanja bio odlučiv, što znamo da nije tačno. Otuda ni Peanova aritmetika ne može biti odlučiva. □

Napomena 6.14. Na osnovu teoreme 6.5, iz neodlučivosti Peanove aritmetike direktno sledi njena nepotpunost. Na prvi pogled, ispada da je čitav trud oko direktnog dokazivanja Gedelove teoreme bio suvišan, s obzirom da smo nepotpunost mogli dokazati znatno jednostavnije. Ipak, značaj direktnog dokaza Gedelove teoreme je veliki – on nam eksplicitno konstruiše rečenicu koja je nedokaziva u Peanovoj aritmetici. Takođe, istorijski gledano, nepotpunost aritmetike dokazana je pre njene neodlučivosti.

Teorema 6.12 (Druga Gedelova teorema o nepotpunosti)**.** *Konzistentnost Peanove aritmetike nije moguće dokazati u okviru nje same, tj.* $\nvdash_{\mathcal{P}} \neg pr(\lceil \bot \rceil)$.

Dokaz. Za ovo je dovoljno dokazati da iz $\vdash_{\mathcal{P}} \neg pr(\lceil \bot \rceil)$ sledi $\vdash_{\mathcal{P}} \Phi$, gde je Φ formula iz teoreme 6.10 za koju znamo da nije dokaziva. Kao u dokazu teoreme 6.10, pokazuje se da važi $pr(\lceil \Phi \rceil) \vdash_{\mathcal{P}} pr(\lceil \bot \rceil)$, odnosno $\vdash_{\mathcal{P}} pr(\lceil \Phi \rceil) \Rightarrow pr(\lceil \bot \rceil)$. Kontrapozicijom, može se dokazati $\vdash_{\mathcal{P}} \neg pr(\lceil \bot \rceil) \Rightarrow \neg pr(\lceil \Phi \rceil)$. Zbog svojstva $\vdash_{\mathcal{P}} \neg pr(\lceil \Phi \rceil) \Rightarrow \Phi$ formule Φ, sledi da važi $\vdash_{\mathcal{P}} \neg pr(\lceil \bot \rceil) \Rightarrow \Phi$, pa iz dokazivosti formule $\neg pr(\lceil \bot \rceil)$ sledi dokazivost formule Φ. Kako za nju znamo da je nedokaziva, sledi da ni $\neg pr(\lceil \bot \rceil)$ nije dokaziva, tj. nije moguće dokazati nedokazivost kontradikcije u Peanovoj aritmetici u okviru same Peanove aritmetike. □

Dakle, u okviru same aritmetike nije moguće rezonovati o konzistentnosti same aritmetike. Kao i u slučaju prve Gedelove teoreme, i ova teorema važi za svaku konzistentnu teoriju koja je u stanju da na svom jeziku izrazi dovoljno aritmetike. Na primer, ovo važi i za ZFC teoriju skupova. Posledično, konzistentnost čitave matematike nije moguće formalno dokazati.

Glava 7

Složenost izračunavanja

U prethodnim glavama bavili smo se problemom izračunljivosti, gde smo utvrdili da postoje problemi koji su algoritamski nerešivi, tj. za koje ne postoji nada da će ikada biti konstruisan algoritam koji ih rešava. Sa druge strane, za probleme koji *jesu* algoritamski rešivi, postavlja se pitanje *cene* tog rešavanja. Ova cena se izražava u količini potrebnih resursa da se odgovarajuće izračunavanje obavi. Glavni resurs koji se razmatra je *vreme izvršavanja*, koje predstavlja broj koraka koje algoritam izvršava za dati ulaz. Ovaj broj, naravno, zavisi od konkretne ulazne instance. Pritom, za očekivati je da će za „veće" ulazne instance taj broj koraka biti veći. Otuda nam je cilj da vreme izvršavanja izrazimo u funkciji od *veličine ulaza* n (tj. kao $f(n)$).

Tu se javljaju dva problema. Prvi je na koji način definisati veličinu ulaza. Cilj je da veličina ulaza n na neki način opisuje količinu memorijskog prostora koji se koristi za reprezentaciju ulaznog podatka, tj. prostora potrebnog za kodiranje ulazne instance. U zavisnosti od izabrane reprezentacije ulaznog podatka, ovaj prostor može biti manji ili veći. Na primer, ako prirodan broj x na ulazu predstavimo kao niz od $x + 1$ jedinica, tada će veličina ulaza biti $n = x + 1$. Sa druge strane, ako broj x predstavimo u binarnom zapisu (pomoću nula i jedinica) na standardni način, tada će broj potrebnih binarnih cifara biti jednak $n = \lfloor \log_2 x \rfloor + 1$. U realnosti ćemo obično birati ovu drugu reprezentaciju, zato što je prostorno kompaktnija, pa ćemo i složenost obično razmatrati u odnosu na tako definisanu veličinu ulaza. S obzirom da znamo da se svi drugi podaci mogu kodirati prirodnim brojevima, na ovaj način ćemo definisati veličinu ulaza i u slučaju podataka drugih tipova.

Drugi problem je to što čak i za fiksiranu veličinu ulaza n može postojati više različitih instanci problema zadate veličine n. Za različite instance iste veličine broj koraka algoritma može biti različit. Najčešće nas zanima gornja granica potrebnog vremena izvršavanja, tj. zelimo da uspostavimo garanciju da će se za sve instance date veličine n izvršavanje završiti u najviše $f(n)$ koraka. Ovo znači da razmatramo *najgori slučaj* tj. „najtežu" instancu veličine n. Ovakvu gornju granicu nazivamo *vremenska složenost* datog algoritma.

Drugi najznačajniji resurs jeste *prostor*, tj. količina memorije koju algoritam konzumira tokom svog rada za dati ulaz. I ovde se potrebni memorijski prostor izražava kao funkcija od veličine ulaza n, tj. veličine memorijskog prostora koji konzumira sam ulazni podatak. Takođe, po pravilu ćemo razmatrati najgori mogući slučaj za datu veličinu ulaza n, tj. uspostavljaćemo gornju granicu $g(n)$

za prostor potreban da se izvrši algoritam za proizvoljnu instancu veličine n. Ovakva gornja granica se naziva *prostorna složenost* datog algoritma.

Dati problem se često može rešiti na više načina, tj. postoji više različitih algoritama koji rešavaju isti problem. Složenosti ovih algoritama mogu biti različite, a obično smo zainteresovani za najefikasnije moguće rešenje, tj. za algoritam najmanje moguće složenosti. U tom smislu, probleme svrstavamo u *klase složenosti* u zavisnosti od složenosti najefikasnijeg algoritma kojim se mogu rešiti. Oblast teorijskog računarstva koja razmatra klase složenosti i odnose među njima naziva se *teorija složenosti izračunavanja*. U ovoj glavi iznosimo neke najpoznatije rezultate iz ove oblasti.

Naglasimo da postoji i srodna oblast računarstva koja se naziva *analiza algoritama*. Ona se bavi preciznim razmatranjem složenosti konkretnih algoritama i njihovim međusobnim poređenjem. Pritom, analiza koja se tu sprovodi je obično znatno preciznija, jer nam daje odgovor na pitanje da li je, na primer, nekom algoritmu potrebno n^2 ili $n \log n$ koraka za izvršavanje za ulaze veličine n. Takođe, prilikom praktične analize algoritama, često razmatramo i *prosečno vreme izvršavanja* za ulaze date veličine n. U ovom tekstu, takvom analizom se nećemo baviti.

Sa druge strane, klase složenosti koje razmatramo u okviru složenosti izračunavanja su obično znatno grublje. Na primer, pitamo se da li za neki problem postoji algoritam *polinomske složenosti*, tj. algoritam složenosti $p(n)$, gde je $p(n)$ neki polinom po n. Pritom, nebitno je da li je, na primer, $p(n) = n^2$ ili $p(n) = n^{10}$. Dakle, cilj složenosti izračunavanja je da se problemi relativno grubo klasifikuju u one koji su *praktično rešivi* i one koji to nisu, iako se formalno mogu algoritamski rešiti. U praktično rešive probleme obično ubrajamo one za koje postoji algoritam polinomske složenosti. Intuitivni razlog za tako nešto je sledeći: za algoritme polinomske složenosti možemo očekivati da ćemo uz razumno povećanje datih resursa moći značajno da povećamo veličinu instanci problema koje možemo rešavati. Na primer, ako je složenost algoritma n^2, tada ćemo, ako se računar ubrza 100 puta (što je razumno očekivanje, s obzirom na razvoj tehnologije), moći da za isto dato vreme rešimo instance koje su 10 puta veće. Sa druge strane, ako je složenost algoritma 2^n (tzv. *eksponencijalna složenost*), tada ćemo sa 100 puta bržim računarom za isto zadato vreme moći da rešimo instance koje su svega za 6 veće (jer je već $2^7 = 128$). Zbog toga je za očekivati da će daljim razvojem tehnologije problemi rešivi algoritmima polinomske složenosti biti sve pristupačniji za rešavanje pomoću realnih računara, dok to nije slučaj za probleme koji nisu rešivi algoritmima polinomske složenosti (za takve probleme se obično kaže da su *neukrotivi*, engl. *intractable*). Zato nam je od najvećeg značaja da izučimo klasu polinomski rešivih problema, kao i kako se ta klasa odnosi sa drugim klasama složenosti.

7.1 Tjuringova mašina

Da bismo formalno definisali pojam složenosti algoritma, potrebno je da unapred fiksiramo formalni model izračunavanja. U dosadašnjem razmatranju, izučili smo dva takva formalizma – μ-rekurzivne funkcije i URM programe. Na žalost, ni jedan od ta dva formalizma nije pogodan za definisanje pojma složenosti izračunavanja. Naime, oba ova modela izračunavanja su definisani u terminima aritmetičkih operacija nad prirodnim brojevima, pri čemu se pretpo-

stavlja da resursi za izvođenje pojedinačnih operacija nad brojevima (poput Z i S operacija) ne zavise od vrednosti samog broja nad kojim se primenjuju. Na primer, kod URM programa, pretpostavljamo da svaki registar može da čuva bilo koji prirodan broj. Ovo prilično odudara od realnih računara, kod kojih ni jedan registar ne može čuvati bilo koji prirodan broj, već samo prirodne brojeve iz nekog unapred datog konačnog skupa (npr. 8-bitni registar može čuvati samo brojeve od 0 do 255). S obzirom da znamo da se svaki podatak može na kraju kodirati jednim prirodnim brojem, sledi da se kod URM programa svaki podatak može smestiti u jedan registar, što nam ne omogućava da realno odredimo veličinu ulaza. Takođe, URM program pretpostavlja da se svaki prirodan broj može uvećati za 1 pomoću jedne S instrukcije. U praksi, mi brojeve predstavljamo nizom njihovih cifara, pri čemu aritmetičke operacije nad njima mogu zahtevati transformisanje više cifara u zapisu. Na primer, uvećanje broja 199 za 1 zahteva promenu tri cifre, dok uvećanje broja 1999999 za 1 zahteva promenu sedam cifara. Dakle, URM programi ne mogu realno opisati vremensku složenost postupka izračunavanja. Najzad, URM program izračunava zbir dva broja x i y tako što uzastopno uvećava broj x y puta. Realni računari ovo rade znatno efikasnije, s obzirom da operišu sa pozicionim zapisima brojeva. Slično važi i za ostale osnovne aritmetičke operacije.

Da bismo prevazišli nabrojane probleme, uvodimo još jedan formalizam za opis algoritama – *Tjuringove mašine*. U pitanju je jedan od najstarijih formalnih modela izračunavanja koji je Alan Tjuring (slika 7.1) osmislio kako bi dokazao neodlučivost logike prvog reda. Princip rada Tjuringove mašine podseća na rad pisaće mašine i spada u formalizme najnižeg nivoa, s obzirom da ne barata sa brojevima, već sa simbolima pomoću kojih se podaci zapisuju. Zbog toga su Tjuringove mašine naročito pogodne za razmatranje složenosti izračunavanja, jer omogućavaju da se realno ocene veličine podataka sa kojima se radi, kao i vreme koje je potrebno za njihovu obradu.

Slika 7.1: Britanski matematičar Alan Turing (1912-1954)

Da bismo definisali pojam Tjuringove mašine, najpre uvodimo definiciju azbuke, reči i jezika.

Definicija 7.1. *Azbuka* (ili *alfabet*) $\Sigma = \{a_1, \ldots, a_n\}$ je konačan skup *simbola*. *Reč* nad azbukom Σ je bilo koji konačan niz $a_{i_1} a_{i_2} \ldots a_{i_k}$ simbola iz Σ. *Dužina reči* $w = a_{i_1} \ldots a_{i_k}$ (u oznaci $|w|$) je jednaka broju simbola u njoj. Specijalno,

prazna reč ε je reč koja ne sadrži ni jedan simbol azbuke i njena dužina je $|\varepsilon| = 0$. Skup svih reči nad Σ označavamo sa Σ^*. *Jezik* L nad azbukom Σ je bilo koji podstkup od Σ^*.

Primer 7.1. Neka je data azbuka $\Sigma = \{0, 1\}$. Reči nad ovom azbukom su binarne niske, poput 01101 ili 111011. Pritom, imamo da je $|01101| = 5$ i $|111011| = 6$. Reč ε je prazna niska i ona je takođe reč nad Σ.

Skup $L = \{w \in \Sigma^* \mid w \text{ sadrži paran broj nula}\}$ je jedan jezik nad Σ. Na primer, $01101 \in L$, a $111011 \notin L$. Prazna reč ε takođe pripada L, s obzirom da ne sadrži ni jednu nulu.

Definicija 7.2. *Tjuringova mašina sa* k *traka* je uređena sedmorka $(\Gamma, b, \Sigma, Q, q_0, F, \delta)$, gde je:

- Γ je *azbuka trake*
- $\square \in \Gamma$ je *simbol praznog polja*
- $\Sigma \subseteq (\Gamma \setminus \{\square\})$ je *ulazna azbuka*
- Q je konačan skup *stanja*
- $q_0 \in Q$ je *početno stanje*
- $F \subseteq Q$ je skup *završnih stanja*
- $\delta : (Q \setminus F) \times \Gamma^k \to Q \times (\Gamma \times \{\leftarrow, \rightarrow, \downarrow\})^k$ je *funkcija prelaska*

Konfiguracija Tjuringove mašine se može opisati kao $(k+1)$-torka $(q, \square^\omega u_1 \widehat{a_1} w_1 \square^\omega, \ldots, \square^\omega u_k \widehat{a_k} w_k \square^\omega)$, gde je $q \in Q$ trenutno stanje, $\square^\omega u_i \widehat{a_i} w_i \square^\omega$ je konfiguracija i-te trake ($u_i, w_i \in \Gamma^*$, $a_i \in \Gamma$, $\square^\omega = \ldots\square\square\square\ldots$, a oznaka $\widehat{\ }$ označava poziciju *glave za čitanje/pisanje* i-te trake). Početna konfiguracija za dati *ulaz* $w \in \Sigma^*$ je data sa $(q_0, \square^\omega \widehat{a} u \square^\omega, \square^\omega \widehat{\square} \square^\omega, \ldots, \square^\omega \widehat{\square} \square^\omega)$, gde je $w = au$ ($a \in \Sigma$, $w \in \Sigma^*$). Ukoliko je tekuća konfiguracija $(q, \square^\omega u_1 b_1 \widehat{a_1} c_1 w_1 \square^\omega, \ldots, \square^\omega u_k b_k \widehat{a_k} c_k w_k \square^\omega)$ ($q \in Q \setminus F, u_i, w_i \in \Gamma^*, a_i, b_i, c_i \in \Gamma$) i važi $\delta(q, a_1, \ldots, a_k) = (q', ((e_1, d_1), \ldots, (e_k, d_k)))$, gde je $q' \in Q$, $e_i \in \Gamma$, $d_i \in \{\leftarrow, \rightarrow, \downarrow\}$), tada će sledeća konfiguracija mašine biti $(q', \square^\omega u_1 b_1 e_1 c_1 w_1 \square^\omega, \ldots, \square^\omega u_k b_k e_k c_k w_k \square^\omega)$, pri čemu će pozicija i-te glave biti iznad e_i ako je $d_i = \downarrow$, iznad b_i ako je $d_i = \leftarrow$, ili iznad c_i ako je $d_i = \rightarrow$. Konfiguracija $(q, \square^\omega u_1 \widehat{a_1} w_1 \square^\omega, \ldots, \square^\omega u_k \widehat{a_k} w_k \square^\omega)$ je *završna*, ako $q \in F$. Tjuringova mašina *staje* za dati ulaz $w \in \Sigma^*$ ako polazeći iz početne konfiguracije koja odgovara tom ulazu nakon konačno mnogo koraka stiže u neku završnu konfiguraciju. *Izlaz* Tjuringove mašine je niska $w_{out} \in \Sigma^*$ koja se dobija kada se iz niske $u_k a_k w_k \in \Gamma^*$ zapisane na k-toj traci u završnoj konfiguraciji mašine obrišu svi simboli koji ne pripadaju azbuci Σ.

Intuitivno, Tjuringova mašina ima k beskonačnih memorijskih traka koje se sastoje iz polja u koja možemo upisivati simbole iz Γ. Pritom, u svakom trenutku samo konačan broj polja svake trake nije *prazno*, tj. sadrži simbole iz Γ koji nisu $\square$. Svaka traka ima glavu za čitanje/pisanje koja je pozicionirana na nekom polju trake. Prva traka je *ulazna traka* i na njoj se inicijalno nalazi ulazna niska $w \in \Sigma^*$. Poslednja traka je *izlazna traka* i sa nje se na kraju očitava izlaz. *Program* Tjuringove mašine je određen funkcijom prelaska δ. Ona nam govori na koji način menjamo konfiguraciju u svakom koraku. Promena konfiguracije

se vrši na osnovu simbola koji se trenutno nalaze ispod glava za čitanje, kao i na osnovu trenutnog stanja programa (stanje $q \in Q$). Svaki korak menja konfiguraciju mašine tako što menja stanje q mašine u $q' \in Q$, zamenjuje simbol a_i ispod svake od glava za čitanje/pisanje novim simbolom e_i, a zatim glavu eventualno pomera za jedno mesto u levo ili desno. Rad programa se završava kada se dođe u neko završno stanje. Izlaz se sa poslednje trake očitava tako što se sa nje uzmu samo simboli koji pripadaju Σ, dok se simboli iz $\Gamma \setminus \Sigma$ preskaču. Naime, Σ predstavlja ulazno/izlaznu azbuku pomoću koje mašina komunicira sa spoljnom sredinom. Sa druge strane, radna azbuka Γ može pored simbola iz Σ koristiti i razne pomoćne simbole koje zapisuje na trakama, ali oni neće biti prisutni na izlazu nakon završetka rada mašine.

Primer 7.2. Neka je data azbuka $\Sigma = \{a, b\}$. Konstruišimo Tjuringovu mašinu koja proizvoljnu nisku $w \in \Sigma^*$ transformiše tako što slova a stavlja na početak, a slova b na kraj. Na primer, za ulaznu nisku *abbaabbab*, izlaz će biti *aaaabbbbb*. Azbuka Γ će u ovom primeru biti jednaka $\Sigma \cup \{\square\}$, jer nam, osim blanko-simbola $\square$ drugi pomoćni simboli nisu potrebni. Imaćemo dve trake, ulaznu i izlaznu. Na ulaznoj traci će biti inicijalno smeštena ulazna niska, a glava će biti pozicionirana nad početnim karakterom niske. Izlazna traka će inicijalno biti prazna, a glava će biti pozicionirana iznad proizvoljnog praznog polja. Program će se sastojati iz sledećih ,,naredbi":

- $\delta(q_0, a, \square) = (q_0, (a, \rightarrow), (a, \rightarrow))$
- $\delta(q_0, b, \square) = (q_0, (b, \rightarrow), (\square, \downarrow))$
- $\delta(q_0, \square, \square) = (q_1, (\square, \leftarrow), (\square, \downarrow))$
- $\delta(q_1, a, \square) = (q_1, (a, \leftarrow), (\square, \downarrow))$
- $\delta(q_1, b, \square) = (q_1, (b, \leftarrow), (b, \rightarrow))$
- $\delta(q_1, \square, \square) = (q_F, (\square, \downarrow), (\square, \downarrow))$

Pritom, q_0 je početno, a q_F završno stanje mašine. Intuitivno, program radi tako što najpre prolazi kroz ulaznu nisku sa leva na desno i sve simbole a prepisuje na izlaznu traku, dok simbole b preskače. Kada stigne do kraja niske, mašina prelazi u stanje q_1 u kome se kreće sa desna u levo i tom prilikom sve simbole b prepisuje na izlaznu traku, dok simbole a preskače. Na primer, za nisku *abbaa* bi iz početne konfiguracije $(q_0, \square^\omega \widehat{a}bbaa\square^\omega, \square^\omega \widehat{\square}\square^\omega)$ imali sledeći niz konfiguracija:

$$
\begin{array}{ll}
(q_0, \square^\omega \widehat{a}bbaa\square^\omega, \square^\omega \widehat{\square}\square^\omega) & \Longrightarrow \\
(q_0, \square^\omega a\widehat{b}baa\square^\omega, \square^\omega a\widehat{\square}\square^\omega) & \Longrightarrow \\
(q_0, \square^\omega ab\widehat{b}aa\square^\omega, \square^\omega a\widehat{\square}\square^\omega) & \Longrightarrow \\
(q_0, \square^\omega abb\widehat{a}a\square^\omega, \square^\omega a\widehat{\square}\square^\omega) & \Longrightarrow \\
(q_0, \square^\omega abba\widehat{a}\square^\omega, \square^\omega aa\widehat{\square}\square^\omega) & \Longrightarrow \\
(q_0, \square^\omega abbaa\widehat{\square}\square^\omega, \square^\omega aaa\widehat{\square}\square^\omega) & \Longrightarrow \\
(q_1, \square^\omega abba\widehat{a}\square^\omega, \square^\omega aaa\widehat{\square}\square^\omega) & \Longrightarrow \\
(q_1, \square^\omega abb\widehat{a}a\square^\omega, \square^\omega aaa\widehat{\square}\square^\omega) & \Longrightarrow \\
(q_1, \square^\omega ab\widehat{b}aa\square^\omega, \square^\omega aaa\widehat{\square}\square^\omega) & \Longrightarrow \\
(q_1, \square^\omega a\widehat{b}baa\square^\omega, \square^\omega aaab\widehat{\square}\square^\omega) & \Longrightarrow \\
(q_1, \square^\omega \widehat{a}bbaa\square^\omega, \square^\omega aaabb\widehat{\square}\square^\omega) & \Longrightarrow \\
(q_1, \square^\omega \widehat{\square}abbaa\square^\omega, \square^\omega aaabb\widehat{\square}\square^\omega) & \Longrightarrow \\
(q_F, \square^\omega \widehat{\square}abbaa\square^\omega, \square^\omega aaabb\widehat{\square}\square^\omega) &
\end{array}
$$

Odbacivanjem simbola $\square$ iz završne konfiguracije izlazne trake, dobijamo izlaznu nisku $aaabb$, kao što je i očekivano.

Napomena 7.1. Tjuringova mašina može imati proizvoljan, unapred fiksiran broj traka. Broj traka može biti i jedan – u tom slučaju imamo Tjuringovu mašinu sa jednom trakom koja će ujedno biti i ulazna i izlazna. Originalna definicija Tjuringove mašine je podrazumevala da postoji samo jedna traka. Može se pokazati da veći broj traka ne utiče na moć izračunavanja, jer se svaka Tjuringova mašina sa k traka može simulirati pomoću Tjuringove mašine sa jednom trakom. Ipak, postojanje više traka može značajno da olakša opis postupka izračunavanja, jer nam omogućava da pristupamo većem broju podataka istovremeno.

Primer 7.3. Rešimo isti zadatak koji smo imali u prethodnom primeru, ali ovoga puta pomoću Tjuringove mašine sa jednom trakom. Ideja je da izlaz upisujemo ispred ulaza na istoj traci. Zato ćemo u azbuci Γ pored simbola a, b i $\square$ imati još jedan simbol $|$ koji predstavlja *separator* između ulaza i izlaza. Program će sada izgledati ovako:

- $\delta(q_0, a) = (q_0, (a, \leftarrow))$
- $\delta(q_0, b) = (q_0, (b, \leftarrow))$
- $\delta(q_0, \square) = (q_b, (|, \rightarrow))$

Dakle, u stanju q_0 se pomeramo levo od ulaza i ispred ulaza upisujemo separator $|$ i prelazimo u stanje q_b.

Sada je ideja da prođemo kroz ulaznu nisku i sve simbole b prepišemo ispred separatora $|$. Da bismo vodili evidenciju koje smo simbole b prepisali, a koje nismo, uvodimo pomoćni simbol B u azbuku Γ koji predstavlja *marker* prepisanih slova b. U stanju q_b se krećemo sa leva u desno, preskačemo slova a i markere B i stajemo kod prvog neprepisanog slova b. Njega zamenjujemo markerom B i prelazimo u stanje q_{bw} u kome nam je cilj da to b prepišemo na izlaz. Ako ne pronađemo ni jedno b do kraja ulazne niske, prelazimo u stanje q_r.

- $\delta(q_b, a) = (q_b, (a, \rightarrow))$
- $\delta(q_b, B) = (q_b, (B, \rightarrow))$

- $\delta(q_b, b) = (q_{bw}, (B, \leftarrow))$
- $\delta(q_b, \square) = (q_r, (\square, \leftarrow))$

Stanje q_{bw} služi da se pronađeni simbol b upiše levo od separatora.

- $\delta(q_{bw}, *) = (q_{bw}, (*, \leftarrow))$
- $\delta(q_{bw}, \square) = (q_{sb}, (b, \rightarrow))$

U ovom stanju za sve simbole koji nisu $\square$ (što je gore označeno sa $*$, da ne bismo ponavljali istu naredbu za sve simbole), se samo pomeramo u levo, a kada stignemo do $\square$, zamenjujemo ga sa b i prelazimo u stanje q_{sb}. U ovom stanju je cilj da ponovo dođemo na početak ulazne niske i da nastavimo sa pretragom sledećeg simbola b.

- $\delta(q_{sb}, *) = (q_{sb}, (*, \rightarrow))$
- $\delta(q_{sb}, |) = (q_b, (|, \rightarrow))$

pri čemu je sa $*$ označen bilo koji simbol osim separatora $|$.

Kada stignemo u stanje q_r, znači da smo prepisali sve b simbole na izlaz. Sada je potrebno levo od njih na izlazu prepisati sva slova a iz ulazne niske. Najpre je potrebno vratiti se na početak ulaza:

- $\delta(q_r, *) = (q_r, (*, \leftarrow))$
- $\delta(q_r, |) = (q_a, (|, \rightarrow))$

pri čemu ovde $*$ označava sve simbole osim separatora $|$. U stanju q_a tražimo prvi simbol a iz ulazne niske:

- $\delta(q_a, B) = (q_a, (B, \rightarrow))$
- $\delta(q_a, A) = (q_a, (A, \rightarrow))$
- $\delta(q_a, a) = (q_{aw}, (A, \leftarrow))$
- $\delta(q_a, \square) = (q_c, \leftarrow))$

Kao i ranije, uvodimo marker A u azbuku Γ da označimo one simbole a koji su već prepisani na izlaz. Njih preskačemo, kao i markere B. Kada naiđemo na simbol a, zamenjujemo ga markerom A i prelazimo u stanje q_{aw} koje služi za upis pročitanog simbola a na izlaz. Ako nije preostalo ni jedno neprepisano a, tada ćemo stići do praznog polja. U tom slučaju prelazimo u stanje q_c čiji je zadatak da obriše sa trake sve osim onoga što treba da ostane na izlazu.

- $\delta(q_{aw}, *) = (q_{aw}, (*, \leftarrow))$
- $\delta(q_{aw}, \square) = (q_{sa}, (a, \rightarrow))$

Ovde ponovo $*$ označava sve osim $\square$. Kada naiđemo na prvo prazno polje na levom kraju, u njega upisujemo a i zatim prelazimo u stanje q_{sa} koje služi da se vratimo na početak ulaza i tražimo sledeće a.

- $\delta(q_{sa}, *) = (q_{sa}, (*, \rightarrow))$

- $\delta(q_{sa}, |) = (q_a, (|, \rightarrow))$

pri čemu $*$ označava sve osim $|$. Najzad, u stanju q_c radimo sledeće:

- $\delta(q_c, *) = (q_c, (\square, \leftarrow))$
- $\delta(q_c, |) = (q_F, (\square, \downarrow))$

Dakle, sve simbole osim $|$ brišemo i pomeramo se dalje u levo. Kada stignemo do $|$, njega takođe brišemo i prelazimo u završno stanje q_F. Za nisku $abbaa$ imaćemo sledeći niz konfiguracija:

$$
\begin{array}{lllll}
(q_0, \square^\omega \widehat{a}bbaa\square^\omega) & \Longrightarrow & (q_0, \square^\omega \widehat{\square}abbaa\square^\omega) & \Longrightarrow & (q_b, \square^\omega|\widehat{a}bbaa\square^\omega) \Longrightarrow \\
(q_b, \square^\omega|\widehat{ab}baa\square^\omega) & \Longrightarrow & (q_{bw}, \square^\omega|\widehat{a}Bbaa\square^\omega) & \Longrightarrow & (q_{bw}, \square^\omega\widehat{|}aBbaa\square^\omega) \Longrightarrow \\
(q_{bw}, \square^\omega\widehat{\square}|aBbaa\square^\omega) & \Longrightarrow & (q_{sb}, \square^\omega\widehat{b}|aBbaa\square^\omega) & \Longrightarrow & (q_b, \square^\omega b|\widehat{a}Bbaa\square^\omega) \Longrightarrow \\
(q_b, \square^\omega b|a\widehat{B}baa\square^\omega) & \Longrightarrow & (q_b, \square^\omega b|a\widehat{Bb}aa\square^\omega) & \Longrightarrow & (q_{bw}, \square^\omega b|a\widehat{B}Baa\square^\omega) \Longrightarrow \\
(q_{bw}, \square^\omega b|\widehat{a}BBaa\square^\omega) & \Longrightarrow & (q_{bw}, \square^\omega\widehat{b}|aBBaa\square^\omega) & \Longrightarrow & (q_{bw}, \square^\omega\widehat{b}|aBBaa\square^\omega) \Longrightarrow \\
(q_{bw}, \square^\omega\widehat{\square}b|aBBaa\square^\omega) & \Longrightarrow & (q_{sb}, \square^\omega\widehat{b\square}|aBBaa\square^\omega) & \Longrightarrow & (q_{sb}, \square^\omega b\widehat{b}|aBBaa\square^\omega) \Longrightarrow \\
(q_b, \square^\omega bb|\widehat{a}BBaa\square^\omega) & \Longrightarrow & (q_b, \square^\omega bb|a\widehat{B}Baa\square^\omega) & \Longrightarrow & (q_b, \square^\omega bb|aB\widehat{B}aa\square^\omega) \Longrightarrow \\
(q_b, \square^\omega bb|aBB\widehat{a}a\square^\omega) & \Longrightarrow & (q_b, \square^\omega bb|aBBa\widehat{a}\square^\omega) & \Longrightarrow & (q_b, \square^\omega bb|aBBaa\widehat{\square}\square^\omega) \Longrightarrow \\
(q_r, \square^\omega bb|aBBa\widehat{a}\square^\omega) & \Longrightarrow & (q_r, \square^\omega bb|aBB\widehat{a}a\square^\omega) & \Longrightarrow & (q_r, \square^\omega bb|aB\widehat{B}aa\square^\omega) \Longrightarrow \\
(q_r, \square^\omega bb|a\widehat{B}Baa\square^\omega) & \Longrightarrow & (q_r, \square^\omega bb|\widehat{a}BBaa\square^\omega) & \Longrightarrow & (q_r, \square^\omega bb\widehat{|}aBBaa\square^\omega) \Longrightarrow \\
(q_a, \square^\omega bb|\widehat{a}BBaa\square^\omega) & \Longrightarrow & (q_{aw}, \square^\omega bb\widehat{|}ABBaa\square^\omega) & \Longrightarrow & (q_{aw}, \square^\omega b\widehat{b}|ABBaa\square^\omega) \Longrightarrow \\
(q_{aw}, \square^\omega\widehat{bb}|ABBaa\square^\omega) & \Longrightarrow & (q_{aw}, \square^\omega\widehat{\square}bb|ABBaa\square^\omega) & \Longrightarrow & (q_{sa}, \square^\omega a\widehat{b}b|ABBaa\square^\omega) \Longrightarrow \\
(q_{sa}, \square^\omega ab\widehat{b}|ABBaa\square^\omega) & \Longrightarrow & (q_{sa}, \square^\omega abb\widehat{|}ABBaa\square^\omega) & \Longrightarrow & (q_a, \square^\omega abb|\widehat{A}BBaa\square^\omega) \Longrightarrow \\
(q_a, \square^\omega abb|A\widehat{B}Baa\square^\omega) & \Longrightarrow & (q_a, \square^\omega abb|AB\widehat{B}aa\square^\omega) & \Longrightarrow & (q_a, \square^\omega abb|ABB\widehat{a}a\square^\omega) \Longrightarrow \\
(q_{aw}, \square^\omega abb|AB\widehat{B}Aa\square^\omega) & \Longrightarrow & (q_{aw}, \square^\omega abb|A\widehat{B}BAa\square^\omega) & \Longrightarrow & (q_{aw}, \square^\omega abb|\widehat{A}BBAa\square^\omega) \Longrightarrow \\
(q_{aw}, \square^\omega abb\widehat{|}ABBAa\square^\omega) & \Longrightarrow & (q_{aw}, \square^\omega ab\widehat{b}|ABBAa\square^\omega) & \Longrightarrow & (q_{aw}, \square^\omega a\widehat{b}b|ABBAa\square^\omega) \Longrightarrow \\
(q_{aw}, \square^\omega\widehat{a}bb|ABBAa\square^\omega) & \Longrightarrow & (q_{aw}, \square^\omega\widehat{\square}abb|ABBAa\square^\omega) & \Longrightarrow & (q_{sa}, \square^\omega a\widehat{a}bb|ABBAa\square^\omega) \Longrightarrow \\
(q_{sa}, \square^\omega aa\widehat{b}b|ABBAa\square^\omega) & \Longrightarrow & (q_{sa}, \square^\omega aab\widehat{b}|ABBAa\square^\omega) & \Longrightarrow & (q_{sa}, \square^\omega aabb\widehat{|}ABBAa\square^\omega) \Longrightarrow \\
(q_a, \square^\omega aabb|\widehat{A}BBAa\square^\omega) & \Longrightarrow & (q_a, \square^\omega aabb|A\widehat{B}BAa\square^\omega) & \Longrightarrow & (q_a, \square^\omega aabb|AB\widehat{B}Aa\square^\omega) \Longrightarrow \\
(q_a, \square^\omega aabb|ABB\widehat{A}a\square^\omega) & \Longrightarrow & (q_a, \square^\omega aabb|ABBA\widehat{a}\square^\omega) & \Longrightarrow & (q_{aw}, \square^\omega aabb|ABB\widehat{A}A\square^\omega) \Longrightarrow \\
(q_{aw}, \square^\omega aabb|AB\widehat{B}AA\square^\omega) & \Longrightarrow & (q_{aw}, \square^\omega aabb|A\widehat{B}BAA\square^\omega) & \Longrightarrow & (q_{aw}, \square^\omega aabb|\widehat{A}BBAA\square^\omega) \Longrightarrow \\
(q_{aw}, \square^\omega aabb\widehat{|}ABBAA\square^\omega) & \Longrightarrow & (q_{aw}, \square^\omega aab\widehat{b}|ABBAA\square^\omega) & \Longrightarrow & (q_{aw}, \square^\omega aa\widehat{b}b|ABBAA\square^\omega) \Longrightarrow \\
(q_{aw}, \square^\omega a\widehat{a}bb|ABBAA\square^\omega) & \Longrightarrow & (q_{aw}, \square^\omega\widehat{a}abb|ABBAA\square^\omega) & \Longrightarrow & (q_{aw}, \square^\omega\widehat{\square}aabb|ABBAA\square^\omega) \Longrightarrow \\
(q_{sa}, \square^\omega a\widehat{a}abb|ABBAA\square^\omega) & \Longrightarrow & (q_{sa}, \square^\omega aa\widehat{a}bb|ABBAA\square^\omega) & \Longrightarrow & (q_{sa}, \square^\omega aaa\widehat{b}b|ABBAA\square^\omega) \Longrightarrow \\
(q_{sa}, \square^\omega aaab\widehat{b}|ABBAA\square^\omega) & \Longrightarrow & (q_{sa}, \square^\omega aaabb\widehat{|}ABBAA\square^\omega) & \Longrightarrow & (q_a, \square^\omega aaabb|\widehat{A}BBAA\square^\omega) \Longrightarrow \\
(q_a, \square^\omega aaabb|A\widehat{B}BAA\square^\omega) & \Longrightarrow & (q_a, \square^\omega aaabb|AB\widehat{B}AA\square^\omega) & \Longrightarrow & (q_a, \square^\omega aaabb|ABB\widehat{A}A\square^\omega) \Longrightarrow \\
(q_a, \square^\omega aaabb|ABBA\widehat{A}\square^\omega) & \Longrightarrow & (q_a, \square^\omega aaabb|ABBAA\widehat{\square}\square^\omega) & \Longrightarrow & (q_c, \square^\omega aaabb|ABBA\widehat{A}\square^\omega) \Longrightarrow \\
(q_c, \square^\omega aaabb|ABB\widehat{A}\square^\omega) & \Longrightarrow & (q_c, \square^\omega aaabb|AB\widehat{B}\square^\omega) & \Longrightarrow & (q_c, \square^\omega aaabb|A\widehat{B}\square^\omega) \Longrightarrow \\
(q_c, \square^\omega aaabb|\widehat{A}\square^\omega) & \Longrightarrow & (q_c, \square^\omega aaabb\widehat{|}\square^\omega) & \Longrightarrow & (q_F, \square^\omega aaabb\widehat{\square}\square^\omega) \Longrightarrow
\end{array}
$$

Primer 7.4. Razmotrimo Tjuringovu mašinu koja izračunava zbir dva broja koji su predstavljeni svojim binarnim zapisima. Dakle, azbuka će biti $\Sigma = \{0, 1, |\}$, pri čemu ćemo simbol $|$ koristiti kao separator između zapisa dva sabirka na ulazu. Na primer, ako želimo da saberemo brojeve 3 i 5, na ulazu ćemo imati nisku 11|101. Tjuringova mašina će imati tri trake. Na prvoj traci će inicijalno biti ulaz, a zatim ćemo prvi sabirak prebaciti na drugu traku, dok će na prvoj traci ostati samo drugi sabirak. Nakon toga ćemo oba sabirka obilaziti sa desna u levo, simulirajući uobičajen postupak sabiranja binarnih broja i zapisujući cifre zbira na trećoj, izlaznoj traci. Informacije o prenosu ćemo čuvati u stanju Tjuringove mašine. Formalno, imaćemo da je $\Gamma = \Sigma \cup \{\square\}$, a program mašine će izgledati ovako:

- $\delta(q_0, 0, \square, \square) = (q_0, (\square, \rightarrow), (0, \rightarrow), (\square, \downarrow))$

- $\delta(q_0, 1, \square, \square) = (q_0, (\square, \rightarrow), (1, \rightarrow), (\square, \downarrow))$
- $\delta(q_0, |, \square, \square) = (q_1, (\square, \rightarrow), (\square, \downarrow), (\square, \downarrow))$
- $\delta(q_1, 0, \square, \square) = (q_1, (0, \rightarrow), (\square, \downarrow), (\square, \downarrow))$
- $\delta(q_1, 1, \square, \square) = (q_1, (1, \rightarrow), (\square, \downarrow), (\square, \downarrow))$
- $\delta(q_1, \square, \square, \square) = (q_n, (\square, \leftarrow), (\square, \leftarrow), (\square, \downarrow))$

Ovim delom programa vršimo kopiranje prvog sabirka na drugu traku (stanje q_0), a zatim premotavanje prve trake na desni kraj drugog sabirka (stanje q_1). Na kraju stižemo u stanje q_n u kome započinjemo sabiranje.

- $\delta(q_n, 0, 0, \square) = (q_n, (0, \leftarrow), (0, \leftarrow), (0, \leftarrow))$
- $\delta(q_n, 0, 1, \square) = (q_n, (0, \leftarrow), (1, \leftarrow), (1, \leftarrow))$
- $\delta(q_n, 1, 0, \square) = (q_n, (1, \leftarrow), (0, \leftarrow), (1, \leftarrow))$
- $\delta(q_n, 1, 1, \square) = (q_c, (1, \leftarrow), (1, \leftarrow), (0, \leftarrow))$
- $\delta(q_c, 0, 0, \square) = (q_n, (0, \leftarrow), (0, \leftarrow), (1, \leftarrow))$
- $\delta(q_c, 0, 1, \square) = (q_c, (0, \leftarrow), (1, \leftarrow), (0, \leftarrow))$
- $\delta(q_c, 1, 0, \square) = (q_c, (1, \leftarrow), (0, \leftarrow), (0, \leftarrow))$
- $\delta(q_c, 1, 1, \square) = (q_c, (1, \leftarrow), (1, \leftarrow), (1, \leftarrow))$

Ovaj deo programa vrši sabiranje binarnih cifara i pomeranje u levo na sledeće cifre u zapisima brojeva. Stanje q_n znači da ne postoji prenos sa prethodne pozicije, dok stanje q_c znači da taj prenos postoji. Ostaje da se obradi slučaj kada jedan od sabiraka ima manje cifara od drugog – tada ćemo u nekom trenutku stići do praznog polja u jednom sabirku, dok će drugi imati još cifara.

- $\delta(q_n, \square, 0, \square) = (q_n, (\square, \downarrow), (0, \leftarrow), (0, \leftarrow))$
- $\delta(q_n, 0, \square, \square) = (q_n, (0, \leftarrow), (\square, \downarrow), (0, \leftarrow))$
- $\delta(q_n, \square, 1, \square) = (q_n, (\square, \downarrow), (1, \leftarrow), (1, \leftarrow))$
- $\delta(q_n, 1, \square, \square) = (q_n, (1, \leftarrow), (\square, \downarrow), (1, \leftarrow))$
- $\delta(q_c, \square, 0, \square) = (q_n, (\square, \downarrow), (0, \leftarrow), (1, \leftarrow))$
- $\delta(q_c, 0, \square, \square) = (q_n, (0, \leftarrow), (\square, \downarrow), (1, \leftarrow))$
- $\delta(q_c, \square, 1, \square) = (q_c, (\square, \downarrow), (1, \leftarrow), (0, \leftarrow))$
- $\delta(q_c, 1, \square, \square) = (q_c, (1, \leftarrow), (\square, \downarrow), (0, \leftarrow))$

Na kraju, kada više nema cifara ni u jednom sabirku, završavamo algoritam:

- $\delta(q_n, \square, \square, \square) = (q_F, (\square, \downarrow), (\square, \downarrow), (\square, \downarrow))$
- $\delta(q_c, \square, \square, \square) = (q_F, (\square, \downarrow), (\square, \downarrow), (1, \leftarrow))$

pri čemu je sa q_F označeno jedino završno stanje. Čitaocu za veŏu prepuštamo demonstraciju rada programa na primeru ulazne niske 11|101.

Primer 7.5. Čitaocu ostavljamo za vežbu da osmisli programe Tjuringovih mašina koje obavljaju aritmetičke operacije oduzimanja, množenja i celobrojnog deljenja nad brojevima zapisanim u binarnom zapisu. *Uputstvo*: simulirati standardne postupke računanja u binarnom sistemu.

Definicija 7.3. *Parcijalna funkcija nad* Σ^* je funkcija $f : L \to \Sigma^*$ gde je $L \subseteq \Sigma^*$ neki jezik nad Σ koji nazivamo *domen* funkcije f i označavamo ga sa $dom(f)$. Pritom, za nisku $w \notin L$ kažemo da je funkcija f *nedefinisana* za w i pišemo $f(w) = -$. Za parcijalnu funkciju f nad Σ^* kažemo da je *totalna* ako je definisana za svaku nisku $w \in \Sigma^*$.

Definicija 7.4. Parcijalna funkcija f_T nad Σ^* koju *izračunava* Tjuringova mašina T je definisana na sledeći način:

- ako za ulaz $w \in \Sigma^*$ Tjuringova mašina T staje i na izlazu daje $u \in \Sigma^*$, tada je $f_T(w) = u$
- u suprotnom, ako za ulaz $w \in \Sigma^*$ Tjuringova mašina T ne staje, tada je funkcija f_T nedefinisana za w.

Za parcijalnu funkciju f nad Σ^* kažemo da je *Tjuring izračunljiva* (ili T-izračunljiva) ako postoji Tjuringova mašina koja je izračunava.

Dakle, za razliku od URM programa i μ-rekurzivnih funkcija, Tjuringova mašina definiše funkcije nad rečima neke proizvoljne azbuke Σ, a ne nad prirodnim brojevima. Ovo na prvi pogled znači da ne možemo direktno upoređivati klase URM izračunljivih (tj. μ-rekurzivnih) i Tjuring izračunljivih funkcija, s obzirom da nisu definisane nad istim domenima. Ipak, lako se može videti da se svakoj parcijalnoj funkciji nad Σ^* može pridružiti parcijalna funkcija nad $\mathbb{N}$. Za ovo je dovoljno omogućiti da se proizvoljna niska $w \in \Sigma^*$ kodira prirodnim brojem. Ovo možemo uraditi tako što najpre simbolima azbuke pridružimo prirodne brojeve $0, 1, \ldots, n-1$, gde je $n = |\Sigma|$, čime se svaka niska $w \in \Sigma^*$ transformiše u konačnu sekvencu prirodnih brojeva. Ova sekvenca se može kodirati jednim prirodnim brojem na uobičajen način prikazan ranije. Ako sa $\alpha : \Sigma^* \to \mathbb{N}$ označimo ovu funkciju kodiranja, a sa f proizvoljnu parcijalnu funkciju nad Σ^*, tada će funkcija $g = \alpha \circ f \circ \alpha^{-1}$ biti parcijalna funkcija nad $\mathbb{N}$ arnosti 1. Funkcija g operiše nad kôdovima niski, pri čemu važi da je $g(\alpha(w)) = \alpha(u)$ ako i samo ako je $f(w) = u$ za svako $w, u \in \Sigma^*$. Sada važi sledeća teorema.

Teorema 7.1. *Za svaku Tjuring izračunljivu funkciju f njoj pridružena parcijalna funkcija $g = \alpha \circ f \circ \alpha^{-1}$ nad $\mathbb{N}$ je URM izračunljiva.*

Dokaz. (skica) Intuitivno, dati broj $x \in \mathbb{N}$ koji predajemo funkciji g predstavlja kôd ulazne niske w (tj. $x = \alpha(w)$). Porebno je simulirati rad Tjuringove mašine sa ulazom w pomoću URM programa sa ulazom x. Ovo je moguće, zato što je sadržaj trake Tjuringove mašine uvek konačan niz simbola (ne računajući simbole $\square$), pa je sadržaje traka moguće kodirati brojevima i čuvati ih u registrima URM mašine. Poseban registar će se koristiti za čuvanje stanja Tjuringove mašine. Sada je svaku naredbu Tjuringove mašine moguće simulirati tako što se dekodiraju sadržaji traka, na osnovu kodova očitanih simbola i trenutnog stanja

se odrede prelazi i zatim se tako izmenjeni sadržaji traka ponovo kodiraju i upišu u odgovarajuće registre. Ovaj intuitivni postupak se može prevesti u konkretan URM program. Na kraju, kôd sadržaja izlazne trake treba prebaciti u registar R_1, jer je to rezultat funkcije g. Dakle, funkcija g je URM izračunljiva. □

Iz prethodne teoreme sledi da se svaka Tjuringova mašina može simulirati URM programom. Za formalne modele izračunavanja koji mogu simulirati bilo koju Tjuringovu mašinu kažemo da su *Tjuring kompletni*. Dakle, URM je jedan Tjuring kompletan formalizam. Iz ekvivalentnosti URM programa i μ-rekurzivnih funkcija sledi da su i μ-rekurzivne funkcije Tjuring kompletne.

Napomena 7.2. Pojam Tjuringove kompletnosti se u praksi često koristi i za realne programske jezike, čime se izražava činjenica da se na tim programskim jezicima može realizovati bilo koji algoritam koji se može realizovati i Tjuringovom mašinom.

Razmotrimo sada obrnut slučaj. Neka je f parcijalna funkcija nad $\mathbb{N}$ arnosti k, i neka je sa $[\mathbb{N}]$ označen skup svih konačnih sekvenci prirodnih brojeva. Neka je sa $\gamma : [\mathbb{N}] \to \Sigma^*$ ($\Sigma = \{0, 1, |\}$) definisana funkcija kodiranja koja konačnu sekvencu brojeva $[x_1, \ldots, x_k]$ prevodi u nisku $w_1|w_2|\ldots|w_k \in \Sigma^*$, gde je w_i binarni zapis broja x_i, a simbol | se koristi kao separator. Tada će funkcija $g = \gamma \circ f \circ \gamma^{-1}$ biti parcijalna funkcija nad Σ^*, pri čemu će za svaku k-torku $(x_1, \ldots, x_k) \in \mathbb{N}^k$ važiti $g(\gamma(x_1, \ldots, x_k)) = \gamma(y)$ ako i samo ako je $f(x_1, \ldots, x_k) = y$. Sada važi sledeća teorema.

Teorema 7.2. *Za svaku URM izračunljivu funkciju f, njoj pridružena parcijalna funkcija $g = \gamma \circ f \circ \gamma^{-1}$ nad Σ^* je T-izračunljiva.*

Dokaz. (skica) Intuitivno, ako URM program P izračunava funkciju f, tada je potrebno konstruisati Tjuringovu mašinu koja će simulirati izvršavanje instrukcija programa P nad binarnom reprezentacijom podataka. Indeks tekuće instrukcije će se predstavljati stanjem Tjuringove mašine. Za svaki registar R_i koji koristi URM program P imaćemo posebnu traku na kojoj se nalazi binarna reprezentacija vrednosti u tom registru. Sada je samo potrebno omogućiti simulaciju operacija Z, S, T i J, što nije posebno teško. □

Iz prethodne dve teoreme sledi da su Tjuringove mašine suštinski ekvivalentne po izražajnosti sa druga dva formalizma (URM mašine i μ-rekurzivne funkcije) koja smo ranije upoznali, s obzirom da izračunavanje svake URM izračunljive funkcije možemo simulirati Tjuringovom mašinom i obrnuto, izračunavanje svake Tjuring izračunljive funkcije možemo simulirati URM programom.

7.2 Definicija složenosti

Definicija 7.5. *Vreme izvršavanja* Tjuringove mašine T za dati ulaz $w \in \Sigma^*$ (u oznaci $time(T(w))$) je jednako potrebnom broju koraka da se od početnog stanja stigne do završnog stanja pri ulazu w. *Prostor izvršavanja* Tjuringove mašine T za dati ulaz $w \in \Sigma$ (u oznaci $space(T(w))$) jednak je ukupnom broju polja na svim trakama mašine T koja se koriste tokom izvršavanja za ulaz w (tj. polja koja su u bar jednom trenutku tokom rada različita od □).

Pod veličinom ulaza w Tjuringove mašine podrazumevamo njegovu dužinu $|w|$, kada ga posmatramo kao nisku nad ulaznom azbukom Σ. Sada možemo definisati vremensku i prostornu složenost date Tjuringove mašine u funkciji od veličine ulaza.

Definicija 7.6. *Vremenska složenost* Tjuringove mašine T je funkcija $t_T(n)$ za koju važi:

$$t_T(n) = max_{|w|=n} time(T(w))$$

Prostorna složenost Tjuringove mašine T je funkcija $s_T(n)$ za koju važi:

$$s_T(n) = max_{|w|=n} space(P(w))$$

Dakle, u oba slučaja računamo maksimum potrebnih resursa za sve instance čija je ukupna veličina jednaka n. Dakle, uzimamo najgori slučaj među svim instancama date veličine.

Definicija 7.7. Neka su date dve funkcije $f, g : \mathbb{N} \to \mathbb{N}$. Tada je $f(n) = O(g(n))$ ako postoji konstanta $C \in \mathbb{N}$ i vrednost $n_0 \in \mathbb{N}$ takvi da je $f(n) \leq C \cdot g(n)$ za svako $n > n_0$.

Prethodna definicija uvodi uobičajenu ,,veliko-O" notaciju. Pritom, naglasimo da $O(g(n))$ nije jedna funkcija, već označava čitavu klasu funkcija koje asimptotski ne rastu brže od funkcije g. Oznaka $f(n) = O(g(n))$ znači da je f jedna takva funkcija. Intuitivno, ova oznaka znači da je $f(n)$ *najviše reda* $g(n)$, tj. da se $f(n)$ može *asimptotski ograničiti* sa $g(n)$. Na primer, važi da je $n^2 + n + 4 = O(n^2)$, jer za, na primer, $C = 2$ za svako $n \geq 3$ važiće da je $n^2 + n + 4 \leq 2n^2$.

Veliko-O notacija će nam omogućiti da uvedemo klase složenosti problema. Pritom, ,,problem" će ovde biti bilo koja totalna izračunljiva funkcija. Kao što znamo, izračunljiva funkcija f se može izračunati potencijalno pomoću različitih Tjuringovih mašina, ali nas zanima ona koja ima najmanju složenost.

Definicija 7.8. Neka je data totalna izračunljiva funkcija f arnosti k. Za f kažemo da pripada klasi vremenske (prostorne) složenosti $O(g(n))$ za neku funkciju $g : \mathbb{N} \to \mathbb{N}$ ako postoji Tjurngova mašina T koja izračunava funkciju f takva da je $t_T(n) = O(g(n))$ $(s_T(n) = O(g(n)))$.

Dakle, problem pripada klasi vremenske složenosti $O(g(n))$ ako se vreme izvšavanja može asimptotski ograničiti sa $g(n)$. Iz ove definicije sledi da isti problem može pripadati većem broju klasa, jer njegova vremenska složenost može imati više različitih gornjih granica. Na primer, ako imamo mašinu T koja izračunava funkciju f takva da je $t_T(n) = n^2 + n + 4$, tada je problem f u klasi vremenske složenosti $O(n^2)$, ali i u klasi $O(n^3)$, kao i u klasi $O(2^n)$, jer važi $t_T(n) = O(n^2)$, $t_T(n) = O(n^3)$ i $t_T(n) = O(2^n)$. Takođe, jasno je da neka klasa problema može biti potskup neke druge klase problema. Na primer, klasa $O(n^2)$ je potskup klase $O(n^3)$, jer svaki problem koji pripada klasi $O(n^2)$ je i u klasi $O(n^3)$. Iste napomene važe i za prostornu složenost.

Primer 7.6. Setimo se Tjuringove mašine iz primera 7.4 koja je sabirala dva prirodna broja u binarnom zapisu. Ako su n_1 i n_2 redom dužine zapisa prvog i drugog sabirka, tada je program najpre prolazio jednom kroz celu ulaznu nisku

da bi iskopirao prvi sabirak na drugu traku i ,,premotao" glavu prve trake na desni kraj drugog sabirka ($n_1 + n_2 + 1$ koraka), a zatim je vršio sabiranje standardnim postupkom ($max(n_1, n_2) + 1$ koraka). Ako je $n = n_1 + n_2 + 1$ veličina ulaza, jasno je da je ukupan broj koraka linearno ograničen u odnosu na n, tj. vremenska složenost ovog algoritma je $O(n)$.

Primer 7.7. Čitaocu ostavljamo za vežbu da se uveri da je složenost operacije oduzimanja takođe $O(n)$, dok je složenost operacija množenja i celobrojnog deljenja kvadratna – $O(n^2)$. Pritom, podrazumevamo standardne računske postupke u binarnom brojevnom sistemu.

Napomena 7.3. Složenost aritmetičkih operacija koje smo diskutovali u prethodnim primerima se odnose na Tjuringove mašine sa više traka. Ako bismo se ograničili na jednu traku, tada je složenost nešto veća. Na primer, prilikom sabiranja, morali bismo oba sabirka, kao i zbir, da držimo na istoj traci, kao i da stalno premotavamo glavu od cifara jednog sabirka ka ciframa drugog sabirka i zbira. Ovo znači da ćemo za svaku binarnu poziciju prilikom sabiranja morati da izvršimo $O(n)$ operacija pomeranja glave. Kako je broj pozicija koje se sabiraju jednak $max(n_1, n_2) + 1 = O(n)$, sledi da će ukupan broj koraka biti $O(n^2)$. Uopšte, može se pokazati da se svaki algoritam koji zahteva $O(f(n))$ koraka na Tjuringovoj mašini sa više traka može simulirati Tjuringovom mašinom sa jednom trakom u $O(f(n)^2)$ koraka. Dakle, postoji kvadratno usporenje.

Napomena 7.4. Iako smo složenost izračunavanja formalno definisali u terminima Tjuringovih mašina, u praksi je možemo razmatrati i manje formalno, ali na suštinski isti način, u kontekstu realnih računara. U tom kontekstu, pod veličinom ulaza se smatra memorijski prostor koji ulazni podatak zauzima. Veličina ovog prostora je obično izražena u bajtovima, ali može biti izražena i u bitovima ili npr., 32-bitnim rečima. Izbor konkretne jedinice nije od suštinske važnosti za složenost, s obzirom da se sve ove jedince međusobno razlikuju za konstantan faktor (npr. jedan bajt je osam bitova), a konstantni faktori ne utiču na složenost (u smislu ,,veliko-O" notacije). Broj koraka algoritma predstavlja broj izvršenih aritmetičko logičkih operacija i operacija transfera podataka. Prostorna složenost algoritma će odgovarati maksimalnom ukupnom zauzeću memorije tokom rada programa.

Ovako definisana složenost realnih računarskih programa će u dobroj meri odgovarati složenosti odgovarajućih algoritama implementiranih pomoću Tjuringove mašine sa više traka. Na primer, složenost sabiranja binarnih brojeva u realnim računarima je linearna, baš kao i kod sabiranja binarnih brojeva pomoću Tjuringove mašine sa više traka. Ipak, postoje i slučajevi gde će realni računari moći da implementiraju neke postupke efikasnije nego što bi to bilo moguće na Tjuringovim mašinama. Na primer, algoritam binarne pretrage implementiran na realnom računaru zahteva $O(log_2(n))$ koraka pretrage, gde je n broj elemenata niza. Svaki korak pretrage se obavlja u konstantnom vremenu, zato što je moguće neposredno pristupiti elementu niza sa proizvoljnim indeksom. Zato je i ukupna složenost ovog algoritma $O(log_2(n))$. Tjuringova mašina bi za svaki korak pretrage morala da pomera glavu ka odabranoj poziciji u nizu, što zahteva $O(n)$ koraka. Otuda će složenost binarne pretrage na Tjuringovoj mašini biti $O(n \cdot log_2(n))$.

Napomena 7.5. S obzirom da konstantni faktori ne utiču na složenost, u praksi nije neophodno precizno izbrojati bajtove koje neki ulaz zauzima. Dovoljno je da za veličinu ulaza n uzmemo neku veličinu koja je proporcionalna stvarnom broju

bajtova. Na primer, grafovi se u računarima često predstavljaju kao liste listi – imamo listu čvorova, pri čemu je svakom čvoru pridružena lista grana koje iz tog čvora izlaze. Veličina ove strukture podataka je proporcionalna zbiru broja čvorova i broja grana $|V| + |E|$, pa se ovaj zbir može uzeti za veličinu ulaznog podatka n. Stvarni broj bajtova može biti nekoliko puta veći, u zavisnosti od veličine celobrojnih tipova podataka kojima predstavljamo čvorove i grane, kao i zbog eventualnog prisustva pokazivača u čvorovima listi. Ipak, složenost će ostati asimptotski ista, uz nešto drugačiji konstantni faktor.

Isto važi i kada je u pitanju prebrojavanje koraka programa. Na primer, izvršavanje naredbe $x = y+z$ nad podacima celobrojnog (*int*) tipa može zahtevati nekoliko mašinskih instrukcija (učitavanje podataka y i z u registre, sabiranje u ALU jedinici i upis rezultata u promenljivu x), ali je taj broj koraka konstantan i unapred fiksiran. Zato ovu naredbu možemo brojati kao jedan korak algoritma. Sa takvom računicom će se broj koraka algoritma smanjiti za konstantan faktor, ali to neće uticati na složenost u asimptotskom smislu.

Napomena 7.6. Složenost izračunavanja se uvek izražava asimptotski – razmatramo koliko brzo raste funkcija složenosti $t_T(n)$ (ili $s_T(n)$) kada n teži beskonačnosti. To znači da uvek moramo pretpostaviti da se veličina ulaza može neograničeno uvećavati. U realnim programima, ako nam je na ulazu, na primer, jedan podatak x tipa *int*, tada se on ne može neograničeno uvećavati, jer je njegova veličina ograničena veličinom tog tipa podatka (32 bita na većini računara). Da bismo analizirali složenost ovakvih algoritama, neophodno je da pretpostavimo da je tip *int* zamenjen nekim celobrojnim tipom čija veličina u bitovima nije unapred fiksirana i može se uvećavati po potrebi.[1] Tek tada možemo pristupiti realnoj analizi složenosti algoritma.

Primer 7.8. Neka je dat sledeći program u jeziku `C++`:

```
i = 0;
while(i*i <= x) i++;
i--;
```

Ovaj program za ulaz x izračunava $\lfloor\sqrt{x}\rfloor$. Dominatni deo ovog algoritma je *while* petlja. Broj iteracija petlje je jednak $\lfloor\sqrt{x}\rfloor$. Da bismo asimptotski izrazili složenost, moramo pretpostaviti da je x nekog celobrojnog tipa čija veličina nije ograničena, te da broj bitova $n = \lfloor log_2(x)\rfloor + 1$ neophodan za zapis podatka x može neograničeno da raste. U svakoj iteraciji petlje dominira množenje čija je složenost $O(n^2)$. Kako je $x = O(2^n)$, asimptotsku složenost gornjeg algoritma možemo izraziti u funkciji od n kao $O(n^2 \cdot \sqrt{2^n}) = O(2^{log_2(n^2)} \cdot 2^{\frac{n}{2}}) = O(2^{2 \cdot log_2(n) + \frac{n}{2}})$. Dakle, algoritam je *eksponencijalne* složenosti u odnosu na veličinu ulaza.

7.3 Složenost problema odlučivanja

Posebna pažnja u proučavanju složenosti izračunavanja se posvećuje problemima odlučivanja, tj. problemima čiji je izlaz uvek odgovor „da” ili „ne”.

[1] Takvi tipovi obično ne postoje kao ugrađeni tipovi u programskim jezicima, ali su često podržani kao korisnički definisani tipovi u različitim bibliotekama za rad sa *velikim brojevima* (poput *GNU Multiprecision Library* (GMP) biblioteke).

Ukoliko ulazne instance problema posmatramo kao niske nad nekom azbukom Σ, tada skup svih ulaznih niski $w \in \Sigma^*$ za koje je odgovor „da" čine jedan jezik L nad azbukom Σ. Problem koji rešavamo ćemo u nastavku poistovećivati sa upravo sa tim jezikom L.

Definicija 7.9. *Karakteristična funkcija* jezika $L \subseteq \Sigma^*$ je funkcija $\chi_L : \Sigma^* \to \{0, 1\}$, definisana na sledeći način:

$$\chi_L(w) = \begin{cases} 1, & \text{za } w \in L \\ 0, & \text{za } w \notin L \end{cases}$$

Za jezik L kažemo da je *odlučiv* ili *rekurzivan* ako je χ_L izračunljiva.

U nastavku ove glave podrazumevamo da radimo sa odlučivim jezicima, tj. problemima.

Tjuringova mašina koja izračunava karakterističnu funkciju jezika L bi na izlazu trebalo da daje 0 ili 1. To znači da bismo azbuku Σ morali da proširimo specijalnim simbolima kojima označavamo ova dva moguća izlaza, uz restrikciju da se oni ne mogu nalaziti na ulazu. Kako bismo izbegli tehničke komplikacije ovog tipa, obično se Tjuringove mašine za rešavanje problema odlučivanja definišu tako da uopšte ne proizvode izlaz – umesto toga, podrazumevamo da postoje dva *završna stanja* – q_{da} i q_{ne} u kojima se rad Tjuringove mašine završava u zavisnosti od toga da li ulazna instanca pripada jeziku L ili mu ne pripada. Mi ćemo se u ovom tekstu takođe držati ove konvencije.

Primer 7.9. Razmotrimo problem odlučivanja „Da li je dati broj x paran?". Ako broj x predstavimo svojim binarnim zapisom, Tjuringova mašina koja odlučuje o ovom problemu bi prosto premotala glavu na desni kraj ulaza i proverila vrednost najnižeg bita. Ako je taj bit jednak 1, tada bi mašina prešla u stanje q_{ne}, a ako je jednak 0, tada bi prešla u stanje q_{da}. U oba slučaja, u konačnom broju koraka dobijamo odgovor na postavljeno pitanje. Dakle, ovaj problem je odlučiv. Složenost ove procedure odlučivanja je linearna, jer zahteva jedan prolaz kroz niz bitova.

Primer 7.10. Razmotrimo problem odlučivanja „Da li je broj x deljiv brojem y?". Ovaj problem bi se mogao rešiti tako što konstruišemo Tjuringovu mašinu koja vrši celobrojno deljenje i računa količnik i ostatak pri deljenju. Nakon što izračuna ostatak, potrebno je uporediti ga sa nulom – ako je jednak nuli, tada prelazimo u stanje q_{da}, a u suprotnom u q_{ne}. Složenost postupka celobrojnog deljenja je $O(n^2)$, dok poređenje sa nulom zahteva jedan prolaz kroz binarni zapis ostatka (što je $O(n)$). Dakle, ceo postupak je kvadratne složenosti.

Primer 7.11. Razmotrimo problem SAT i proceduru odlučivanja zasnovanu na istinitosnim tablicama. Ova procedura podrazumeva ispitivanje vrednosti ulazne KNF formule za 2^m valuacija, gde je m broj iskaznih slova koje se u formuli pojavljuju. Veličina ulaza n je jednaka ukupnom broju svih literala u svim klauzama ulazne KNF formule. Ispitivanje vrednosti formule u svakoj fiksiranoj valuaciji se može obaviti u jednom prolazu kroz formulu, tj. u $O(n \cdot m)$ koraka, jer je za svaki od n literala potrebno proveriti da li je tačan u datoj valuaciji (što zahteva prolaz kroz valuaciju[2]). Dakle, ukupan broj koraka je u najgorem slučaju $O(n \cdot m \cdot 2^m)$. Sa druge strane, veličina ulaza n može biti eksponencijalno velika

[2]U zavisnosti od implementacije valuacije, ispitivanje tačnosti literala u valuaciji je moguće uraditi i efikasnije, čak u konstantnom vremenu.

u odnosu na m, jer je ukupan broj mogućih različitih klauza nad m iskaznih promenljivih jednak 2^{2m}, a svaka klauza može sadržati najviše $2m$ literala. Otuda je u tom slučaju $n = 2m \cdot 2^{2m}$, odnosno, važi da je $\log_2 n = 1 + \log_2 m + 2m$, pa je $m \approx \frac{\log_2 n}{2} = \log_2 \sqrt{n}$. U tim ekstremnim slučajevima bi se gornja složenost svela na $O(n \cdot \log_2 \sqrt{n} \cdot \sqrt{n}) = O(n^{\frac{3}{2}} \log \sqrt{n})$. Ipak, najgori slučaj se dobija kada razmatramo ,,kompaktnije" formule, sa znatno manjim brojem klauza i literala. Na primer, ako važi $n = m$, tada je složenost $O(n^2 \cdot 2^n)$. Ovo je najgori slučaj, zato što KNF formula u kojoj se pojavljuje m iskaznih slova ne može imati manje od m literala.

7.4 Klasa P

U ovom poglavlju razmatramo jednu od najznačajnijih klasa složenosti – klasu *polinomske vremenske složenosti* koju označavamo sa P (ili PTIME).

Definicija 7.10. Klasu problema *polinomske vremenske složenosti* P čine svi problemi odlučivanja za koje postoji procedura odlučivanja čija je vremenska složenost $O(p(n))$ za neki polinom p.

Primer 7.12. Sledeći problemi pripadaju klasi P:

- ,,Da li je broj x paran?"
- ,,Da li je broj x deljiv brojem y?"
- ,,Da li je $x < y$?"
- ,,Da li binarni zapis broja x sadrži dve uzastopne nule?"
- ,,Da li je dati broj x stepen dvojke?"
- ,,Da li u datom nizu brojeva $x_1, \ldots, x_n$ postoji par brojeva (x_i, x_j) takvi da je njihov proizvod $x_i \cdot x_j$ jednak datom broju x?"

itd. Za neke od ovih problema smo već razmotrili procedure odlučivanja i odredili njihovu složenost. Za ostale probleme ostavljamo čitaocu da to uradi za vežbu.

Primer 7.13. Razmotrimo problem ,,Da li je dati neusmereni graf $G = (V, E)$ povezan?" Graf je povezan ako za svaka dva čvora $u, v \in V$ postoji put $u = v_0, v_1, \ldots, v_k = v$, pri čemu je $(v_i, v_{i+1}) \in E$ za svako $i \in \{0, 1, \ldots, k-1\}$. Ovaj problem možemo rešiti tako što izvršimo obilazak grafa iz proizvoljnog početnog čvora i označimo sve čvorove koji su tom prilikom dostignuti. Ako su na kraju svi čvorovi dostignuti, znači da je graf povezan, dok u suprotnom nije. Procedura obilaska grafa (poput, npr. *obilaska u dubinu*) zahteva da se svaki čvor i svaka grana obiđe po jednom, što znači da je složenost $O(|V| + |E|)$, tj. $O(n)$ ako uzmemo da je veličina grafa $n = |V| + |E|$. Dakle, ovaj problem pripada klasi P.

Primer 7.14. Problem ,,Da li dati usmereni graf $G = (V, E)$ sadrži ciklus?", tj. da li postoji put $v_0, v_1, \ldots, v_k$ takav da je $(v_i, v_{i+1}) \in E$ za $i \in \{0, \ldots, k-1\}$ i $v_0 = v_k$ se može, kao i prethodni, svesti na obilazak grafa. Ukoliko se tokom obilaska u dubinu naiđe na čvor koji je već označen, znači da imamo ciklus, u suprotnom ciklus ne postoji. Dakle, složenost ove procedure je takođe linearna u odnosu na veličinu grafa, pa i ovaj problem pripada klasi P.

Primer 7.15. Razmotrimo problem ,,Da li je dati broj x potpun kvadrat?". Ovaj problem je ekvivalentan sa ispitivanjem da li je $\lfloor\sqrt{x}\rfloor^2 = x$. U primeru 7.8 smo razmotrili jedan algoritam za izračunavanje $\lfloor\sqrt{x}\rfloor$ koji je bio eksponencijalne složenosti. Ipak, postoje i efikasniji algoritmi da se ovo izračuna. Ideja je da posmatramo interval $I = \{1, 2, \ldots, x\}$ i da broj y takav da je $y \cdot y = x$ tražimo binarnom pretragom. Ako sa n označimo broj bitova koji zauzima zapis broja x, tada je $x = O(2^n)$, tako da interval I ima $O(2^n)$ elemenata. Binarnom pretragom je potrebno $O(log_2(2^n)) = O(n)$ koraka da se u ovom intervalu locira traženi element y, ako postoji, ili da se utvrdi da takav element ne postoji. Za računanje središnjeg elementa tekućeg podintervala potrebno je vreme $O(n)$ (sabiranje i deljenje sa dva). Za njegovo kvadriranje je potrebno $O(n^2)$ koraka, a za upoređivanje sa x još $O(n)$ koraka. Otuda je složenost celog algoritma $O((n + n^2 + n) \cdot n) = O(n^3)$. Dakle, problem pripada klasi P.

Primer 7.16. Razmotrimo problem ,,Da li je niska $u \in \Sigma^*$ *faktor* niske $w \in \Sigma^*$?". Za nisku u kažemo da je faktor niske w ako se niska w može zapisati u obliku $w = \alpha u \beta$, gde su $\alpha, \beta \in \Sigma^*$. Ovaj problem se može rešiti na sledeći način. Neka je $w = a_1 a_2 \ldots a_m$, a $u = b_1 \ldots b_k$. Ako je $m < k$, onda je odgovor definitivno ,,ne". U suprotnom, najpre možemo uporediti niske $a_1 \ldots a_k$ i $b_1 \ldots b_k$. Ako su ove niske jednake, vraćamo ,,da". U suprotnom, treba uporediti $a_2 \ldots a_{k+1}$ sa $b_1 \ldots b_k$, zatim $a_3 \ldots a_{k+2}$ sa $b_1 \ldots b_k$ i tako dalje, zaključno sa $a_{m-k+1} \ldots a_m$. Ako ni jedna od ovih niski nije jednaka $b_1 \ldots b_k$, vraćamo ,,ne", u suprotnom vraćamo ,,da". Složenost opisanog postupka je jednaka $k \cdot (m - k + 1)$. Ako je $n = m + k$ ukupna veličina ulaza, tada je $k = O(n)$ i $m = O(n)$, pa je složenost $O(n^2)$ u najgorem slučaju. Dakle, ovaj problem je takođe u klasi P. Napomenimo da postoje i efikasniji algoritmi koji rešavaju ovaj problem (npr. *Knut-Moris-Prat* (KMP) algoritam, čija je složenost linearna).

Primer 7.17. Za problem ,,Da li je dati broj x prost?" dugo nije bio poznat ni jedan algoritam polinomske složenosti, tako da se nije znalo da li pripada klasi P ili ne. Tek 2002. godine je konstruisan prvi algoritam odlučivanja za koji je pokazano da je polinomske složenosti. Složenost ovog algoritma (poznatog i kao *AKS (Agrawal–Kayal–Saxena) test primalnosti*) uz sva poboljšanja koja su naknadno usledila, je $O(n^6)$, gde je n broj cifara broja x.

Napomena 7.7. Kada utvrđujemo da li je problem u klasi P, tada nam nije važan tačan stepen polinoma kojim je vreme izvršavanja programa ograničeno. Zbog toga analiza može biti dosta ,,gruba" i lišena mnogih nepotrebnih detalja. Na primer, ne moramo se truditi da pronađemo najefikasniji mogući algoritam – bilo koji algoritam polinomske složenosti je dovoljan da utvrdimo da je problem u klasi P. Takođe, veličinu ulaza možemo zameniti bilo kojom veličinom n koja je *polinomski povezana* sa stvarnom veličinom ulaza N, tj. za koju važi da postoje polinomi p i q takvi da je $n = O(p(N))$ i $N = O(q(n))$. Na primer, kada analiziramo grafovske probleme, za veličinu grafa možemo uzeti broj čvorova grafa $n = |V|$. Iako znamo da je veličina grafa proporcionalna sa $N = |V| + |E|$, broj grana $|E|$ je uvek $O(n^2)$, pa će važiti da je $N = |V| + |E| = n + O(n^2) = O(n^2)$, dok je $n \leq N$, tj. važi $n = O(N)$. Zato možemo koristiti n kao veličinu ulaza. Ovo može dovesti do toga da rezultujući polinom kojim ograničavamo vreme izvršavanja bude drugačijeg stepena, ali će i dalje biti polinom. Na primer, procedure zasnovane na obilasku grafa (kao u primerima 7.13 i 7.14) će sada biti kvadratne složenosti u odnosu na n, ali to je i dalje polinomsko ograničenje.

Podsetimo se da je za problem odlučivanja Π, njegov *komplement* $\neg\Pi$ pro-

blem kod koga su svi odgovori negirani, tj. instance problema Π za koje je odgovor ,,da" postaju instance problema ¬Π za koje je odgovor ,,ne" i obrnuto. U tom smislu, klasa P je zatvorena u odnosu na operaciju komplementa problema, o čemu govori sledeća teorema.

Teorema 7.3. *Ako je neki problem* Π *u klasi P, tada je i njegov komplement* ¬Π *u klasi P.*

Dokaz. Ako je T Tjuringova mašina koja rešava problem Π u polinomskom vremenu, tada će mašina T' dobijena od T tako što se svi prelazi u stanje q_{da} zamene prelascima u stanje q_{ne} i obrnuto rešavati problem ¬Π (intuitivno, rešimo problem Π i onda negiramo odgovor). Ova mašina će imati istu složenost kao i T, dakle, polinomsku. □

7.5 Klasa EXP

Ukoliko se neki problem odlučivanja ne može rešiti u polinomskom vremenu, to znači da vremenska složenost svakog algoritma za rešavanje tog problema raste brže od bilo kog polinoma. Tada je prirodno razmotriti da li postoji algoritam za rešavanje tog problema čija se složenost može ograničiti eksponencijalnom funkcijom. U skladu sa tim, imamo sledeću definiciju.

Definicija 7.11. Klasu problema *eksponencijalne vremenske složenosti* čine svi problemi odlučivanja za koje postoji procedura odlučivanja čija je vremenska složenost $O(2^{p(n)})$, za neki polinom p. Ovu klasu označavamo sa EXP (ili EXPTIME).

Primetimo da za svaki polinom p važi $O(p(n)) \subseteq O(2^n)$, pa je svaki problem koji je u klasi P ujedno i u klasi EXP. Dakle, važi:

$$\mathrm{P} \subseteq \mathrm{EXP}$$

Ipak, može se pokazati da je gornja inkluzija stroga, tj. da postoje problemi koji su u EXP, a nisu u P, tj. važi:

$$\mathrm{P} \subsetneq \mathrm{EXP}$$

Jedan takav primer je problem *ograničenog zaustavljanja* (engl. *bounded halting problem*): za datu Tjuringovu mašinu T, njen ulaz x i dati broj k, treba ispitati da li se mašina za ulaz x zaustavlja u najviše k koraka. Naime, može se pokazati da u opštem slučaju ne postoji efikasniji način da proverimo da li se proizvoljna Tjuringova mašina zaustavlja u najviše k koraka za proizvoljan ulaz x nego da simuliramo tih k koraka rada i vidimo da li će se program zaustaviti (dokaz ove činjenice izostavljamo). Analizirajmo složenost ovog postupka simulacije. Veličina ulaza će biti $n = |M| + |x| + |k|$, gde je $|M|$ broj bitova koji je potreban za kodiranje mašine M, $|x|$ je broj bitova u binarnom zapisu broja x, a $|k|$ je broj bitova u binarnom zapisu broja k. Kako je M proizvoljna Tjuringova mašina, a $|x|$ proizvoljan ulaz, u najgorem slučaju možemo pretpostaviti da $|k|$ dominira u veličini ulaza, tj. da je $n \approx |k|$ (npr. možemo imati jednostavan program M koji za relativno mali ulaz x radi veoma dugo). U tom slučaju će broj koraka ove simulacije biti $O(k) = O(2^{log_2(k)}) = O(2^n)$. Dakle, opisani

postupak simulacije zahteva eksponencijalno vreme, pa ovaj problem nije rešiv u polinomskom vremenu.

Slično kao i za klasu P, i klasa EXP ima svojstvo zatvorenosti za operaciju komplementa, o čemu govori sledeća teorema.

Teorema 7.4. *Ako je* $\Pi \in EXP$*, tada je i* $\neg\Pi \in EXP$*.*

Dokaz. Dokaz ide isto kao i u slučaju P klase. □

7.6 Klasa NP

U prethodnom poglavlju smo videli da postoje problemi koji se mogu rešiti u eksponencijalnom vremenu, a koji dokazano nisu u P. Ipak, većina realnih problema koji su vremenski zahtevni za rešavanje se nalaze u *sivoj zoni* – svi poznati algoritmi za njihovo rešavanje su eksponencijalne složenosti, ali *nije dokazano* da ne postoji algoritam polinomske složenosti, tj. da nisu u P. Drugim rečima, teorijski je moguće da će za neki od tih problema neko u budućnosti pronaći algoritam polinomske složenosti i time pokazati da problem pripada klasi P. Na primer, za ranije pomenuti problem ,,da li je dati prirodan broj n prost?" su dugo postojali samo algoritmi eksponencijalne složenosti, da bi tek nakon više decenija istraživanja bio konstruisan algoritam polinomske složenosti.

Primer 7.18. Jedan od najpoznatijih problema koji spadaju u ovu kategoriju je problem SAT. U primeru 7.11 smo videli da metod istinitosnih tablica ima eksponencijalnu složenost u najgorem slučaju. Iako danas postoje algoritmi za rešavanje SAT problema koji su u praksi mnogo brži, pokazuje se da u najgorem slučaju svi ti algoritmi i dalje imaju eksponencijalnu složenost u odnosu na veličinu ulazne KNF formule. Ipak, niko nikada nije dokazao da nije moguće ovaj problem rešiti u polinomskom vremenu. Drugim rečima, još uvek postoji nada da će neko nekada uspeti da pronađe polinomski algoritam.

Primer 7.19. Još jedan zanimljiv problem koji upada u ovu ,,sivu zonu" jeste problem bojenja grafa: da li je moguće obojiti svaki od čvorova datog grafa jednom od tri boje tako da nikoja dva susedna čvora ne budu obojeni istom bojom? Naivni pristup rešavanju ovog problema bi se sastojao u tome da ispitujemo sva moguća bojenja – njih ima 3^n, gde je n broj čvorova grafa (koji možemo uzeti za veličinu ulaza). Iako postoje i znatno pametniji načini za rešavanje ovog problema, u najgorem slučaju, svi postojeći algoritmi i dalje imaju eksponencijalnu složenost. Međutim, ni za ovaj problem niko nikada nije dokazao da nije moguće rešiti ga u polinomskom vremenu.

Primer 7.20. Razmotrimo problem *particije*: imamo multiskup[3] od k brojeva $X = \{x_1, \ldots, x_k\}$ i pitamo se da li je moguće ovaj skup podeliti u dva podskupa A i B takvih da je $X = A \cup B$ i $\sum_{x_i \in A} x_i = \sum_{x_j \in B} x_j$? Naivni pristup bi se sastojao u tome da razmatramo sve moguće podskupove $A \subseteq X$ i $B = X \setminus A$, računamo odgovarajuće sume i upoređujemo ih. Kako je broj takvih potskupova A jednak 2^k, potrebno je ispitati 2^k slučajeva. Sa druge strane, veličina ulaza n je jednaka $|x_1| + \ldots + |x_k|$. Može se pokazati da je za svaki izbor skupova A i B izračunavanje gornjih suma i proveru jednakosti moguće izvršiti u $O(n)$ koraka. Sa druge strane, broj k može u najgorem slučaju biti blizak n (ako su brojevi x_i relativno mali). Otuda je složenost u najgorem slučaju jednaka $O(n \cdot 2^n)$.

[3] Za razliku od običnog skupa, u multiskupu se elementi mogu ponavljati.

Iako postoje i algoritmi koji su u proseku efikasniji od ovog naivnog pristupa, u najgorem slučaju su svi oni i dalje eksponencijalne složenosti. Ipak, niko još uvek nije dokazao da ovaj problem nije moguće rešiti u polinomskom vremenu.

Za probleme opisane u prethodnim primerima znamo da su u klasi EXP, ali *ne znamo* da li su u klasi P. Ipak, navedeni problemi (kao i mnogi drugi praktični problemi iz klase EXP, ali ne svi!) imaju jedno zanimljivo svojstvo: oni dopuštaju *polinomsku verifikaciju potvrdnih odgovora*. To znači da za svaku instancu problema odlučivanja za koju je odgovor potvrdan („da") postoji *sertifikat* polinomske veličine u odnosu na veličinu ulaza – u pitanju je struktura koja tvrdi da je odgovor za datu instancu potvrdan. Proveru ispravnosti ovog sertifikata je moguće izvršiti u polinomskom vremenu i tako se uveriti se da je odgovor zaista potvrdan za datu instancu. Ono što u opštem slučaju ne znamo je na koji način konstruisati ovaj sertifikat u polinomskom vremenu. Drugim rečima, traganje za ovim sertifikatom je ono što postojeće algoritme za rešavanje ovakvih problema čini eksponencijalnim.

Primer 7.21. Za problem SAT, sertifikat bi bila proizvoljna zadovoljavajuća valuacija. Njena veličina je uvek polinomski ograničena u odnosu na veličinu ulazne KNF formule. Ako nam je data zadovoljavajuća valuacija, u polinomskom vremenu možemo se uveriti da je formula za tu valuaciju zaista tačna i time *verifikovati* zadovoljivost date KNF formule. Dakle, pronaći zadovoljavajuću valuaciju je teško, ali uveriti se da je ona zaista zadovoljavajuća je nešto što je moguće uraditi u polinomskom vremenu.

Primer 7.22. U problemu bojenja grafa, sertifikat bi bilo proizvoljno ispravno bojenje. Bojenje se može zapisati u prostoru veličine $O(n)$, gde je n broj čvorova grafa, pa je sertifikat polinomske veličine u odnosu na veličinu ulaza. Ako nam je dato takvo bojenje, mi možemo u polinomskom vremenu proveriti da je ono zaista ispravno, i time verifikovati potvrdan odgovor (tj. da je graf moguće obojiti na ispravan način). Opet, pronaći ispravno bojenje je teško, a uveriti se da je neko unapred dato bojenje ispravno nije.

Primer 7.23. U problemu particije, sertifikat bi bio proizvoljan potskup A skupa $X = \{x_1, \ldots, x_k\}$ takav da je $\sum_{x_i \in A} x_i = \sum_{x_j \in X \setminus A} x_j$. Veličina ovog skupa je polinomski ograničena veličinom ulaza n. Ako nam je A dat, u polinomskom vremenu možemo proveriti da gornja jednakost zaista važi i time se uveriti da je odgovor za datu instancu potvrdan.

Primetimo da u svim ovim primerima postoji polinomski sertifikat za instance za koji je odgovor „da". Sa druge strane, za instance za koje je odgovor „ne" (npr. nezadovoljive KNF formule, neobojivi grafovi) struktura ovih problema ne nudi jasan način na koji bismo mogli da konstruišemo sertifikat polinomske veličine koji potvrđuje takav negativan odgovor. Dakle, kod problema opisanim u prethodnim primerima postoji izvesna asimetrija između instanci za koje je odgovor „da" i instanci za koje je odgovor „ne".

Rešavanje opisanih problema je, makar u slučaju instanci za koje je odgovor potvrdan, moguće svesti na fazu pronalaženja sertifikata i na fazu verifikacije sertifikata. Pritom, faza pronalaženja sertifikata je teška, dok se verifikacija može obaviti u polinomskom vremenu. Alternativa je da sertifikat generišemo na slučajan način (izaberemo slučajnu valuaciju, slučajno bojenje). Ovo „pogađanje" možemo izvršiti u polinomskom vremenu, jer je veličina sertifikata polinomska. Ako bismo bili „vidoviti", mogli bismo iz prve pogoditi pravi

sertifikat, čime bi ceo algoritam pogađanja i verifikacije potvrdnog odgovora bio završen u polinomskom vremenu. Ovakve algoritme koji se sastoje iz faze slučajnog pogađanja sertifikata i faze provere nazivamo *nedeterminističkim algoritmima*. Formalno, imamo sledeću definiciju.

Definicija 7.12. *Nedeterministička Tjuringova mašina* je Tjuringova mašina za probleme odlučivanja (tj. mašina koja ima dva završna stanja, q_{da} i q_{ne}) kod koje se izvršavanje sastoji iz faze *pogađanja* i faze *verifikacije*:

- faza pogađanja se sastoji u tome da se ulaz x transformiše u ulaz $w\Box x$, pri čemu je $w \in \Gamma^*$ *sertifikat* koji je generisan na slučajan način u vremenu $|w|$.
- faza verifikacije se izvršava po uobičajenim principima rada Tjuringove mašine nad ulazom $w\Box x$, pri čemu je glava za čitanje/pisanje inicijalno pozicionirana na početnom simbolu niske x.

Mašina T *prihvata* ulaz x ako postoji sertifikat w takav da faza verifikacije nad niskom $w\Box x$ završava rad u stanju q_{da}. U suprotnom, T *ne prihvata* (*odbacuje*) ulaz x. *Jezik* (*problem*) $L(T) \subseteq \Sigma^*$ *prepoznat* mašinom T je skup svih niski $x \in \Sigma^*$ koje mašina prihvata, tj. za koje postoji sertifikat w za koji se rad mašine završava u stanju q_{da}.

Dakle, pojam nedeterminističke Tjuringove mašine formalizuje prethodno opisani pristup slučajnog pogađanja sertifikata i njegove provere. Termin *nedeterministička* potiče odatle što za svaki ulaz x možemo imati beskonačno mnogo različitih izvršavanja – za svaki sertifikat w po jedno. Naime, Tjuringova mašina može (i najčešće hoće) obrađivati i sertifikat w tokom faze verifikacije, pa će samim tim njen rad zavisiti od izbora sertifikata w. Na primer, u slučaju SAT problema, sertifikat će biti neka valuacija, a algoritam provere će obilaziti tu valuaciju i proveravati istinitost ulazne formule u toj valuaciji. Za različit izbor valuacije, ova provera će teći na različite načine. S obzirom da se sertifikat bira na slučajan način, nije unapred poznato na koji način će se izvršavati program za dati ulaz x. Ipak, da bismo smatrali da je za instancu x odgovor potvdan, dovoljno je da postoji *bar jedno* izvršavanje za koje će mašina završiti svoj rad u stanju q_{da}.

Primetimo da vreme i prostor potrebni za izvršavanje Tjuringove mašine T za dati ulaz x zavise od izbora sertifikata w. S obzirom da je naša pretpostavka da mi možemo ,,pogoditi" pravi sertifikat, razmatraćemo najbolji mogući slučaj:

$$nd\_time(T(x)) = min_{w \in \Sigma^* \wedge T(w\Box x)=da}(|w| + time(T(w\Box x)))$$

Dakle, *nedeterminističko vreme izvršavanja* mašine T za nisku $x \in L(T)$ (u oznaci $nd\_time(T(x))$ je jednako najmanjem vremenu potrebnom za uspešno pogađanje (vreme $|w|$) i determinističku verifikaciju (vreme $time(T(w\Box x))$).

Sada složenost nedeterminističke Tjuringove mašine T definišemo na sledeći način.

Definicija 7.13. *Vremenska složenost* nedeterminističke Tjuringove mašine T je funkcija $t_T^{nd}(n)$ za koju važi:

$$t_T^{nd}(n) = max_{x \in L(T) \wedge |x|=n} nd\_time(T(x))$$

Primetimo da se složenost nedeterminističke Tjuringove mašine razmatra isključivo u odnosu na instance koje pripadaju jeziku $L(T)$, tj. instance za koje je odgovor „da". Instance za koje je odgovor „ne" nisu relevantne za definiciju nedeterminističke složenosti.

Od posebnog značaja su nedeterministički algoritmi polinomske vremenske složenosti. Problemi odlučivanja za koje postoje takvi algoritmi pripadaju klasi NP.

Definicija 7.14. Klasu problema *nedeterminističke polinomske vremenske složenosti* NP čine svi problemi odlučivanja za koje postoji nedeterministička Tjuringova mašina T koja ih prepoznaje, a čija je vremenska složenost $O(p(n))$ za neki polinom p.

Primetimo da izvršavanje nedeterminističkog algoritma u polinomskom vremenu podrazumeva da se obe faze – pogađanje i verifikacija, izvrše u polinomskom vremenu. Kako je vreme pogađanja jednako samoj veličini sertifikata, sledi da sertifikat mora biti polinomske veličine u odnosu na ulaz, kao i da se verifikacija (koja se izvršava deterministički) obavi u polinomskom vremenu. Problemi za koje je ovako nešto moguće spadaju u klasu NP.

Primer 7.24. Problemi razmatrani u prethodnim primerima (problem SAT, bojenje grafova, problem particije) svi pripadaju klasi NP, s obzirom da su njihovi sertifikati polinomske veličine, a verifikacija se obavlja u polinomskom vremenu.

Primer 7.25. Razmotrimo sledeći problem *faktorizacije*: „Za date brojeve $x, k \in \mathbb{N}$, da li postoji broj q, takav da je $1 < q \leq k$ i da q deli x?". Ovaj problem je definitivno u klasi EXP, s obzirom da se može, grubom silom rešiti tako što redom ispitujemo deljivost broja n sa svakim od brojeva $q \in \{2, 3, \ldots, k\}$. Ako je veličina ulaza $n = |x| + |k|$, tada se svako ispitivanje deljivosti može obaviti u vremenu $O(n^2)$, pa je složenost izražena sa $O(k \cdot n^2)$. Kako je $k = O(2^{\log_2(k)}) = O(2^n)$, ukupna složenost ovog naivnog postupka je $O(n^2 \cdot 2^n)$.

Ispitajmo da li je ovaj problem u klasi NP. Za instance za koje je odgovor potvrdan postojaće neko $q \leq k$ takvo da q deli x. To q pretstavlja sertifikat koji treba pogoditi. Njegova veličina je $O(n)$, jer je $q \leq x$, pa je $|q| \leq |x| \leq n$. Dakle, sertifikat je polinomske veličine. Dalje, da bismo za dati sertifikat q verifikovali potvrdan odgovor, dovoljno je izračunati ostatak pri deljenju x sa q. Ovo je moguće uraditi u vremenu $O(n^2)$. Otuda je problem faktorizacije u klasi NP.

Napomena 7.8. Napomenimo da do sada nije pronađen ni jedan (deterministički) algoritam polinomske složenosti koji rešava problem faktorizacije. Dakle, nije poznato da li je ovaj problem u klasi P.

Na prvi pogled, ovo deluje kontradiktorno sa ranije navedenom činjenicom da je problem ispitivanja da li je dati broj prost rešiv polinomskim algoritmom. Naime, deluje da su ova dva problema komplementarna, jer je broj prost ako i samo ako nema netrivijalne faktore. Ipak, ako pažljivije pogledamo definiciju problema faktorizacije, vidimo da tu zahtevamo da postoji faktor *manji ili jednak* od datog broja k. Dakle, problem „Da li postoji bilo koji faktor broja x različit od 1 i x?", odnosno „Da li je x složen broj?" jeste polinomski rešiv, jer je to komplement problema „da li je x prost?". Sa druge strane, problem faktorizacije ima dodatni argument k na ulazu i pita nas da li postoji faktor manji ili jednak od k. Za ovaj problem još uvek ne znamo da li je u klasi P.

Problem faktorizacije je značajan u praksi zato što bismo njegovim efikasnim rešavanjem mogli da efikasno rastavimo proizvoljan prirodan broj x na proste

činioce. Zaista, ako bismo imali mašinu T koja efikasno (u polinomskom vremenu) rešava problem faktorizacije (koji je problem odlučivanja), tada bismo mogli da efikasno pronađemo *najmanji* faktor broja x veći od 1. Ovo bismo mogli da uradimo binarnom pretragom intervala $\{2, \ldots, x\}$, tako što krenemo od broja $k = div(x, 2)$ i pitamo se da li postoji faktor manji od k. Ako je odgovor potvrdan, pretragu nastavljamo u podintervalu $\{2, \ldots, k\}$, a u suprotnom, pretragu dalje vršimo u podintervalu $\{k + 1, \ldots, x\}$. Kada pronađemo najmanji faktor, tada znamo da on mora biti prost. Nakon toga možemo podeliti x sa tim najmanjim faktorom i dobiti broj x' nad kojim dalje primenjujemo isti postupak.[4]

S obzirom da za sada nije poznato ni jedno efikasno rešenje problema faktorizacije, sledi da je rastavljanje velikih prirodnih brojeva na proste faktore i dalje izazovan zadatak, čak i za moderne računare. Otuda se neki od najpoznatijih *kriptografskih algoritama* (poput *RSA* algoritma) upravo zasnivaju na nemogućnosti lakog rastavljanja velikih prirodnih brojeva na proste faktore.

Primer 7.26. Razmotrimo problem UNSAT: da li je data iskazna formula u KNF-u *nezadovoljiva*? Ovaj problem je komplementaran SAT problemu, pa je samim tim odlučiv. Kako nije poznat ni jedan polinomski algoritam za rešavanje problema SAT, jasno je da to isto važi i za problem UNSAT. Dakle, ne znamo da li je ovaj problem u klasi P. Sa druge strane, da bi ovaj problem bio u klasi NP potrebno bi bilo da za svaku instancu za koju je odgovor ,,da" (tj. za svaku nezadovoljivu KNF formulu) postoji sertifikat polinomske veličine pomoću koga je moguće u polinomskom vremenu verifikovati nezadovoljivost. Do sada niko nije pokazao da takvi sertifikati postoje u opštem slučaju. Otuda nije poznato ni da li problem UNSAT pripada klasi NP.

Primetimo da ovaj problem trivijalno pripada klasi EXP, s obzirom da toj klasi pripada i problem SAT (npr. metod istinitosnih tablica je eksponencijalne složenosti, a njime se rešava i problem UNSAT).

Prethodni primer demonstrira asimetričnost klase NP: to što je neki problem u klasi NP ne znači da je i njegov komplement u toj klasi. Ovo je upravo zato što, po definiciji, problemi iz klase NP poseduju polinomske sertifikate samo za instance za koje je odgovor ,,da", ali ne i za one instance za koje je odgovor ,,ne". Za komplementarni problem odgovori se obrću (,,da" postaje ,,ne" i obrnuto), tako da za njegove potvrdne instance polinomski sertifikati u opštem slučaju ne moraju da postoje, osim ako eksplicitno ne pronađemo način da ih konstruišemo.

Iz ove diskusije sledi da klasa NP ne mora biti zatvorena za operaciju komplementa. Zbog toga uvodimo sledeću definiciju.

Definicija 7.15. Klasu co-NP čine svi problemi čiji su komplementi u klasi NP.

Problem UNSAT je tipičan predstavnik klase co-NP. Slično, problem *neobojivosti grafa* (,,da li se graf *ne može* korektno obojiti sa tri boje?") je u klasi co-NP, kao i svi drugi problemi koji se dobijaju komplementiranjem problema za koje znamo da su u klasi NP.

[4] Opisani postupak zasnovan na binarnoj pretrazi je standardni način na koji se pojedini problemi *pretrage* (tj. problemi kod kojih se traži neko y koje je u nekoj zadatoj relaciji sa ulazom x), mogu svoditi na uzastopno rešavanje više instanci nekog pogodno odabranog problema odlučivanja.

7.7 Odnosi između klasa P, NP i EXP

U ovom poglavlju razmatramo odnose između tri uvedene klase vremenske složenosti P, NP i EXP. Ranije smo konstatovali da važi:

$$\mathrm{P} \subsetneq \mathrm{EXP}$$

Ono što se može pokazati je da važi:

$$\mathrm{P} \subseteq \mathrm{NP} \subsetneq \mathrm{EXP}$$

Da bismo dokazali da važi $\mathrm{P} \subseteq \mathrm{NP}$, dovoljno je da se setimo definicije nedeterminističke Tjuringove mašine: ona funkcioniše po istom principu kao i obična Tjuringova mašina, s tim što fazi determinističke verifikacije prethodi faza nedeterminističkog pogađanja. Ako je problem u klasi P, tada postoji deterministička Tjuringova mašina T koja implementira proceduru odlučivanja za dati problem, a koja se izvršava u polinomskom vremenu. Odgovarajuću nedeterminističku mašinu T' možemo dobiti od mašine T tako što dodamo fazu pogađanja sertifikata w, pri čemu će algoritam verifikacije nad ulazom $w\Box x$ najpre prebrisati nisku w (tj. zameniti njene simbole sa $\Box$), nakon čega će premotati glavu za čitanje/pisanje na početak niske x i zatim nastaviti sa izvršavanjem programa mašine T. Pritom, uvek možemo za sertifikat uzeti proizvoljnu nisku $w \in \Gamma^*$ za koju je $|w|$ polinomski ograničeno u odnosu na veličinu ulaza x. Otuda će složenost ovakvog nedeterminističkog postupka biti polinomska.

Da bismo pokazali da je $\mathrm{NP} \subseteq \mathrm{EXP}$, pretpostavimo da je za proizvoljan problem A data nedeterministička Tjuringova mašina T koja ga prepoznaje u polinomskom vremenu $O(p(n))$ za neki polinom $p(n)$. Jasno je da za ulaz x veličine $|x| = n$, veličina sertifikata w takođe mora biti ograničena sa $p(n)$, jer bi u suprotnom već faza pogađanja trajala duže. Ovakvih sertifikata ima konačno mnogo, ali eksponencijalno mnogo, tj. $1 + m + m^2 + \ldots + m^{p(n)} = \frac{1+m^{p(n)+1}}{1+m} = O(2^{p(n)\log_2 m})$, gde je $m = |\Gamma|$. Zato možemo napraviti deterministički Tjuringovu mašinu T' koja će redom nabrajati sve moguće ovakve sertifikate i na njih primenjivati fazu verifikacije mašine T. Vreme izvršavanja mašine T' će biti $O(p(n) \cdot 2^{p(n)\log_2 m}) = O(2^{\log_2(p(n)) \cdot p(n)\log_2 m})$. Dakle, problem A je u klasi EXP.

Dokaz da je ova poslednja inkluzija stroga, tj. da je $\mathrm{NP} \subsetneq \mathrm{EXP}$ ovde izostavljamo. Napomenimo da je ranije pomenuti problem ograničenog zaustavljanja jedan primer problema koji jeste u EXP, a nije u NP.

Pitanje koje je i do danas ostalo otvoreno jeste da li je prva inkluzija stroga ili ne, tj. da li je $\mathrm{P} = \mathrm{NP}$? Ovo je jedan od najznačajnijih otvorenih problema savremenog računarstva. Ako bi bilo $\mathrm{P} = \mathrm{NP}$, onda bi čitava klasa teških problema za koje u ovom trenutku nemamo polinomska rešenja (poput problema SAT) bila polinomski rešiva. Ako bi neko uspeo da dokaže da je $\mathrm{P} = \mathrm{NP}$, to bi bio snažan podsticaj za istraživače da još snažnije pokušavaju da pronađu polinomska rešenja za mnoge praktične probleme za koje danas znamo da su u klasi NP. Još bolje, ako bi neko pronašao efektivan način da se proizvoljan nedeterministički polinomski algoritam transformiše u deterministički polinomski algoritam, tada bi ujedno bio pronađen način za polinomsko rešavanje svih ovih problema. Na žalost, ovo pitanje je otvoreno već decenijama i za sada ne postoje izgledi da će u skorije vreme biti rešeno. Štaviše, nakon višedecenijskog neuspešnog traganja za polinomskim algoritmima za rešavanje mnogih problema

iz klase NP, većina naučnika danas veruje da je P $\neq$ NP, te da će ovo otvoreno pitanje, ako se ikada reši, verovatno biti rešeno odrično.

Još jedno zanimljivo pitanje se tiče odnosa klase co-NP sa ostalim klasama. Najpre, lako se može pokazati da važi:

$$\text{P} \subseteq \text{co-NP}$$

Naime, ako je $\Pi \in$ P, tada je i njegov komplement $\neg\Pi \in$ P, pa je $\neg\Pi \in$ NP. Odavde je $\Pi \in$ co-NP. Slično, ako je $\Pi \in$ co-NP, tada je $\neg\Pi \in$ NP, pa je $\neg\Pi \in$ EXP. Odavde je $\Pi \in$ EXP, pa imamo da je:

$$\text{co-NP} \subseteq \text{EXP}$$

Slično kao i malopre, može se pokazati da je ova poslednja inkluzija stroga, pa imamo:

$$\text{P} \subseteq \text{co-NP} \subsetneq \text{EXP}$$

Sa druge strane, pitanje odnosa klasa NP i co-NP je još uvek otvoreno: i dalje se ne zna da li je NP = co-NP. Naime, postoje problemi poput problema SAT za koje se zna da su u klasi NP, a za njihove komplemente se to još uvek ne zna (niko nije ni dokazao ni opovrgao da je UNSAT u klasi NP). Ono što se zna je da bi iz P = NP sledilo NP = co-NP. Zaista, ako bi bilo P = NP, tada bi važilo i P = co-NP, jer bi svi problemi iz co-NP bili polinomski rešivi (kao komplementi problema iz NP). Dakle, važi implikacija:

$$\text{P} = \text{NP} \Rightarrow \text{NP} = \text{co-NP}$$

kao i implikacija dobjena kontrapozicijom:

$$\text{NP} \neq \text{co-NP} \Rightarrow \text{P} \neq \text{NP}$$

Iz ove druge implikacije sledi da bismo dokazivanje da je P $\neq$ NP mogli svesti na dokazivanje da je NP $\neq$ co-NP (na primer, ako bismo dokazali da problem UNSAT nije u NP, tada bi automatski sledilo P $\neq$ NP).

Primetimo da obrnuta implikacija ne važi: moguće je da bude NP = co-NP, a da i dalje važi P $\neq$ NP.

7.8 NP-kompletnost

U cilju boljeg razumevanja klase NP, potrebno je identifikovati najteže probleme iz ove klase. Međutim, da bismo to uradili, potrebno je najpre nekako definisati šta znači da je neki problem *teži* od nekog drugog problema.

Definicija 7.16. Neka su dati jezici (problemi) A i B nad azbukom Σ i neka je funkcija $f : \Sigma^* \to \Sigma^*$ totalna Tjuring izračunljiva funkcija takva da važi:

- postoji Tjuringova mašina koja izračunava funkciju f u polinomskom vremenu
- za svako $x \in \Sigma^*$ važi da je $x \in A$ ako i samo ako $f(x) \in B$.

Tada za funkciju f kažemo da je *polinomska transformacija jezika A u jezik B*. Ako postoji polinomska transformacija A u B, to ćemo zapisivati sa $A \leq_p B$.

Napomena 7.9. Polinomske transformacije se često nazivaju i *Karpove redukcije*, u čast američkog naučnika Ričarda Karpa (engl. *Richard Karp*, 1935-), koji ih je prvi koristio.

Intuitivno, ako je $A \leq_p B$, tada problem A *nije teži* od problema B, u smislu mogućnosti da bude rešen u polinomskom vremenu. Ovo formalizujemo sledećom lemom.

Lema 7.1. *Ako je $A \leq_p B$, tada iz $B \in P$ sledi $A \in P$.*

Dokaz. Pretpostavimo da postoji Tjuringova mašina T_B koja rešava problem B u polinomskom vremenu. Neka je f polinomska transformacija iz A u B i neka je T_f Tjuringova mašina koja izračunava funkciju f u polinomskom vremenu. Kompozicijom Tjuringovih mašina T_f i T_B dobićemo Tjuringovu mašinu T_A koja rešava problem A. Zaista, za svako $x \in \Sigma^*$, najpre ćemo mašinom T_f izračunati $f(x) \in \Sigma^*$. Pritom $|f(x)|$ će biti polinomski ograničeno u odnosu na $|x|$, s obzirom da mašina T_f u polinomskom vremenu ne može generisati izlaz koji nije polinomski ograničen u odnosu na veličinu ulaza. Dalje ćemo mašinom T_B odrediti da li $f(x)$ pripada B ili ne, a samim tim i da li x pripada A ili ne. Pritom, mašina T_B će završiti svoj rad u polinomskom vremenu u odnosu na $|f(x)|$, što će takođe biti polinomski ograničeno i u odnosu na $|x|$, s obzirom na prethodnu primedbu o polinomskoj veličini $|f(x)|$ u odnosu na $|x|$ i činjenicu da je kompozicija dva polinoma polinom. Kako će obe mašine T_f i T_B završiti svoj rad u broju koraka koji je polinomski ograničen u odnosu na $|x|$, sledi da će i ukupan broj koraka koji je potreban da se utvrdi da li je $x \in A$ biti polinomski ograničen u odnosu na $|x|$. Dakle, ako je $B \in \mathrm{P}$, onda je i $A \in \mathrm{P}$. □

Primetimo da obrnuto ne mora da važi, tj. ako je $A \leq_p B$, tada problem A može biti u P čak i ako problem B nije.

Relacija $\leq_p$ očigledno ima svojstvo refleksivnosti (tj. $A \leq_p A$ za svako $A \in \mathrm{NP}$), jer se svaki problem A može transformisati u samog sebe identičkom funkcijom (koja se izvršava u 0 koraka, što je svakako polinomski ograničeno vreme). Naredna lema govori da relacija $\leq_p$, pored svojstva relfleksivnosti ima i svojstvo tranzitivnosti.

Lema 7.2. *Ako je $A \leq_p B$ i $B \leq_p C$, tada je $A \leq_p C$.*

Dokaz. Ako je f polinomska tranformacija iz A u B, a g polinomska transformacija iz B u C, onda je $g \circ f$ polinomska transformacija iz A u C. Zaista, neka su T_f i T_g Tjuringove mašine koje izračunavu, respektivno, funkcije f i g u polinomskom vremenu. Za svako $x \in \Sigma^*$, $f(x) \in \Sigma^*$ će biti takvo da je $f(x) \in B \Leftrightarrow x \in A$, pri čemu će vrednosti $f(x)$ mašina T_f izračunati u polinomskom vremenu u odnosu na $|x|$. Samim tim, i veličina izlaza $|f(x)|$ će biti polinomski ograničena u odnosu na $|x|$. Dalje, $g(f(x)) \in C \Leftrightarrow f(x) \in B \Leftrightarrow x \in A$, pri čemu se i funkcija g za ulaz $f(x)$ izračunava mašinom T_g u polinomskom vremenu u odnosu na $|f(x)|$, pa samim tim i u odnosu na $|x|$, s obzirom da je kompozicija dva polinoma polinom. Sada se cela procedura transformacije iz A u C svodi na kompoziciju Tjuringovih mašina T_f i T_g koje se obe izvršavaju u polinomskom vremenu u odnosu na $|x|$. □

Sada možemo precizno definisati šta znači biti *najteži* problem u klasi NP: to su oni problemi u koji se mogu transformisati svi drugi problemi iz NP polinomskom transformacijom.

Definicija 7.17. Za problem $A \in \text{NP}$ kažemo da je *NP-kompletan* ako za svaki problem $B \in \text{NP}$ važi da je $B \leq_p A$.

Iz definicije neposredno sledi da ako je neki NP-kompletan problem polinomski rešiv, tada su i svi problemi iz klase NP polinomski rešivi, pa je P = NP. U tome se sastoji značaj izučavanja NP-kompletnih problema – pronalaženje polinomskog algoritma za bilo koji od njih bi automatski značilo da je P = NP. Važi i obrnuto, ako bi neko dokazao da je P $\neq$ NP, tada bismo znali da nijedan NP-kompletan problem nije rešiv u polinomskom vremenu.

Iz definicije je jasno da postoje dva uslova koje problem A mora da ispunjava da bi bio NP-kompetan: da pripada klasi NP i da se svi drugi problemi iz NP mogu polinomski transformisati u A.[5] Ovo prvo je obično lako pokazati. Sa druge strane, dokazati da se svi drugi problemi mogu transformisati u neki dati problem A obično nije tako lako. Naime, potrebno je razmatrati proizvoljnu nedeterminističku Tjuringovu mašinu i pokazati da se jezik koji ona prepoznaje može polinomski transformisati u dati jezik A. Upravo na ovaj način je Stiven Kuk (engl. *Stephen Cook*, slika 7.2) dokazao sledeću čuvenu teoremu.

Teorema 7.5 (Kukova teorema (1971))**.** *SAT problem je NP-kompletan.*

Dokaz. (skica) Neka je dat proizvoljan problem $A \in \text{NP}$. Tada postoji nedeterministička Tjuringova mašina T koja prepoznaje A u polinomskom vremenu. Neka je $p(n)$ polinom koji ograničava vreme izvršavanja mašine T za ulaz veličine n. Jednostavnosti radi, pretpostavimo da Tjuringova mašina ima samo jednu traku. S obzirom da za svako $x \in A$ postoji izvršavanje koje prihvata x u najviše $p(|x|)$ koraka (uključujući i fazu pogađanja i fazu verifikacije), a znajući da glava za čitanje/pisanje može da se pomeri za najviše jednu poziciju u jednom koraku, sledi da će najviše po $p(|x|)$ polja trake levo i desno od početne pozicije glave biti korišćeno od strane Tjuringove mašine tokom njenog rada. Ovo nam daje mogućnost da rad Tjuringove mašine T opišemo konačnom KNF formulom. Uvešćemo tri skupa iskaznih atoma:

- q_{ij} - označava da se mašina u i-tom koraku rada nalazi u stanju q_j. Indeks koraka i može biti od 0 do $p(|x|)$, s obzirom da se izvršavanje zaustavlja u najviše $p(|x|)$ koraka, a indeks j odgovara mogućim stanjima mašine T (kojih ima konačno mnogo)

- h_{ij} - označava da se glava u i-tom koraku nalazi iznad polja na traci sa indeksom j (indeksi j su od $-p(|x|)$ do $p(|x|)$, pri čemu je početna pozicija indeksirana nulom)

- s_{ijk} - označava da se u i-tom koraku u polju sa indeksom j nalazi simbol $a_k \in \Gamma$ (indeksi k odgovaraju simbolima azbuke Γ kojih ima konačno mnogo).

Sada je potrebno konstruisati klauze takve da opisuju rad Tjuringove mašine. Biće nam potrebne sledeće klauze:

[5] Za probleme koji imaju ovo drugo svojstvo, tj. da se svi problemi iz klase NP mogu u njih polinomski transformisati kažemo da su *NP-teški*. Pritom, postoje i NP-teški problemi koji nisu u klasi NP. NP-kompletni problemi su, prema definiciji, upravo NP-teški problemi koji su i sami u klasi NP.

- klauze koje za svaki korak i zahtevaju da se mašina nalazi u tačno jednom stanju, glava se nalazi iznad tačno jednog polja i u svakom polju je upisan tačno jedan simbol
- klauze koje opisuju konfiguraciju mašine na početku faze verifikacije (fiksiramo vrednosti polja na traci desno od početne pozicije tako da sadrže simbole koji kodiraju ulaz x, dok simboli levo od početne pozicije mogu biti proizvoljni, jer odgovaraju slučajnom izboru sertifikata).
- klauze koje opisuju veze između konfiguracije mašine u i-tom i $(i+1)$-vom koraku (na osnovu naredbi Tjuringove mašine)
- klauze koje zahtevaju da se najkasnije u koraku $p(|x|)$ mašina nađe u stanju q_{da}.

Može se pokazati da će ukupan broj ovako definisanih klauza biti polinomski ograničen u odnosu na $|x|$, što znači da će funkcija koja transformiše x u SAT instancu moći da se izvrši u polinomskom vremenu. Pritom, ovako dobijena KNF formula će biti zadovoljiva ako i samo ako postoji nedeterminističko izvršavanje mašine T koje prihvata nisku x. □

Slika 7.2: Američki matematičar Stiven Kuk (1939-)

Zahvaljujući Kukovoj teoremi, SAT problem je postao prvi problem za koji je dokazano da je NP-kompletan. Time su postignute dve značajne stvari. Prvo, pokazano je da postoji bar jedan NP-kompletan problem. Drugo, omogućeno je da se dokazivanje NP-kompletnosti drugih problema postiže znatno lakše. Naime, važi sledeća teorema.

Teorema 7.6. *Neka su $A, B \in NP$, pri čemu je $A \leq_p B$. Ako je problem A NP-kompletan, tada je i B NP-kompletan.*

Dokaz. Problem B je u klasi NP po pretpostavci. Sa druge strane, kako je A NP-kompletan, tada za svaki problem $C \in$ NP važi $C \leq_p A$. Na osnovu leme 7.2, iz $C \leq_p A$ i $A \leq_p B$ sledi $C \leq_p B$, pa je i B NP-kompletan. □

Na osnovu prethodne teoreme, da bismo dokazali da je neki problem B NP-kompletan, ne moramo razmatrati da li se *svaki* problem $C \in$ NP može polinomski transformisati u B. Dovoljno je znati da se *jedan* problem A za koji je već dokazano da je NP-kompletan može polinomski transformisati u B. Ilustrujmo ovo sa par primera.

Primer 7.27. Posmatrajmo problem 3-SAT: u pitanju je specijalni slučaj problema SAT kod koga sve klauze imaju po tačno tri literala. Jasno je da i ovaj problem pripada klasi NP. U opštem slučaju, specijalni slučajevi nekog problema mogu biti lakši od opšteg problema. Ipak, u ovom konkretnom slučaju, ispostavlja se da je 3-SAT i dalje NP-kompletan. Da bismo ovo dokazali, dovoljno je da u polinomskom vremenu transformišemo opšti SAT problem (za koji znamo da je NP-kompletan) u 3-SAT problem. Drugim rečima, potrebno je u pronaći način kako da u polinomskom vremenu proizvoljnu KNF formulu F transformišemo u KNF formulu F' u kojoj sve klauze imaju tačno po tri literala, pri čemu je F' zadovoljiva ako i samo ako je F zadovoljiva. Ideja je da se svaka klauza $C \in F$ zameni skupom tročlanih klauza, uz uvođenje dodatnih iskaznih slova po potrebi. Razmotrimo sledeće slučajeve:

- ako je $C = l$, tj. klauza sa samo jednim literalom l, možemo uvesti dva nova iskazna slova s_1 i s_2 i zameniti klauzu C skupom klauza: $\{s_1 \vee s_2 \vee l, \neg s_1 \vee s_2 \vee l, s_1 \vee \neg s_2 \vee l, \neg s_1 \vee \neg s_2 \vee l\}$. Ovaj skup klauza se može zadovoljiti samo valuacijom u kojoj je l tačno, tj. samo ako je klauza C zadovoljena u toj valuaciji.

- ako je $C = l_1 \vee l_2$, tj. klauza sa dva literala. možemo uvesti jedno novo iskazno slovo s i zameniti klauzu C skupom klauza $\{l_1 \vee l_2 \vee s, l_1 \vee l_2 \vee \neg s\}$. Kao i malopre, da bi ovaj skup klauza bio zadovoljen u nekoj valuaciji, potrebno je da klauza C bude tačna u toj valuaciji.

- ako je klauza $C = l_1 \vee l_2 \vee l_3$, tada je zadržavamo u neizmenjenom obliku, s obzirom da ima tačno tri literala

- ako je klauza $C = l_1 \vee \ldots \vee l_k$, gde je $k \geq 4$, tada je možemo zameniti skupom klauza $\{l_1 \vee l_2 \vee \neg s_1, s_1 \vee l_3 \vee \neg s_2, s_2 \vee l_4 \vee \neg s_3, \ldots, s_{k-3} \vee l_{k-1} \vee l_k\}$, gde su $s_1, \ldots, s_{k-3}$ novouvedena iskazna slova. Ponovo, neka valuacija v zadovoljava ovaj skup klauza ako i samo ako zadovoljava klauzu C.

Konjunkcija svih ovako dobijenih skupova klauza predstavlja formulu F'. Pritom, podrazumeva se da su skupovi novouvedenih iskaznih slova za svake dve originalne klauze iz F disjunktni. Sada će valuacija v zadovoljavati sve klauze iz F' ako i samo ako zadovoljava sve originalne klauze iz F. Pritom, veličina dobijene formule F' je polinomski ograničena u odnosu na veličinu formule F. Zaista, u slučaju jednočlane klauze, broj literala se povećava 12 puta, u slučaju dvočlane 3 puta, dok se za klauze sa $k \geq 4$ dobija skup klauza sa ukupno $3 \cdot (k-2)$ literala, što je $\frac{3 \cdot (k-2)}{k}$ (dakle, manje od 3) puta više u odnosu na polaznu klauzu. Otuda se ukupna veličina povećava ne više od 12 puta, tj. $|F'| \leq 12|F|$, odnosno $|F'| = O(|F|)$. Samim tim, algoritam koji generiše formulu F' na osnovu formule F se može izvršiti u polinomskom broju koraka u odnosu na $|F|$.

Napomena 7.10. Primetimo da je problem 2-SAT, tj. problem zadovoljivosti KNF formula čije sve klauze sadrže tačno po 2 literala, polinomski rešiv. Ostavljamo čitaocu za vežbu da osmisli polinomski algoritam za ovaj problem.

Primer 7.28. Razmotrimo problem *zbira potskupa* (engl. *subset sum*): da li za dati multiskup prirodnih brojeva $X = \{x_1, \ldots, x_k\}$ i dati prirodan broj t postoji potskup $A \subseteq X$ takav da je $\sum_{x_j \in A} x_j = t$? Ovaj problem je u klasi NP. Zaista, za instance za koje je odgovor „da", sertifikat može biti bilo koji skup A čiji elementi u zbiru daju t. Njegova veličina je manja od veličine celog ulaza, pa

je samim tim polinomski ograničena u odnosu na nju. Verifikacija se obavlja jednostavno u polinomskom vremenu – saberemo elemente skupa A i uporedimo zbir sa brojem t.

Da bismo pokazali da je problem zbira potskupa NP-kompletan, razmotrićemo transformaciju problema 3-SAT (za koji smo u prethodnom primeru dokazali da je NP-kompletan) u ovaj problem. Neka je data proizvoljna KNF formula F čije sve klauze imaju po tačno tri literala. Neka je $\{p_1, \ldots, p_m\}$ skup iskaznih slova koje se pojavljuju u F i neka je $\{C_1, \ldots, C_n\}$ skup klauza formule F. Za ovu formulu konstruišimo multiskup brojeva X takvih da svi brojevi tog skupa imaju po $m + n$ cifara u *dekadnom* zapisu. Pritom, ako pozicije u dekadnom zapisu numerišemo brojevima $1, 2, \ldots, m+n$ sa leva u desno, tada će pozicije $1, \ldots, m$ odgovarati redom iskaznim slovima $p_1, \ldots, p_m$, dok će pozicije $m + 1, \ldots, m + n$ odgovarati redom klauzama $C_1, \ldots, C_n$ formule F. Imaćemo sledeće brojeve u X:

- za svako iskazno slovo p_i uvodimo dva broja a_i i a'_i: oba broja će imati cifru 1 na poziciji i, dok će na pozicijama koje odgovaraju ostalim iskaznim slovima imati cifru 0. Takođe, broj a_i će imati cifru 1 na pozicijama koje odgovaraju onim klauzama koje sadrže literal p_i, dok će cifre na pozicijama koje odgovaraju ostalim klauzama biti 0. Slično, broj a'_i će imati cifru 1 na pozicijama koje odgovaraju klauzama koje sadže literal $\neg p_i$, dok će na pozicijama koje odgovaraju ostalim klauzama imati cifru 0

- za svaku klauzu C_j imaćemo dva broja c^1_j i c^2_j. Oba broja će biti jednaka među sobom i imaće samo cifru 1 na poziciji koja odgovara klauzi C_j, dok će na ostalim pozicijama imati nule.

Neka je broj t od $m + n$ cifara u dekadnom zapisu dobijen na sledeći način:

- na svim pozicijama $1, \ldots, m$ nalazi se cifra 1

- na svim pozicijama $m + 1, \ldots, m + n$ nalazi se cifra 3

Za ovako dobijenu instancu problema zbira potskupa važiće da postoji potskup čiji je zbir jednak t ako i samo ako je polazna formula F zadovoljiva. Zaista, ako je F zadovoljiva, tada postoji valuacija v u kojoj su sve klauze $C_1, \ldots, C_n$ tačne. Za svako iskazno slovo p_i u skup A uzimamo broj a_i ako je $v(p_i) = 1$, a a'_i ako je $v(p_i) = 0$. Sada će zbir ovih brojeva biti broj čije su cifre na pozicijama $1, \ldots, m$ jednake 1 (jer smo za svako od iskaznih slova p_i uzeli tačno jedan od brojeva a_i ili a'_i, a to su jedini brojevi iz X koji na poziciji i imaju cifru 1). Sa druge strane, kako će svaka od klauza biti zadovoljena, zbir svih ovih brojeva će na pozicijama $m + 1, \ldots, m + n$ imati cifre koje su 1, 2 ili 3 (cifra na poziciji $m + j$ odgovara broju literala klauze C_j koji su tačni u valuaciji v, a to može biti najviše 3, a mora biti bar 1, jer je svaka klauza zadovoljena valuacijom v). Za one klauze C_j za koje je cifra na poziciji $m + j$ jednaka 2, tada ćemo u skup A dodati i broj c^1_j, a ako je cifra jednaka 1, dodaćemo oba broja c^1_j i c^2_j. Kako su pitanju brojevi zapisani u osnovi 10, neće biti prenosa pri sabiranju, pa će zbir ovako izabranog skupa A biti upravo jednak broju t.

Obrnuto, ako postoji skup $A \subseteq X$ takav da je zbir njegovih elemenata jednak broju t, tada će za svako iskazno slovo p_i u tom skupu biti tačno jedan od brojeva a_i ili a'_i (ako bi bila oba, tada bi odgovarajuća cifra bila jednaka 2, a ako ne bi bio ni jedan, cifra bi bila 0, jer prenos nije moguć). Neka je valuacija v takva

da je $v(p_i) = 1$ ako je $a_i \in A$, a $v(p_i) = 0$ ako je $a'_i \in A$. Kako su sve cifre broja t koje odgovaraju klauzama jednake 3, to znači da će za svaku poziciju $m + j$ bar jedan od brojeva a_i ili a'_i koji su u skupu A imati cifru 1 na toj poziciji (jer u skupu X imamo samo još dva broja c_j^1 i c_j^2 koji imaju cifru 1 na toj poziciji). Otuda bar jedan od literala koji su tačni u valuaciji v pripada klauzi C_j. Dakle, sve klauze su tačne u valuaciji v, pa je formula F zadovoljiva.

Najzad, primetimo da je veličina dobijene instance problema zbira potskupa polinomski ograničena u odnosu na veličinu polazne formule F (ostavljamo čitaocu za vežbu da se u to uveri). Dakle, imamo polinomsku transformaciju 3-SAT problema u problem zbira potskupa. Otuda je problem zbira potskupa NP-kompletan.

Primer 7.29. Razmotrimo sada ranije uvedeni problem particije, za koji smo konstatovali da pripada klasi NP. Primetimo da je problem particije zapravo specijalni slučaj problema zbira potskupa (potrebno je samo za T uzeti polovinu zbira celog skupa). Naravno, kao što smo ranije konstatovali, specijalni slučaj često može biti lakši od opšteg problema. Ipak, u ovom slučaju, problem particije je i dalje NP-kompletan. Da bismo to pokazali, svešćemo opšti problem zbira potskupa na svoj specijalni slučaj – problem particije. Naime, neka je X proizvoljan multiskup brojeva i neka je t proizvoljan broj. Neka je s zbir svih elemenata skupa X. Formirajmo skup X' tako što u skup X dodamo još dva broja: $u = s + t$ i $v = 2s - t$. Sada u skupu X postoji potskup čiji je zbir elemenata jednak t ako i samo ako skup X' možemo podeliti na dva potskupa sa jednakim zbirovima. Zaista, zbir svih elemenata skupa X' je jednak $s + u + v = s + s + t + 2s - t = 4s$. Pretpostavimo da postoji skup $A \subseteq X$ takav da je zbir elemenata skupa A jednak t. Sada dodavanjem broja v u taj skup dobijamo skup čiji je zbir elemenata jednak $t + 2s - t = 2s$, što je upravo polovina zbira skupa X'. Otuda je skup X' moguće podeliti na dva skupa jednakih zbirova. Obratno, ako je skup X' moguće podeliti na dva skupa jednakih zbirova, to znači da postoji skup $A \subseteq X'$ takav da je zbir njegovih elemenata jednak $2s$. Skup A mora sadržati broj u ili broj v (jer je zbir svih ostalih elemenata skupa X' jednak s), ali ne oba (jer bi onda zbir bio bar $3s$). Ako je $u \in A$, tada odbacivanjem ovog elementa dobijamo skup $A' \subseteq X$ čiji je zbir jednak $2s - s - t = s - t$, pa je $B = X \setminus A'$ skup čiji je zbir elemenata jednak t. Slično, ako je $v \in A$, tada se odbacivanjem ovog elementa dobija skup $A'' \subseteq X$ čiji je zbir jednak $2s - 2s + t = t$. Dakle, u oba slučaja postoji potskup od X čiji je zbir jednak t.

Jasno je da je opisana transformacija skupa X u skup X' polinomska. Otuda je problem particije takođe NP-kompletan.

Primetimo da ako imamo problem koji je u klasi NP, a znamo da je neki njegov specijalni slučaj NP-kompletan, onda je i opšti problem tim pre NP-kompletan. Zaista, uvek možemo ,,transformisati" specijalni slučaj u opšti problem u polinomskom vremenu – transformacija je identička funkcija koja se uvek izvršava u polinomskom vremenu (u 0 koraka). Intuitivno, proceduru za rešavanje opšteg problema možemo u neizmenjenom obliku koristiti i za instance specijalnog slučaja, tj. nije neophodna nikakva transformacija instanci.

Primer 7.30. Razmotrimo problem zadovoljivosti *proizvoljne* iskazne formule. Ovaj problem je u klasi NP, jer se uvek može u polinomskom vremenu proveriti da li neka valuacija (koja predstavlja sertifikat) zadovoljava datu formulu. Sa druge strane, problem SAT je specijalni slučaj ovog problema. Otuda za-

ključujemo da je opšti problem iskazne zadovoljivosti NP-kompletan, jer se SAT problem (za koji znamo da je NP-kompletan) može trivijalno transformisati u njega u polinomskom vremenu.

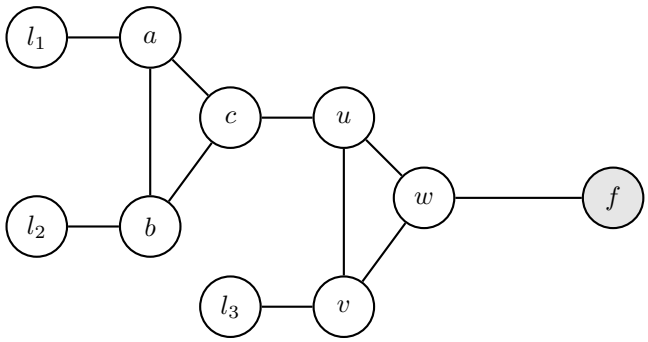
Primer 7.31. Razmotrimo problem bojenja grafa u tri boje. Za ovaj problem smo ranije utvrdili da pripada klasi NP. Da bismo dokazali da je NP-kompletan, možemo razmotriti transformaciju problema 3-SAT u problem bojenja grafa u tri boje. Neka je data proizvoljna KNF formula F u kojoj sve klauze imaju po tri literala. Neka su $p_1, \ldots, p_m$ iskazna slova koja se pojavljuju u F i neka su $C_1, \ldots, C_n$ klauze formule F. Konstruišimo neusmeren graf G na sledeći način:

- *osnovni trougao* čine čvorovi označeni sa t, f i n koji su međusobno povezani. Ova tri čvora će samim tim morati da se oboje u tri različite boje, pa ih možemo upravo identifikovati sa tim bojama. Intuitivno, boja t znači *tačno*, boja f znači *netačno*, a boja n označava treću, „neutralnu" boju.
- za svako iskazno slovo p_i imaćemo dva čvora – jedan koji odgovara literalu p_i i jedan koji odgovara literalu $\neg p_i$. Ova dva čvora će biti povezana međusobno, kao i sa čvorom n. Na ovaj način obezbeđujemo da se jedan od ova dva čvora oboji bojom t a drugi bojom f, što će odgovarati izboru zadovoljavajuće valuacije za formulu F
- za svaku klauzu $l_1 \vee l_2 \vee l_3$ konstruišemo podgraf koji simulira izračunavanje disjunkcije (slika 7.3). Čvorovi označeni sa l_1, l_2 i l_3 su prethodno uvedeni čvorovi koji odgovaraju ovim literalima, a čvor f je odgovarajući čvor iz osnovnog trougla. Ovaj podgraf sprečava da se sva tri čvora l_1, l_2 i l_3 oboje u boju f, tj. bar jedan se mora obojiti u boju t.

Broj čvorova ovog grafa je $6n+2m+3 = O(n)$ (jer je $m \leq 3n$), što je polinomski ograničeno u odnosu na veličinu formule F (koja se može izraziti brojem klauza n). Dakle, graf se može konstruisati u polinomskom vremenu. Pritom, ako je formula F zadovoljiva, tada postoji ispravno bojenje grafa. Zaista, neka je v valuacija u kojoj je F tačna. Za svako iskazno slovo p_i obojmo čvor p_i u boju t ako je $v(p_i) = 1$, a u boju f u suprotnom. Obojmo čvor $\neg p_i$ u suprotnu boju. Kako je za svaku klauzu $l_1 \vee l_2 \vee l_3$ bar jedan od literala tačan u v, odgovarajući podgraf sa slike 7.3 je moguće obojiti na ispravan način. Na primer, ako je l_1 obojen u t, tada se čvor a može obojiti u f, a čvor b u n (jer l_2 sigurno nije obojen u n). Odavde će čvor c biti obojen u t. Sličnim rezonovanjem, čvor u će biti obojen u f, čvor v u n, a čvor w ponovo u t. Kako ovo možemo uraditi za svaki podgraf, imamo ispravno bojenje grafa.

Obrnuto, neka postoji ispravno bojenje ovog grafa u tri boje t, f i n. Kako znamo da su čvorovi p_i i $\neg p_i$ povezani međusobno, kao i sa čvorom n, oni će biti obojeni u boje t i f (ali će biti međusobno različito obojeni). Definišimo valuaciju v takvu da je $v(p_i) = 1$ ako i samo ako je čvor p_i obojen u boju t. Ova valuacija će zadovoljavati formulu F. Zaista, pretpostavimo suprotno – da postoji klauza $l_1 \vee l_2 \vee l_3$ koja je netačna u v. Tada bi u odgovarajućem podgrafu sva tri ulazna čvora bila obojena bojom f. Odatle bi čvorovi a i b bili obojeni bojama t i n (ali različito obojeni), pa će čvor c biti obojen bojom f. Kako je i l_3 obojen bojom f, sledilo bi da i čvorovi u i v moraju biti obojeni bojama t i n, pa će i čvor w biti obojen bojom f, što nije moguće, jer je povezan sa čvorom f. Dakle, konstruisana valuacija zadovoljava sve klauze formule F.

Kako smo problem 3-SAT sveli na bojenje grafa u 3 boje, sledi da je i problem bojenja grafa NP-kompletan.



Slika 7.3: Podgraf za predstavljanje klauze $l_1 \vee l_2 \vee l_3$

Primer 7.32. Možemo razmatrati i opštiji problem *bojenja grafa u k boja*: kod ovog problema, na ulazu je neusmeren graf G i broj k, a pitamo se da li je moguće obojiti svaki od čvorova grafa jednom od k datih boja tako da nikoja dva susedna čvora ne budu obojena istom bojom. Jasno je da će i ovaj problem biti u klasi NP, jer za one instance za koje je odgovor potvrdan uvek možemo uzeti neko od ispravnih bojenja kao sertifikat, koji zatim možemo verifikovati u polinomskom vremenu. Sa druge strane, ovaj problem je i NP-kompletan, jer za njegov specijalni slučaj koji se dobija fiksiranjem vrednosti parametra k na vrednost 3 znamo da je NP-kompletan.

Primetimo da je specijalni slučaj za $k = 2$ polinomski rešiv. Naime, može se pokazati da je graf 2-obojiv ako i samo ako ne sadrži cikluse neparne dužine (za vežbu). Sada se provera 2-obojivosti svodi na prost obilazak grafa u traganju za ciklusima neparne dužine (linearna složenost).

7.8.1 Primeri NP-kompletnih problema

U ovom odeljku navodimo primere nekih poznatih NP-kompletnih problema iz različitih oblasti, bez dokaza njihove NP-kompletnosti. Napomenimo da će mnogi od ovih problema odlučivanja u svojoj formulaciji imati kao jedan od ulaznih parametara i neki broj k, a pitanje će biti oblika „da li postoji neka struktura veličine bar k ili najviše k?". Ovakvi problemi po pravilu imaju i svoju *optimizacionu formu*: „odrediti najveće (najmanje) k takvo da postoji tražena struktura te veličine". Ukoliko imamo algoritam za rešavanje datog problema odlučivanja, lako možemo konstruisati algoritam za rešavanje odgovarajućeg optimizacionog problema: potrebno je samo binarnom pretragom pronaći najveće (najmanje) k za koji je odgovor potvrdan (uzastopnim pozivanjem algoritma odlučivanja za različito k). Pritom, ako algoritam za rešavanje problema odlučivanja u slučaju potvrdnog odgovora ujedno i vraća traženu strukturu veličine k, tada je pomenutim postupkom moguće odrediti traženu strukturu najveće (najmanje) veličine. Važi i obratno – ako imamo algoritam za rešavanje optimizacionog problema, tada za svako k možemo odrediti da li postoji tražena struktura veličine najviše (ili bar) k, prosto tako što uporedimo k sa optimalnom vrednošću dobijenom pomoću optimizacionog algoritma.[6] Dakle, ove dve

[6]Za ovako zadate optimizacione probleme kažemo da su *NP-teški*, jer se svaki problem iz klase NP može na njih svesti u polinomskom vremenu. Ipak, oni nisu u klasi NP, s obzirom

varijante problema se uvek mogu jednostavno svesti jedna na drugu, pa je u praksi svejedno koju ćemo rešavati.

Pokrivanje grana čvorovima. Neka je dat neusmeren graf $G = (V, E)$. Za skup $V' \subseteq V$ kažemo da je *pokrivač veličine k* (ili *k-pokrivač*), ako je $|V'| = k$ i za svaku granu $(u, v) \in E$ važi da je bar jedan od čvorova u i v u skupu V'. Problem k-pokrivača glasi: ,,za dati neusmerni graf G i dati broj k, da li u grafu G postoji k-pokrivač?". Ovaj problem je NP-kompletan, a dokaz se može izvosti transformacijom 3-SAT problema. Optimizaciona verzija ovog problema glasi: ,,Pronaći pokrivač sa najmanjim brojem čvorova u datom grafu G."

Klika. Pod *klikom veličine k* (ili *k-klikom*) u neusmerenom grafu $G = (V, E)$ podrazumevamo potskup $V' \subseteq V$ od k čvorova takvih da su svaka dva čvora $u, v \in V'$ međusobno susedna. Problem k-klika glasi: ,,da li u datom neusmerenom grafu G postoji klika date veličine k?" Ovaj problem je NP-kompletan, a dokaz se može izvesti transformacijom problema k-pokrivača. Optimizaciona verzija ovog problema glasi: ,,Pronaći kliku najveće veličine u datom grafu G".

Hamiltonov ciklus. *Hamiltonov ciklus* u neusmerenom grafu G pretstavlja ciklus koji sadrži svaki od čvorova grafa tačno po jednom. Problem ,,Da li u datom neusmerenom grafu G postoji Hamiltonov ciklus?" je NP-kompletan.

Napomena 7.11. Problem Hamiltonovog ciklusa se može razmatrati i za usmerene grafove. Pokazuje se da je i ta varijanta problema NP-kompletna. Takođe, postoji i varijanta problema poznata kao *Hamiltonov put*, gde se zahteva da postoji put koji sadrži svaki od čvorova grafa tačno po jednom (dakle, ne mora biti ciklus, tj. ne mora postojati grana između prvog i poslednjeg čvora u putu). I ova varijanta je NP-kompletna.

Sa druge strane, srodan problem *Ojlerovog ciklusa* – ,,da li postoji ciklus koji svaku *granu* sadrži tačno po jednom?" je polinomski rešiv.

Trgovački putnik. *Potpun težinski neusmereni graf G* je neusmeren graf u kome:

- postoji grana između svaka dva čvora grafa
- svakoj grani e je dodeljena nenegativna *težina* $w(e)$

Problem trgovačkog putnika glasi: ,,Za dati potpun težinski neusmereni graf G i dati broj k da li postoji Hamiltonov ciklus ukupne težine manje ili jednake od k?". Intuitivno, ako čvorovi predstavljaju gradove, a grane puteve između njih (pri čemu težine grana odgovaraju rastojanjima između gradova), da li trgovački putnik može obići sve gradove, a da ne pređe put duži od k? Ovaj problem je takođe NP-kompletan, jer se problem Hamiltonovog ciklusa može posmatrati kao njegov specijalni slučaj (svim granama se dodeli težina 1, a zatim pitamo da li postoji Hamiltonov ciklus ukupne težine manje ili jednake n, gde je n broj čvorova grafa). Optimizaciona verzija ovog problema glasi: ,,U potpunom težinsko grafu G, pronaći Hamiltonov ciklus najmanje moguće težine."

---

da nisu problemi odlučivanja.

Izomorfni podgraf. Za dva grafa $G_1 = (V_1, E_1)$ i $G_2 = (V_2, E_2)$ kažemo da su *izomorfni* ako postoji bijektivno preslikavanje $f : V_1 \to V_2$ takvo da je $(u, v) \in E_1$ ako i samo ako je $(f(u), f(v)) \in E_2$. *Problem izomorfnog podgrafa* glasi: „Za date grafove $G_1 = (V_1, E_1)$ i $G_2 = (V_2, E_2)$, da li postoji podgraf $G_1' = (V_1', E_1')$ grafa G_1 (tj. graf za koji važi da je $V_1' \subseteq V_1$ i $E_1' \subseteq E_1$), takav da su G_1' i G_2 izomorfni?" Ovaj problem je NP-kompletan. Ovo sledi iz činjenice da je problem k-klika njegov specijalni slučaj, gde za graf G_2 uzimamo potpuni graf sa k čvorova.

Napomena 7.12. Primetimo da se problem *izomorfizma grafova* – „Da li je dati graf G_1 izomorfan datom grafu G_2?" može svesti na problem izomorfnog podgrafa: ako grafovi G_1 i G_2 imaju različite brojeve čvorova, tada oni definitivno nisu izomorfni, u suprotnom treba samo ispitati da li je G_2 izomorfan nekom podgrafu grafa G_1 (ovo je moguće samo ako je izomorfan sa samim G_1, s obzirom da su im brojevi čvorova jednaki). Ipak, za problem izomorfizma grafova još uvek nije dokazano ni da je NP-kompletan ni da je u klasi P.

Najduži put. Neka je dat neusmeren težinski graf $G = (V, E)$, pri čemu su težine grana nenegativne. *Prost put* u grafu G je niz *međusobno različitih* čvorova $v_1, \ldots, v_m$ takav da je $(v_i, v_{i+1}) \in E$ za svako $i \in \{1, \ldots, m-1\}$. Razmotrimo problem: „da li u datom grafu G postoji prost put od datog čvora s do datog čvora t ukupne dužine veće od datog broja k?". Može se pokazati da je ovaj problem NP-kompletan. Njegova optimizaciona varijanta glasi: „Pronaći najduži prost put između dva data čvora u datom grafu".

Napomena 7.13. Primetimo da je sličan problem ispitivanja da li postoji prost put između dva zadata čvora koji je dužine *najviše* k polinomski rešiv.

Trodimenziono uparivanje. Neka su data tri disjunktna konačna skupa U, V i W sa po tačno n elemenata, kao i skup $M \subseteq U \times V \times W$ *dopustivih* trojki. Problem *trodimenzionog uparivanja* se može formulisati ovako: „Da li postoji $M' \subseteq M$ takav da je $|M'| = n$ i za svake dve trojke (x, y, z) i (x', y', z') iz M' važi $x \neq x'$, $y \neq y'$ i $z \neq z'$?". Za ovaj problem je dokazano da je NP-kompletan. Intuitivno, skup M možemo razumeti kao skup mogućih tročlanih „timova", a zadatak je odabrati timove tako da nijedan igrač ne igra u dva različita tima.

Napomena 7.14. Primetimo da je sličan problem *dvodimenzionog uparivanja* polinomski rešiv.

Disjunktni skupovi. Neka je data kolekcija konačnih skupova $\mathcal{C}$, kao i broj k. Razmotrimo problem: „Da li postoji bar k disjunktnih skupova u $\mathcal{C}$?". Ovaj problem je NP-kompletan. Optimizaciona verzija ovog problema glasi: „U datoj kolekciji konačnih skupova $\mathcal{C}$, pronaći najveći mogući skup međusobno disjunktnih skupova".

Podela skupova. Neka je dat konačan skup S i kolekcija njegovih konačnih potskupova $\mathcal{C}$. Problem glasi: „Da li je moguće skup S podeliti na dva disjunktna potskupa S_1 i S_2, tako da ni za jedan skup $C \in \mathcal{C}$ nije ni $C \subseteq S_1$ ni $C \subseteq S_2$?". Ovaj problem je takođe NP-kompletan.

Pogađajući skup. Neka je dat konačan skup S, kolekcija njegovih konačnih potskupova $\mathcal{C}$ kao i pozitivan broj k. Problem glasi: ,,Da li postoji potskup $S' \subseteq S$ takav da je $|S'| \leq k$ i da je $S' \cap C \neq \varnothing$ za svako $C \in \mathcal{C}$?". Takav skup S' nazivamo i *pogađajući skup* (engl. *hitting set*). I ovaj problem je NP-kompletan. Njegova optimizaciona verzija glasi: ,,Odrediti pogađajući skup najmanje moguće kardinalnosti.".

Problem pakovanja. Pretpostavimo da imamo konačan skup A pri čemu je svakom elementu $a \in A$ pridružena pozitivna celobrojna *veličina* $s(a)$. Takođe, neka je b pozitivan ceo broj. *Problem pakovanja* (engl. *bin packing problem*) se može formulisati ovako: ,,Za date pozitivan broj k, da li je moguće skup A particionisati u k disjunktnih potskupova $A_1, \ldots, A_k$ tako da je zbir veličina elemenata u svakom skupu A_i najviše b?". Ovaj problem je NP-kompletan. Intuitivno, elemente skupa A možemo razumeti kao predmete različitih veličina koje pokušavamo da spakujemo u kutije veličine b. Optimizaciona verzija problema se sastoji u određivanju najmanjeg broja k za koji je takvo pakovanje moguće, tj. cilj je spakovati predmete u što manji broj kutija.

Problem ranca. Neka je dat konačan skup A i neka je svakom elementu $a \in A$ pridružena pozitivna celobrojna *veličina* $s(a)$ i pozitivna celobrojna *vrednost* $v(a)$. Neka je dat i pozitivan broj b. *Problem ranca* (engl. *knapsack problem*) se može formulisati ovako: ,,Za dati pozitivan broj k, da li postoji potskup $A' \subseteq A$ takav da je $\sum_{a \in A'} s(a) \leq b$ i $\sum_{a \in A'} v(a) \geq k$?" Ovaj problem je NP-kompletan. Intuitivno, imamo skup predmeta različitih veličina i vrednosti, kao i ranac veličine b. Pitamo se da li je u ranac moguće spakovati predmete u ukupnoj vrednosti bar k. Optimizaciona varijanta podrazumeva pakovanje predmeta u ranac tako da ukupna vrednost spakovanih predmeta bude što je moguće veća.

Sekvencijalno raspoređivanje poslova. Neka je dat skup poslova T, takav da je svakom poslu $t \in T$ pridruženo *trajanje* $l(t)$, minimalno vreme početka $r(t)$ i rok završetka $d(t)$ (sve pozitivni celi brojevi). Razmatramo problem: ,,Da li je moguće sve poslove iz T sekvencijalno izvršiti tako da ni jedan posao $t \in T$ ne počinje pre trenutka $r(t)$, kao i da se svaki posao $t \in T$ završava najkasnije do roka $d(t)$?". Ovaj problem je NP-kompletan. Intuitivno, potrebno je da jedna osoba završi neki skup poslova, pri čemu su zadata ograničenja u vidu vremenskih intervala u kojima se svaki od poslova mora započeti i završiti.

Paralelno raspoređivanje poslova. Neka je m broj radnika i T skup poslova, pri čemu svaki posao $t \in T$ ima pridruženo *trajanje* $l(t)$ (pozitivan ceo broj). Problem raspoređivanja poslova možemo formulisati ovako: ,,Da li je moguće poslove iz T rasporediti na m radnika tako da se svi poslovi završe do zadatog *roka* d?" Ovaj problem je NP-kompletan. Njegova optimizaciona verzija glasi: ,,Rasporediti poslove iz T na m radnika tako da se svi poslovi završe u najkraćem mogućem roku".

Napomena 7.15. Umesto m radnika, možemo razmatrati m procesora na koje treba rasporediti date programe tako da se ceo posao završi u najkraćem mogućem vremenu.

Napomena 7.16. Prethodna dva problema su samo primeri čitave klase različitih *problema raspoređivanja* za koje je dokazano da su NP-kompletni.

7.9 Klasa PSPACE

U ovom poglavlju razmatramo prostornu složenost problema, zadržavajući se na problemima koji su rešivi u polinomskom prostoru.

Definicija 7.18. Klasu *prostorne polinomske složenosti* PSPACE čine svi problemi odlučivanja za koje postoji procedura odlučivanja čija je prostorna složenost $O(p(n))$ za neki polinom p.

Teorema 7.7. *Za proizvoljan problem odlučivanja* Π*, ako je* $\Pi \in P$ *tada je* $\Pi \in$ *PSPACE.*

Dokaz. Ako je problem Π odlučiv u polinomskom vremenu, tada postoji Tjuringova mašina T čiji je broj koraka ograničen nekim polinomom $p(n)$, gde je n veličina ulaza. Kako Tjuringova mašina u svakom koraku pristupa po jednom polju na svakoj traci Tjuringove mašine, ukupan broj polja koje Tjuringova mašina može koristiti u toku svog rada je takođe $O(p(n))$, pa je i prostorna složenost polinomska. □

Dakle, važi inkluzija:

$$\text{P} \subseteq \text{PSPACE}$$

Pritom, u ovom trenutku nije poznato da li važi jednakost, tj. da li je P = PSPACE. Potpuno analogni zaključci važe i za klase NP i co-NP.

Teorema 7.8. *Za proizvoljan problem odlučivanja* Π*, ako je* $\Pi \in$ *NP (*$\Pi \in$ *co-NP) tada je* $\Pi \in$ *PSPACE.*

Dokaz. Dokaz ide slično kao i za klasu P. □

Prema tome, imamo inkluzije:

$$\text{NP} \subseteq \text{PSPACE}$$

i

$$\text{co-NP} \subseteq \text{PSPACE}$$

Pritom, ni za ove inkluzije za sada nije utvrđeno da li su stroge.

Kako u klasi NP postoje mnogi problemi za koje u ovom trenutku nisu poznati deterministički polinomski algoritmi, u ovom trenutku je verovatnije da je P $\neq$ PSPACE. Ipak, to još uvek niko nije dokazao.

Primetimo da bi iz P = PSPACE sledilo P = NP. Obratno ne mora da važi, tj. mogli bi da važi P = NP a da i dalje ne važi P = PSPACE.

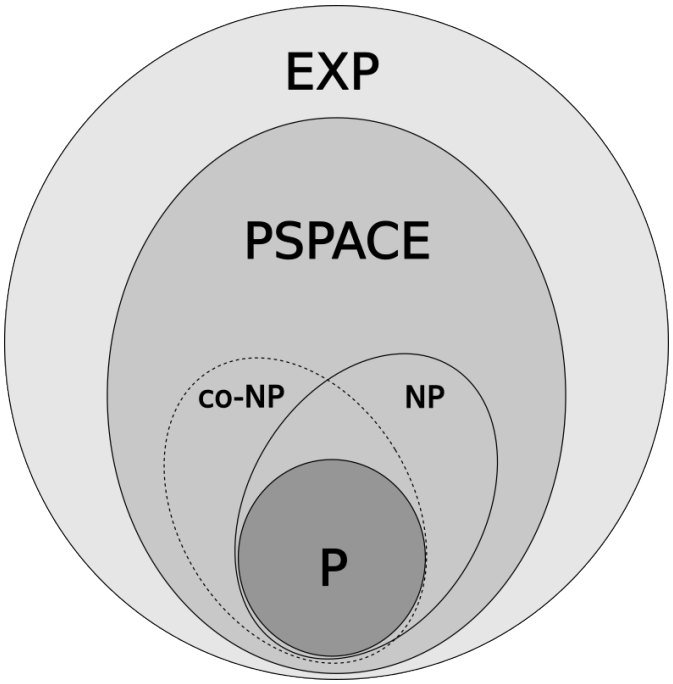
Teorema 7.9. *Za proizvoljan problem odlučivanja* Π*, ako je* $\Pi \in$ *PSPACE, tada je* $\Pi \in$ *EXP.*

Dokaz. Ako je $\Pi \in$ PSPACE, tada postoji deterministička Tjuringova mašina koji za ulaz veličine n koristi prostor ograničen nekim polinomom $p(n)$. Broj različitih *konfiguracija* Tjuringove mašine u prostoru ograničenom sa $p(n)$ je $O(|\Gamma|^{p(n)} \cdot |Q|) = O(2^{p(n)\cdot \log_2 |\Gamma|})$. Pod pretpostavkom da je problem odlučiv, za svaki ulaz će mašina u konačnom broju koraka ući u završno stanje q_{da} ili q_{ne}. To znači da nije moguće da se mašina dva puta nađe u istoj konfiguraciji, jer bi to značilo ,,beskonačnu petlju", pa se mašina ne bi zaustavljala. Dakle, broj koraka je ograničen brojem različitih konfiguracija mašine u prostoru ograničenom sa $p(n)$, tj. sa $O(2^{p(n)\cdot \log_2 |\Gamma|})$. Ovo znači da je $\Pi \in$ EXP. □

Iz prethodne teoreme sledi inkluzija:

$$\text{PSPACE} \subseteq \text{EXP}$$

Pritom, još uvek nije poznato da li je PSPACE = EXP (pretpostavlja se da nije, ali to još uvek niko nije dokazao).



Slika 7.4: Pretpostavljeni odnos između klasa P, NP, co-NP, PSPACE i EXP

Glava 8

Lambda račun

U ovom poglavlju se upoznajemo sa još jednim formalizmom za zasnivanje algoritama, odnosno izračunljivih funkcija. U pitanju je *lambda račun* – jedan od najstarijih formalizama koji je osmislio Alonzo Čerč (slika 8.1) tokom tridesetih godina 20. veka. Za razliku od Tjuringovih mašina i URM programa koji algoritme opisuju na imperativni način – u obliku niza naredbi kojima se menja stanje programa, u lambda računu se izračunljive funkcije opisuju izrazima koji nam govore šta funkcija radi sa svojim argumentima, a izračunavanje se svodi na *redukciju* izraza koja nastaje kada se funkcija primeni na svoje argumente. Lambda račun je teorijski osnov za *funkcionalne programske jezike*, među kojima je najstariji jezik *LISP*, a u modernije spadaju *ML* i *Haskell*.

Slika 8.1: Američki matematičar Alonzo Čerč (1903-1995)

8.1 Sintaksa lambda izraza

Definicija 8.1. Neka je dat prebrojiv skup *promenljivih* V. *Lambda izraz nad* V se dobija konačnom primenom sledećih pravila:

- ako je $x \in V$, tada je x lambda izraz
- ako su E i F lambda izrazi, tada je $E\ F$ lambda izraz (*aplikacija izraza* E *na* F)

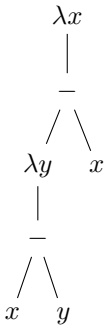
- ako je E lambda izraz, a $x \in V$, tada je $\lambda x.E$ lambda izraz (*apstrakcija izraza E po x*)

Skup svih lambda izraza označavaćemo sa Λ.

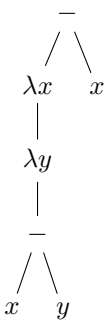
Napomena 8.1. Intuitivno, operator apstrakcije kreira funkciju po x, tj. $\lambda x.E$ je funkcija po x čije je telo dato izrazom E. Sa druge strane, aplikacija $E\ F$ predstavlja primenu funkcije E na argument F. Ovaj smisao lambda izraza ćemo u operativnom smislu, precizno definisati nešto kasnije.

Napomena 8.2. Operator aplikacije je levo asocijativan i višeg je prioriteta od operatora apstrakcije. To znači da npr. izraz $\lambda x.\lambda y.x\ y\ x$ sintaksno odgovara izrazu $\lambda x.(\lambda y.((x\ y)\ x))$, odnosno, predstavljen je sledećim apstraktnim sintaksnim stablom:

pri čemu smo čvorove koji predstavljaju operator aplikacije označili simbolom $-$. Sa druge strane, izrazu $\lambda x.(\lambda y.x\ y)\ x$ odgovara sledeće sintaksno stablo:



dok izrazu $(\lambda x.\lambda y.x\ y)\ x$ odgovara sledeće sintaksno stablo:



Napomena 8.3. Uzastopne apstrakcije poput $\lambda x.\lambda y.\lambda z.x\ y\ z$ ćemo često kraće zapisivati kao $\lambda xyz.x\ y\ z$.

Operator apstrakcije je sintaksno sličan kvantifikatorima u logici prvog reda. On je niskog prioriteta i njegovo dejstvo se, ako se zagradama ne odredi drugačije, prostire do kraja izraza. Slično kao i kvantifikatori u logici prvog reda, operator apstrakcije takođe *vezuje* slobodna pojavljivanja odgovarajuće promenljive u podizrazu na koji se primenjuje. Preciznije, imamo sledeću definiciju.

Definicija 8.2. *Slobodna* i *vezana* pojavljivanja promenljivih u lambda izrazima definišemo na sledeći način:

- u izrazu x, pojavljivanje promenljive x je slobodno
- pojavljivanja promenljivih koja su slobodna (vezana) u E i F su slobodna (vezana) i u izrazu $E\ F$
- ako je pojavljivanje promenljive y različite od x slobodno (vezano) u E, tada je slobodno (vezano) i u $\lambda x.E$
- slobodna pojavljivanja promenljive x u E su vezana u $\lambda x.E$ tom apstrakcijom. Vezana pojavljivanja promenljive x u E ostaju vezana i u $\lambda x.E$ istom apstrakcijom kojom su bila vezana i u izrazu E.

Za izraz kažemo da je *zatvoren* (ili *kombinator*) ako ne sadrži slobodne promenljive.

Primer 8.1. Neka je dat izraz $x\ (\lambda x.x\ (\lambda x.x\ (\lambda y.y\ z)))$. U ovom izrazu je prvo pojavljivanje promenljive x slobodno, drugo je vezano prvom apstrakcijom po x, dok je treće vezano drugom apstrakcijom po x. Jedino pojavljivanje promenljive y je vezano apstrakcijom po y, dok je pojavljivanje promenljive z slobodno.

Takođe, kao u logici prvog reda, definišemo sintaksnu operaciju *zamene promenljive izrazom*.

Definicija 8.3. Zamenu promenljive x izrazom F u izrazu E (u oznaci $E[x \mapsto F]$) definiv semo na sledeći način:

- $x[x \mapsto F] = F$
- $y[x \mapsto F] = y$ (za $y \neq x$)
- $(E_1\ E_2)[x \mapsto F] = E_1[x \mapsto F]\ E_2[x \mapsto F]$
- $(\lambda x.E)[x \mapsto F] = \lambda x.E$
- $(\lambda y.E)[x \mapsto F] = \lambda y.E[x \mapsto F]$, ako F ne sadrži slobodna pojavljivanja promenljive y
- $(\lambda y.E)[x \mapsto F] = \lambda z.E[y \mapsto z][x \mapsto F]$, ako F sadrži slobodna pojavljivanja od y, pri čemu je z promenljiva koja se ne pojavljuje kao slobodna ni u E ni u F.

U poslednjoj stavci prethodne definicije smo, u slučaju da izraz F sadrži slobodna pojavljivanja apstrahovane promenljive y, morali da najpre *preimenujemo* apstrahovanu promenljivu nekom novom promenljivom z, pa tek onda da vršimo zamenu $[x \mapsto F]$ u podizrazu E. Ovo preimenovanje apstrahovane promenljive ćemo, kao i u logici prvog reda, nazivati *α-konverzijom*.

Definicija 8.4. Neka je $\lambda x.E$ lambda izraz i neka je z promenljiva koja se ne pojavljuje kao slobodna u izrazu E. Tada za izraz $\lambda z.E[x \mapsto z]$ kažemo da je dobijen *α-konverzijom* od izraza $\lambda x.E$. Za izraze E i F kažemo da su *α-ekvivalentni* (u oznaci $E \equiv_\alpha F$) ako se mogu dobiti jedan od drugog konačnom primenom α-konverzija.

Lako se može pokazati da je relacija $\equiv_\alpha$ relacija ekvivalencije nad skupom svih lambda izraza Λ. α-ekvivalentne izraze ćemo u nastavku smatrati suštinski jednakim izrazima i to nećemo posebno naglašavati.

Primer 8.2. Izraz $x\ (\lambda x.x\ (\lambda x.x\ (\lambda y.y\ z)))$ iz primera 8.1 je α-ekvivalentan izrazu $x\ (\lambda v.v\ (\lambda u.u\ (\lambda y.y\ z)))$. Iako je dozvoljeno postojanje više apstrakcija po istoj promenljivoj u istom izrazu, obično je izraz jasniji i čitljiviji kada toga nema. Zato je u takvim situacijama poželjno izvršiti α-konverziju, kao u ovom primeru.

Primer 8.3. Neka je dat izraz $\lambda x.(\lambda y.y\ x)\ y$. Zamenom $[y \mapsto x\ x]$ u ovom izrazu dobijamo izraz $\lambda z.(\lambda y.y\ z)\ (x\ x)$. Dakle, zamenjuje se samo slobodno pojavljivanje promenljive y u izrazu. Pritom, kako izraz $x\ x$ kojim ga zamenjujemo sadrži slobodna pojavljivanja promenljive x, kako ne bi došlo do vezivanja ovih pojavljivanja apstrakcijom po x, najpre se vrši preimenovanje vezane promenljive x u z u polaznom izrazu, nakon čega se u njemu vrši zamena y sa $x\ x$. Takođe, primetimo da smo u rezultujućem izrazu (tj. u zapisu njegove konkretne sintakse) $x\ x$ stavili u zagrade – u suprotnom bi, zbog leve asocijativnosti operatora aplikacije, izraz $\lambda z.(\lambda y.y\ z)\ x\ x$ bio sintaksno identičan izrazu $\lambda z.((\lambda y.y\ z)\ x)\ x$, što nije ono što želimo.

8.2 Redukcije lambda izraza

Do ovog trenutka, lambda izraze razmatrali smo isključivo kao statičke sintaksne objekte. Kako bismo im pridružili operativni karakter, uvodimo pojam *redukcije*, koja će nam omogućiti da u terminologiji lambda izraza definišemo mehanizam formalnog izračunavanja.

8.2.1 β-redukcija

Najznačajnija vrsta redukcije lambda izraza je *β-redukcija*, koja u formalnom smislu definiše šta se dešava kada se funkcija (formirana operatorom apstrakcije) primeni (operatorom aplikacije) na neki izraz.

Definicija 8.5. Relacija *β-redukcije* (u oznaci $\Rightarrow_\beta$) je najmanja relacija za koju važi:

- $(\lambda x.E)\ F \Rightarrow_\beta E[x \mapsto F]$
- ako $E \Rightarrow_\beta E'$, tada $E\ F \Rightarrow_\beta E'\ F$
- ako $F \Rightarrow_\beta F'$, tada $E\ F \Rightarrow_\beta E\ F'$
- ako $E \Rightarrow_\beta E'$, tada $\lambda x.E \Rightarrow_\beta \lambda x.E'$

Ako za dva izraza H i H' važi $H \Rightarrow_\beta H'$, tada kažemo da se *izraz H β-redukuje u H'*.

U skladu sa ovom definicijom, da bi se izraz H mogao β-redukovati, neophodno je da sadrži podizraz oblika $(\lambda x.E)\ F$. Takav izraz nazivamo *reducibilni izraz* ili *redex*. β-redukcija se vrši tako što se u izrazu H redex $(\lambda x.E)\ F$ zameni sa $E[x \mapsto F]$. Ovo intuitivno odgovara primeni funkcije po x date izrazom $\lambda x.E$ na argument F – svuda u telu funkcije E zamenjujemo parametar x argumentom poziva F. Ukoliko izraz H u sebi nema ni jedan redex, tada ga nije moguće β-redukovati. Za takav izraz kažemo da je u *β-normalnoj formi*.

Tranzitivno i refleksivno zatvorenje relacije $\Rightarrow_\beta$ označavaćemo sa $\Rightarrow_\beta^*$. Za izraz H' kažemo da je *β-normalna forma* izraza H ako je H' u β-normalnoj formi i važi $H \Rightarrow_\beta^* H'$.

Intuitivno, β-redukciju možemo razumeti kao operativni mehanizam *izračunavanja* u kontekstu lambda izraza, dok se β-normalna forma koju dobijemo na kraju tog izračunavanja može razumeti kao rezultat tog izračunavanja. Ključno pitanje je da li je tako definisano izračunavanje jednoznačno. U tom smislu, imamo sledeću teoremu, koju navodimo bez dokaza.

Teorema 8.1 (Čerč-Rozerova teorema)**.** *Svaki lambda izraz ima najviše jednu β-normalnu formu (do na α-ekvivalenciju).*

Da bismo razjasnili značaj prethodne teoreme, primetimo najpre da lambda izraz H može u sebi sadržati više od jednog redex-a. To znači da primenom β-redukcije za različite redex-e možemo dobiti više različitih izraza H' takvih da je $H \Rightarrow_\beta H'$. Dalje, u izrazu H' takođe može biti više različitih redex-a, što nam daje više izbora za dalju redukciju, i td. Ovo znači da polazeći od istog izraza H možemo imati veliki broj različitih lanaca redukcije $H \Rightarrow_\beta H' \Rightarrow_\beta H'' \Rightarrow_\beta \ldots$, tj. različitih izračunavanja. Gornja teorema tvrdi da koji god lanac redukcija da odaberemo, β-normalne forme u koje stižemo će biti jednake. Dakle, rezultat izračunavanja je jednoznačan, bez obzira na odabrani poredak redukcija.

Primer 8.4. Razmotrimo izraz $(\lambda f.(\lambda xy.f\ x\ y)\ u\ v)\ (\lambda xy.x)$. U ovom izrazu imamo dva redex-a: $(\lambda x.\lambda y.f\ x\ y)\ u$ i ceo izraz $(\lambda f.(\lambda xy.f\ x\ y)\ u\ v)\ (\lambda xy.x)$. Prvi način je da najpre izvršimo redukciju celog izraza. Imamo:

$$(\lambda f.(\lambda xy.f\ x\ y)\ u\ v)\ (\lambda xy.x) \Rightarrow_\beta (\lambda xy.(\lambda xy.x)\ x\ y)\ u\ v$$

jer je:

$$((\lambda xy.f\ x\ y)\ u\ v)[f \mapsto (\lambda xy.x)] = (\lambda xy.(\lambda xy.x)\ x\ y)\ u\ v$$

Unutrašnji izraz $(\lambda xy.x)\ x\ y$ se može redukovati (jedino) na sledeći način:

$$((\lambda x.\lambda y.x)\ x)\ y \Rightarrow_\beta (\lambda y.x)\ y \Rightarrow_\beta x$$

jer je $(\lambda y.x[x \mapsto x])\ y = (\lambda y.x)\ y$ i $x[y \mapsto y] = x$. Sada imamo imamo da je:

$$(\lambda xy.(\lambda xy.x)\ x\ y)\ u\ v \Rightarrow_\beta^* (\lambda xy.x)\ u\ v \Rightarrow_\beta (\lambda y.u)\ v \Rightarrow_\beta u$$

s obzirom da je $((\lambda y.x)[x \mapsto u])\ v = (\lambda y.u)\ v$ i $u[y \mapsto v] = u$. Dakle, u je β-normalna forma polaznog izraza. Sa druge strane, da smo najpre redukovali redex $(\lambda x.\lambda y.f\ x\ y)\ u$ u $\lambda y.f\ u\ y$, imali bismo:

$$(\lambda f.(\lambda xy.f\ x\ y)\ u\ v)\ (\lambda xy.x) \Rightarrow_\beta (\lambda f.(\lambda y.f\ u\ y)\ v)\ (\lambda xy.x)$$

Sada ponovo imamo dva redex-a, $(\lambda y.f\ u\ y)\ v$ i ceo izraz $(\lambda f.(\lambda y.f\ u\ y)\ v)\ (\lambda xy.x)$. U prvom slučaju, redukovaćemo $(\lambda y.f\ u\ y)\ v$ u $f\ u\ v$, pa ćemo imati:

$$(\lambda f.(\lambda y.f\ u\ y)\ v)\ (\lambda xy.x) \Rightarrow_\beta (\lambda f.f\ u\ v)\ (\lambda xy.x) \Rightarrow_\beta (\lambda xy.x)uv \Rightarrow_\beta (\lambda y.u)\ v \Rightarrow_\beta u$$

pa ponovo dolazimo do β-normalne forme u. U drugom slučaju, imamo:

$$(\lambda f.(\lambda y.f\ u\ y)\ v)\ (\lambda xy.x) \Rightarrow_\beta (\lambda y.(\lambda xy.x)\ u\ y)\ v$$

Sada ponovo imamo dva redex-a, $(\lambda xy.x)\ u$ i ceo izraz $(\lambda y.(\lambda xy.x)\ u\ y)\ v$. U prvom slučaju, redukovaćemo $(\lambda xy.x)\ u$ u $\lambda y.u$, pa imamo:

$$(\lambda y.(\lambda xy.x)\ u\ y)\ v \Rightarrow_\beta (\lambda y.(\lambda y.u)\ y)\ v$$

Odavde ponovo imamo dve redukcije:

$$(\lambda y.(\lambda y.u)\ y)\ v \Rightarrow_\beta (\lambda y.u)\ v \Rightarrow_\beta u$$

kao i:

$$(\lambda y.(\lambda y.u)\ y)\ v \Rightarrow_\beta (\lambda y.u)\ v \Rightarrow_\beta u$$

U drugom slučaju imamo:

$$(\lambda y.(\lambda xy.x)\ u\ y)\ v \Rightarrow_\beta (\lambda xy.x)\ u\ v \Rightarrow_\beta (\lambda y.u)\ v \Rightarrow_\beta u$$

Dakle, svi mogući redosledi redukcija vode ka istoj β-normalnoj formu u.

Primetimo da mogu postojati lambda izrazi koji nemaju ni jednu normalnu formu. O tome govori sledeći primer.

Primer 8.5. Neka je $E = \lambda x.x\ x$. Razmotrimo izraz $\Omega = E\ E$, tj. izraz $(\lambda x.x\ x)\ (\lambda x.x\ x)$. Jedini redex u ovom izrazu je ceo izraz, pa imamo:

$$\Omega = E\ E = (\lambda x.x\ x)\ (\lambda x.x\ x) \Rightarrow_\beta (x\ x)[x \mapsto E] = E\ E = \Omega$$

Dakle, imamo:

$$\Omega \Rightarrow_\beta \Omega \Rightarrow_\beta \Omega \Rightarrow_\beta \ldots$$

Ovo je jedini mogući način redukcije i on ne vodi ka normalnoj formi, već se produžava u beskonačnost (za takav lanac redukcija kažemo da se *ne zaustavlja*). Dakle, izraz Ω nema ni jednu normalnu formu.

Zaključak iz prethodnog primera nije u suprotnosti sa teoremom 8.1, jer ona tvrdi da svaki lambda izraz ima *najviše jednu* β-normalnu formu. Dakle, lambda izraz može da nema ni jednu β-normalnu formu, ali ako je ima, ona je jedinstvena. Ovo svojstvo relacije $\Rightarrow_\beta$ naziva se *konfluentnost*.

U prethodnim primerima razmotrili smo dva izraza – jedan kod koga je svaki redosled redukcija vodio ka normalnoj formi, kao i jedan koji nije imao normalnu formu. Pored toga, mogu postojati i izrazi kod kojih neki redosledi redukcija vode ka normalnoj formi, dok postoje i neki ,,nesrećno" odabrani redosledi redukcija koji se ne zaustavljaju.

Primer 8.6. Razmotrimo izraz $(\lambda xy.x)\ E\ (E\ E)$ gde je $E = \lambda x.x\ x$, kao u primeru 8.5. Jedan mogući redosled redukcije je sledeći:

$$(\lambda xy.x)\ E\ (E\ E) \Rightarrow_\beta (\lambda y.E)\ (E\ E) \Rightarrow_\beta E$$

Dakle, β-normalna forma ovog izraza je izraz E. Sa druge strane, kako važi:

$$E\ E \Rightarrow_\beta E\ E$$

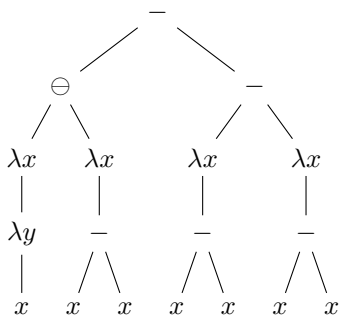
imamo i sledeći niz redukcija:

$$(\lambda xy.x)\ E\ (E\ E) \Rightarrow_\beta (\lambda xy.x)\ E\ (E\ E) \Rightarrow_\beta (\lambda xy.x)\ E\ (E\ E) \Rightarrow_\beta \ldots$$

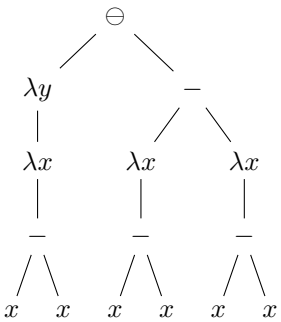
Dakle, ovaj niz redukcija se ne zaustavlja.

Iz prethodnog primera vidimo da ne moraju svi redosledi redukcije voditi ka normalnoj formi, čak i kada ona postoji. Zbog toga je od značaja odgovoriti na pitanje: da li postoji poredak redukcija koji garantovano vodi ka normalnoj formi, ako ona postoji?

Da bismo odgovorili na to pitanje, analizirajmo malo preciznije poredak redukcije koji je u prethodnom primeru doveo do normalne forme. U tom poretku, β-redukciju smo uvek primenjivali na prvi redex na koji nailazimo obilazeći stablo apstraktne sintakse izraza od korena ka listovima, sa leva u desno. U narednom stablu, taj redex je označen simbolom $\ominus$:



Primenom β-redukcije na ovaj redex dobili smo izraz čije je stablo apstraktne sintakse prikazano u nastavku:



I u ovom stablu smo redex na koji ćemo primeniti β-redukciju označili sa $\ominus$ – ponovo je u pitanju redex pronađen pretragom stabla počev od korena ka listovima. Redukcijom ovog redex-a dobija se stablo izraza E:

u kome više nema redex-a, pa se proces redukcije zaustavlja i dolazimo do β-normalne forme.

Definicija 8.6. Poredak redukcije u kome se u svakom koraku redukuje najspoljašnjiji najlevlji redex (tj. redex na koji se prvi nailazi obilaskom stabla izraza od korena ka listovima sa leva u desno) naziva se *normalni poredak redukcije*.

U prethodnom primeru, primenom normalnog poretka redukcije smo uspešno dostigli β-normalnu formu izraza. Ispostavlja se da ovo nije slučajno. Naime, sledeća teorema, koju navodimo bez dokaza, nam kaže da za proizvoljni lambda izraz normalni poredak garantuje redukciju do β-normalne forme, ako ona postoji.

Teorema 8.2. *Ako izraz H ima β-normalnu formu H', tada se do te normalne forme dolazi normalnim poretkom redukcije.*

Normalni poredak ima izuzetan teorijski značaj, zato što nam garantuje zaustavljanje redukcije, pod uslovom da normalna forma postoji. Osnovna ideja normalnog poretka je da se funkcije primenjuju na neredukovane argumente, pri čemu se redukcija argumenata odlaže dok zaista u telu funkcije ne bude neophodna. U našem primeru, funkcija $\lambda xy.x$ vraća svoj prvi argument E, dok se drugi argument $E\ E$ ne koristi, pa ga i ne redukujemo, čime izbegavamo beskonačno izračunavanje koje bi njegova redukcija proizvela.

Sa druge strane, ovaj poredak ima i jedan praktični nedostatak – redukcija po ovom poretku može biti neefikasna. Demonstrirajmo to sledećim primerom.

Primer 8.7. Neka je dat izraz $G = \lambda x.x\ x\ x$. Primenimo ovaj izraz na $H = (\lambda xy.x\ y)\ (\lambda x.x\ x)\ (\lambda x.x)$. Normalnom redukcijom izraza $G\ H$ dobijamo:

$$G\ H \Rightarrow_\beta H\ H\ H$$

Dalje, izraz H se, u normalnom poretku, može redukovati na sledeći način:

$$\begin{array}{ll} (\lambda xy.x\ y)\ (\lambda x.x\ x)\ (\lambda x.x) & \Rightarrow_\beta \\ (\lambda y.(\lambda x.x\ x)\ y)\ (\lambda x.x) & \Rightarrow_\beta \\ (\lambda x.x\ x)\ (\lambda x.x) & \Rightarrow_\beta \\ (\lambda x.x)\ (\lambda x.x) & \Rightarrow_\beta \\ (\lambda x.x) & \end{array}$$

Otuda bi se izraz $H\ H\ H$ gornjim lancom redukcija redukovao u:

$$H\ H\ H \Rightarrow_\beta^* (\lambda x.x)\ H\ H \Rightarrow_\beta H\ H$$

Dalje bismo morali da primenimo isti lanac redukciju i za drugo H:

$$H\ H \Rightarrow_\beta^* (\lambda x.x)\ H \Rightarrow_\beta H$$

Najzad, i za treće H bismo morali da imamo isti lanac redukcija:

$$H \Rightarrow_\beta^* \lambda x.x$$

U prethodnom primeru, videli smo da ako se parametar funkcije u njenom telu pojavljuje na više mesta, zamenom parametra neredukovanim argumentom će zahtevati da se u nastavku svaka kopija argumenta iznova redukuje na isti način. Ovo značajno povećava broj potrebnih koraka redukcije, tj. dužinu izračunavanja.

Često efikasnija alternativa normalnom poretku je *aplikativni poredak*.

Definicija 8.7. Poredak redukcije u kome se u svakom koraku redukuje najunutrašnjiji najlevlji redex naziva se *aplikativni poredak redukcije*.

U aplikativnom poretku se, dakle, redukcija vrši od listova ka korenu, sa leva u desno. To znači da će prilikom redukcije nekog redex-a $(\lambda x.E)\ F$ uvek važiti da su E i F već u β-normalnim formama. Dakle, i telo funkcije E i argument F su već redukovani. Samim tim, zamenom F umesto x u E neće izazvati višestruki posao, čak i ako se parametar x pojavljuje više puta u E, jer je argument F već redukovan.

Primer 8.8. U prethodnom primeru, u izrazu $G\ H$ će se najpre proveriti da li u stablu izraza G postoje redex-i (od listova ka korenu). Kako ih nema, to znači da je G već u normalnoj formi. Dalje se vrši redukcija argumenta H po aplikativnom poretku (od listova ka korenu):

$$\begin{array}{ll}(\lambda xy.x\ y)\ (\lambda x.x\ x)\ (\lambda x.x) & \Rightarrow_\beta \\ (\lambda y.(\lambda x.x\ x)\ y)\ (\lambda x.x) & \Rightarrow_\beta \\ (\lambda y.y\ y)\ (\lambda x.x) & \Rightarrow_\beta \\ (\lambda x.x)\ (\lambda x.x) & \Rightarrow_\beta \\ (\lambda x.x) & \end{array}$$

Sada se redukuje izraz $G\ (\lambda x.x)$:

$$(\lambda x.x\ x\ x)\ (\lambda x.x) \Rightarrow_\beta (\lambda x.x)\ (\lambda x.x)\ (\lambda x.x) \Rightarrow_\beta (\lambda x.x)\ (\lambda x.x) \Rightarrow_\beta (\lambda x.x)$$

Dakle, sada izbegavamo da tri puta primenjujemo dugi lanac redukcija argumenta H, jer je on najpre redukovan, pa prenet funkciji G.

Na žalost, aplikativni poredak ne garantuje da će se do normalne forme doći, čak i kada ona postoji.

Primer 8.9. Vratimo se na izraz iz primera 8.6: redukujemo izraz $(\lambda xy.x)\ E\ (E\ E)$, gde je $E = \lambda x.x\ x$. U aplikativnom poretku bismo najpre obilaskom stabla od listova ka korenu naišli na redex $(\lambda xy.x)\ E$, pa bismo imali sledeću redukciju:

$$(\lambda xy.x)\ E\ (E\ E) \Rightarrow_\beta (\lambda y.E)\ (E\ E)$$

Međutim, sada imamo dva redex-a od kojih se u aplikativnom poretku prvo redukuje onaj unutrašnji: $E\ E$. Dakle, neophodno je najpre redukovati argument pre nego što se isti preda funkciji. Međutim, kako argument $E\ E = \Omega$ nema normalnu formu, ovaj način redukcije dovodi do beskonačnog izračunavanja:

$$(\lambda y.E)\ (E\ E) \Rightarrow_\beta (\lambda y.E)\ (E\ E) \Rightarrow_\beta \ldots$$

Napomena 8.4. Koncepti normalnog i aplikativnog poretka se sreću i u realnim programskim jezicima. Pritom, većina programskih jezika koristi aplikativni poredak, zbog efikasnosti. To je slučaj sa većinom imperativnih programskih jezika, poput jezika *C*, *C++*, *Python*, i td. Svi oni najpre evaluiraju argumente dok ne dobiju njihove vrednosti koje zatim prenose funkciji. Za ovakvo izračunavanje se često kaže da je *striktno*. Naziv koji se takođe koristi za izračunavanje po aplikativnom poretku je i *pozivanje po vrednosti* (engl. *call-by-value*), jer se funkcijama uvek predaju izračunate vrednosti, a ne neizračunati izrazi. Sa druge strane, razne varijante normalnog poretka se koriste u nekim funkcionalnim

programskim jezicima poput jezika *Haskell*. Kod takvih programskih jezika izračunavanje vrednosti argumenata se odlaže dokle god to zaista nije neophodno. Ovakvo izračunavanje se obično naziva *ne-striktno* ili *lenjo izračunavanje*. Koristi se i naziv *pozivanje po imenu* (engl. *call-by-name*), jer se argumenti prenose u svom orignalnom obliku, onako kako su sintaksno zapisani. Kako bi se izbegla neefikasnost koju normalni poredak redukcije potencijalno sa sobom nosi, primenjuju se različite efikasne tehnike implementacije. Jedna od njih je da se pri prvom izračunavanju argumenta u telu funkcije njegova vrednost interno zapamti, kako bi se kasnije mogla koristiti i na drugim mestima, umesto da se ponovo izračunava. Ovakva strategija je poznata i kao *pozivanje po potrebi* (engl. *call-by-need*) i karakteristična je upravo za *Haskell*.

8.2.2 η-redukcija

Neka je E proizvoljni izraz u β-normalnoj formi koji ne sadrži slobodna pojavljivanja promenljive x i neka je $E' = \lambda x.E\ x$. Za proizvoljan izraz F imamo da je:

$$E'\ F = (\lambda x.E\ x)\ F \Rightarrow_\beta E\ F$$

Odavde sledi da izrazi $E'\ F$ i $E\ F$ ili oba nemaju β-normalnu formu ili su im β-normalne forme jednake. Drugim rečima, E i E' su dva izraza u β-normalnoj formi koji se u operativnom smislu ponašaju isto – na koji god izraz F da ih primenimo, redukcija se ili ne zaustavlja ili proizvodi iste normalne forme.

Sa druge strane mi bismo voleli da se sintaksno različiti izrazi u normalnoj formi ponašaju različito. Kako bismo to postigli, uvodimo pojam η-redukcije.

Definicija 8.8. Relacija η-redukcije (u oznaci $\Rightarrow_\eta$) je najmanja relacija nad lambda izrazima za koju važi:

- ako je E izraz koji u sebi ne sadrži slobodna pojavljivanja promenljive x, tada važi $\lambda x.E\ x \Rightarrow_\eta E$
- ako je $E \Rightarrow_\eta E'$, tada je i $E\ F \Rightarrow_\eta E'\ F$
- ako je $F \Rightarrow_\eta F'$, tada je i $E\ F \Rightarrow_\eta E\ F'$
- ako je $E \Rightarrow_\eta E'$, tada je i $\lambda x.E \Rightarrow_\eta \lambda x.E'$

Ako za dva izraza E i E' važi $E \Rightarrow_\eta E'$, tada kažemo da se E *η-redukuje* u E'.

Primer 8.10. Neka je dat izraz $\lambda x.\lambda y.x\ y$. Kako izraz x ne sadrži slobodna pojavljivanja promenljive y, imamo da je $\lambda y.x\ y \Rightarrow_\eta x$, pa je otuda i $\lambda x.\lambda y.x\ y \Rightarrow_\eta \lambda x.x$.

Tranzitivno zatvorenje relacije $\Rightarrow_\eta$ ćemo označavati sa $\Rightarrow_\eta^*$. Sa $\Rightarrow_{\beta\eta}$ ćemo označavati uniju relacija $\Rightarrow_\beta$ i $\Rightarrow_\eta$, tj. važi $H \Rightarrow_{\beta\eta} H'$ ako je ili $H \Rightarrow_\beta H'$ ili $H \Rightarrow_\eta H'$. Ako u izrazu nije moguće primeniti ni β ni η redukciju, tada kažemo da je izraz u $\beta\eta$-normalnoj formi. Za relaciju $\Rightarrow_{\beta\eta}$ takođe važi Čerč-Rozerova teorema – svaki izraz može imati najviše jednu $\beta\eta$-normalnu formu. Štaviše, kako se primenom η-redukcije ne stvaraju novi redex-i, η-redukcije se mogu ostaviti za kraj, tj. možemo najpre odrediti β-normalnu formu, a zatim na nju eventualno primeniti η-redukcije.

Sa $\Rightarrow_{\beta\eta}^{*}$ ćemo označavati refleksivno i tranzitivno zatvorenje relacije $\Rightarrow_{\beta\eta}$, dok ćemo sa $\equiv_{\beta\eta}$ označavati refleksivno, tranzitivno i simetrično zatvorenje relacije $\Rightarrow_{\beta\eta}$. Jasno je da je relacija $\equiv_{\beta\eta}$ relacija ekvivalencije nad skupom Λ. Za dva lambda izraza E_1 i E_2 za koje je $E_1 \equiv_{\beta\eta} E_2$ kažemo da su $\beta\eta$-ekvivalentni. Lako se vidi da je svaki izraz $\beta\eta$-ekvivalentan svojoj normalnoj formi (ako je ima). Takođe, može se pokazati da dva $\beta\eta$-ekvivalentna izraza ili oba nemaju $\beta\eta$-normalne forme, ili su im $\beta\eta$-normalne forme jednake.

Teorema 8.3. *Neka su E_1 i E_2 proizvoljni lambda izrazi. Tada su sledeća dva tvrđenja ekvivalentna:*

- *Izrazi E_1 i E_2 su $\beta\eta$-ekvivalentni*
- *Za svaki izraz F važi da su izrazi $E_1\ F$ i $E_2\ F$ $\beta\eta$-ekvivalentni.*

Dokaz. Ako su E_1 i E_2 $\beta\eta$-ekvivalentni, to znači da postoji konačan lanac koji se sastoji iz β i η redukcija, kao i obrnutih primena β i η redukcija (zbog simetričnosti), kojim se od E_1 dolazi do E_2. Za proizvoljan izraz F, primenom istog lanca transformacija na izraz $E_1\ F$ dolazimo do izraza $E_2\ F$, odakle sledi da su i izrazi $E_1\ F$ i $E_2\ F$ $\beta\eta$-ekvivalentni.

Obratno, neka je $F = x$, gde je x promenljiva koja se ne pojavljuje u E_1 i E_2. Sada su po pretpostavci izrazi $E_1\ x$ i $E_2\ x$ $\beta\eta$-ekvivalentni. Odatle se, slično kao i malopre, može zaključiti da su i izrazi $\lambda x.E_1\ x$ i $\lambda x.E_2\ x$ $\beta\eta$-ekvivalentni. Primenom η-redukcije na ove izraze dobijamo da su i E_1 i E_2 takođe $\beta\eta$-ekvivalentni. □

Dakle, za $\beta\eta$-redukciju zaista važi da se redukovani izrazi operativno ponašaju identično ako i samo ako su sintaksno identični (do na α-konverziju).

8.3 Lambda izrazi kao izračunljive funkcije

Svakom lambda izrazu E se može pridružiti parcijalna funkcija $\overline{E} : \Lambda \to \Lambda$ definisana sa $\overline{E}(F) = [E\ F]_{\beta\eta}$, gde je sa $[E\ F]_{\beta\eta}$ označena $\beta\eta$-normalna forma izraza $E\ F$, ako ista postoji (u suprotnom, $\overline{E}(F)$ je nedefinisano). Na osnovu razmatranja u prethodnom odeljku, dva izraza će definisati istu funkciju ako i samo ako su $\beta\eta$-ekvivalentni.

Primer 8.11. Razmotrimo izraz $I = \lambda x.x$. Za svako F važi $I\ F \Rightarrow_{\beta} F$, pa je $\overline{I}(F) = [F]_{\beta\eta}$. Nad izrazima u $\beta\eta$-normalnoj formi ova funkcija predstavlja *identičku funkciju* . Zaista, za svaki izraz F u $\beta\eta$-normalnoj formi važi $\overline{I}(F) = F$.

Primer 8.12. Razmotrimo izraz $C = \lambda x.E$, gde je E neki izraz u $\beta\eta$-normalnoj formi koji ne sadrži slobodna pojavljivanja promenljive x. Sada za svaki izraz F važi $C\ F \Rightarrow_{\beta} E$, pa je $\overline{C}(F) = E$ za svako F. Dakle, u pitanju je *konstantna funkcija.*

Primer 8.13. Neka je dat izraz $E = \lambda x.x\ x$. Za izraz $I = \lambda x.x$, važi da je $\overline{E}(I) = I$, dok $\overline{E}(E)$ nije definisano.

Napomena 8.5. Primetimo da ako je zatvoren izraz E u $\beta\eta$-normalnoj formi, tada je on oblika $\lambda x.E'$, tj. u pitanju je apstrakcija. Zaista, ako je izraz zatvoren, on ne može biti promenljiva (jer bi ona bila slobodna), a s obzirom da je redukovan, on ne može biti oblika $P\ Q$, jer bi tada morao sadržati redex (P ne može biti promenljiva, jer bi bila slobodna, pa ako nije apstrakcija, onda je aplikacija

na koju se induktivno primenjuje isto rezonovanje). Otuda je vrednost funkcije $\overline{E}(F)$ jednaka $\beta\eta$-normalnoj formi izraza $E'[x \mapsto F]$. Dakle, izračunavanje funkcije se svodi na zamenu parametra x argumentom F, a zatim redukovanjem dobijenog izraza. Ovo je u skladu sa našom intuicijom da izraz oblika $E = \lambda x.E'$ predstavlja funkciju po x, a da izraz $E\ F$ predstavlja primenu te funkcije na argument F.

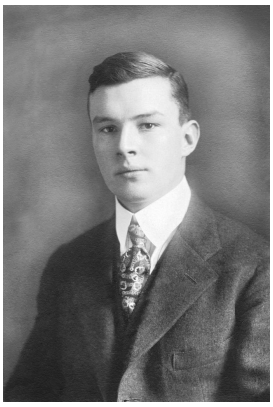
Primetimo da ovde postoji svojevrsni dualizam u tretmanu lambda izraza – lambda izrazi se koriste i kao funkcije, a i kao objekti nad kojima te funkcije operišu. Posebno, ono što funkcija vraća kao svoj rezultat je takođe lambda izraz koji se ponovo može po potrebi posmatrati kao funkcija. Ovo nam omogućava da lambda izraz primenimo na više argumenata. Na primer, ako imamo:

$$E\ F_1\ F_2$$

Zbog leve asocijativnosti operatora aplikacije, ovo je isto što i:

$$(E\ F_1)\ F_2$$

Dakle, mi najpre izraz E primenimo na argument F_1, a dobijeni izraz $E\ F_1$ posmatramo opet kao funkciju koju dalje primenjujemo na izraz F_2. Na ovaj način smo efektivno konstruisali funkciju arnosti 2. Ovakva reprezentacija funkcija veće arnosti u obliku niza funkcija arnosti 1 koje redom uzimaju jedan po jedan argument i vraćaju funkcije koje se dalje primenjuju na preostale argumente zove se *kariranje* (engl. *currying*), u čast američkog matematičara Haskela Karija (engl. *Haskell Curry*, slika 8.2). U kontekstu lambda izraza, funkcije arnosti veće od jedan se uvek predstavljaju u kariranoj formi, s obzirom da se lambda izrazi, po definiciji, uvek primenjuju na jedan argument.



Slika 8.2: Američki matematičar Haskel Kari (1900-1982)

Imajući ovo u vidu, svakom lambda izrazu E se može pridružiti i parcijalna funkcija *proizvoljne arnosti k* $\overline{E}^{(k)} : \Lambda^k \to \Lambda$ definisana na sledeći način:

$$\overline{E}^{(k)}(F_1, \ldots, F_k) = [E\ F_1\ F_2\ \ldots\ F_k]_{\beta\eta}$$

pod pretpostavkom da data $\beta\eta$-normalna forma postoji (u suprotnom, $\overline{E}^{(k)}(F_1, \ldots, F_k)$ nije definisano).

Primer 8.14. Razmotrimo izraz $K = \lambda xy.x$. Za funkciju arnosti 2 definisanu ovim izrazom važi $\overline{K}^{(2)}(F_1, F_2) = [F_1]_{\beta\eta}$. Dakle, u pitanju je funkcija *prve projekcije*.

Primer 8.15. Neka je dat izraz $S = \lambda xyz.(x\ z)\ (y\ z)$. Sada je $\overline{S}^{(3)}(F_1, F_2, F_3) = [(F_1\ F_3)\ (F_2\ F_3)]_{\beta\eta}$.

Napomena 8.6. Izraz $\lambda xy.E$ intuitivno možemo razumeti kao funkciju arnosti 2, po x i y. Činjenica da je to i sintaksno isto što i $\lambda x.\lambda y.E$ nam dodatno rasvetljava pojam kariranja u kontekstu lambda izraza – svaka funkcija veće arnosti se uvek predstavlja u kariranom obliku, nadovezivanjem funkcija arnosti 1.

Definicija 8.9. Parcijalna funkcija $f : \Lambda^k \to \Lambda$ arnosti k je λ-*izračunljiva* ako postoji lambda izraza $E \in \Lambda$ takav da je $\overline{E}^{(k)} = f$. Kažemo da E *izračunava* funkciju f ili da je E λ-*program* funkcije f.

Napomena 8.7. Iz teoreme 8.3 sledi da svaka dva $\beta\eta$-ekvivalentna izraza izračunavaju istu funkciju. Otuda za istu λ-izračunljivu funkciju možemo imati više različitih programa koji je izračunavaju.

S obzirom da je funkcija $\overline{E}^{(k)}(F_1, \ldots, F_k)$ zadata kariranjem, njeno izračunavanje se prirodno svodi na parcijalnu primenu: primenimo E na izraz F_1, pa tako dobijeni izraz primenjujemo na preostale izraze $F_2, \ldots, F_k$. Drugim rečima, važi:

$$\overline{E}^{(k)}(F_1, \ldots, F_k) = \overline{E\ F_1}^{(k-1)}(F_2, \ldots, F_k)$$

Dakle, parcijalnom primenom E na F_1 smo funkciju $\overline{E}^{(k)}$ arnosti k *transformisali* u funkciju $\overline{E\ F_1}^{(k-1)}$ arnosti $k-1$. Pritom, ova transformacija je *efektivna* – ako je izraz E program koji izračunava funkciju $\overline{E}^{(k)}$, tada izraz $E\ F_1$ predstavlja program koji izračunava funkciju $\overline{E\ F_1}^{(k-1)}$. Dakle, program za izračunavanje funkcije $\overline{E\ F_1}^{(k-1)}$ možemo efektivno dobiti na osnovu programa za izračunavanje funkcije $\overline{E}^{(k)}$ i njenog prvog argumenta F_1. Ova činjenica predstavlja manifestaciju *s-m-n* teoreme u kontekstu lambda izraza.

8.3.1 Logičke funkcije u lambda računu

U prethodnom izlaganju smo videli da se lambda izrazi mogu razumeti kao opisi *algoritama* za izračunavanje funkcija. Takođe, lambda izrazi se koriste i kao *podaci* nad kojima te funkcije operišu. Dakle, i algoritme i podatke predstavljamo istim formalizmom, što je jedna od najznačajnijih karakteristika lambda računa. U ovom i narednim odeljcima razmatramo predstavljanje različitih tipova podataka kao i funkcija koje operišu nad tim podacima u okviru lambda računa.

Počećemo sa osnovnim logičkim operacijama. Da bismo definisali logičke operacije u lambda računu, najpre ćemo pomoću lambda izraza predstaviti vrednosti logičkog tipa – *tačno* i *netačno*:

$$\begin{aligned} true &= \lambda x.\lambda y.x \\ false &= \lambda x.\lambda y.y \end{aligned}$$

Dakle, $true$ se definiše kao funkcija dva argumenta koja vraća prvi argument, dok se $false$ definiše kao funkcija dva argumenta koja vraća drugi argument. Sada možemo definisati *uslovni operator*:

$$cond = \lambda b.\lambda x.\lambda y.b\ x\ y$$

Za bilo koja dva izraza E_1 i E_2 imamo da je:

$$cond\ true\ E_1\ E_2 \Rightarrow^*_\beta true\ E_1\ E_2 = (\lambda x.\lambda y.x)\ E_1\ E_2 \Rightarrow^*_\beta E_1$$

kao i:

$$cond\ false\ E_1\ E_2 \Rightarrow^*_\beta false\ E_1\ E_2 = (\lambda x.\lambda y.y)\ E_1\ E_2 \Rightarrow^*_\beta E_2$$

Pomoću uslovnog operatora možemo definisati logičke funkcije:

$$\begin{array}{lcl} not & = & \lambda b.cond\ b\ false\ true \\ and & = & \lambda b_1 b_2.cond\ b_1\ b_2\ false \\ or & = & \lambda b_1 b_2.cond\ b_1\ true\ b_2 \\ xor & = & \lambda b_1 b_2.cond\ b_1\ (not\ b_2)\ b_2 \\ imp & = & \lambda b_1 b_2.or\ (not\ b_1)\ b_2 \end{array}$$

Primer 8.16. Neka je data iskazna formula $(p \vee q) \Rightarrow r$. Funkcija:

$$val = \lambda pqr.imp\ (or\ p\ q)\ r$$

za date vrednosti promenljivih p, q, r određuje vrednost formule u toj valuaciji. Na primer:

$$val\ true\ false\ false \Rightarrow^*_\beta false$$

8.3.2 Uređeni parovi, torke i liste u lambda računu

Uređeni par izraza (E_1, E_2) ćemo predstavljati izrazom $\lambda f.f\ E_1\ E_2$. Na ovaj način unutar lambda izraza čuvamo komponente para, a imamo mogućnost da operišemo sa komponentama tog para tako što mu predamo odgovarajuću funkciju kao argument. Da bismo kreirali par, možemo koristiti funkciju:

$$mk\_pair = \lambda x.\lambda y.\lambda f.f\ x\ y$$

Zaista, za svaka dva izraza E_1 i E_2 važi da je:

$$mk\_pair\ E_1\ E_2 = (\lambda x.\lambda y.\lambda f.f\ x\ y)\ E_1\ E_2 \Rightarrow^*_\beta \lambda f.f\ E_1\ E_2$$

Ako želimo da iz nekog para izdvojimo njegovu prvu ili drugu komponentu, to možemo uraditi pomoću funkcija:

$$\begin{array}{lcl} first & = & \lambda p.p\ true \\ second & = & \lambda p.p\ false \end{array}$$

Ove funkcije paru p koji im se da kao argument predaju funkcije $true$ i $false$ koje su, kao što znamo, definisane kao funkcije prve, odnosno druge, projekcije. Time se izdvaja odgovarajuća komponenta para. Na primer, za par (E_1, E_2) predstavljen izrazom $\lambda f.f\ E_1\ E_2$ imamo:

$$first\ (\lambda f.f\ E_1\ E_2) = (\lambda p.p\ true)\ (\lambda f.f\ E_1\ E_2) \Rightarrow_\beta$$
$$(\lambda f.f\ E_1\ E_2)\ true \Rightarrow_\beta true\ E_1\ E_2 = (\lambda xy.x)\ E_1\ E_2 \Rightarrow_\beta^* E_1$$

kao i:

$$second\ (\lambda f.f\ E_1\ E_2) = (\lambda p.p\ false)\ (\lambda f.f\ E_1\ E_2) \Rightarrow_\beta$$
$$(\lambda f.f\ E_1\ E_2)\ false \Rightarrow_\beta false\ E_1\ E_2 = (\lambda xy.y)\ E_1\ E_2 \Rightarrow_\beta^* E_2$$

Na sličan način je moguće predstaviti i uređene n-torke: n-torka $(E_1, \ldots, E_n)$ će biti predstavljena izrazom $\lambda f.f\ E_1\ \ldots\ E_n$, a funkcije za pristup pojedinačnim elementima torke se mogu realizovati na sličan način kao malopre, što ostavljamo čitaocu za vežbu.

U nastavku ćemo, zbog čitljivosti, parove zapisivati u formi (E_1, E_2), a podrazumevaćemo da to znači $\lambda f.f\ E_1\ E_2$. Slično važi i za proizvoljne torke.

Primer 8.17. Funkcija koja zamenjuje redosled komponenti uređenog para (tj. par (E_1, E_2) transformiše u par (E_2, E_1)) možemo opisati ovako:

$$xchg = \lambda p.mk\_pair\ (second\ p)\ (first\ p)$$

Na primer:

$$xchg\ (true, false) \Rightarrow_\beta mk\_pair\ (second\ (true, false))\ (first\ (true, false)) \Rightarrow_\beta^*$$
$$mk\_pair\ false\ true \Rightarrow_\beta^* (false, true)$$

Listu izraza proizvoljne konačne dužine ćemo predstavljati uređenim parom (H, T), gde je H početni element liste (*glava liste*), a T je lista koja sadrži ostale elemente liste (*rep liste*). Specijalno, *prazna lista* (koju označavamo sa *nil*) će biti predstavljena izrazom $\lambda xy.y$ (dakle, istim izrazom kao i $false$). Na primer, lista $[E_1, E_2, \ldots, E_n]$ će biti predstavljena izrazom:

$$(E_1, (E_2, \ldots (E_n, nil)) \ldots)$$

Da bismo mogli da baratamo sa listama, najpre su nam potrebne osnovne funkcije za konstrukciju liste, kao i za izdvajanje glave i repa:

$$\begin{array}{lcl} cons & = & mk\_pair \\ head & = & first \\ tail & = & second \end{array}$$

Dakle, *cons* konstruiše listu od glave i repa kreirajući odgovarajući par, dok *head* i *tail* izdvajaju prvu i drugu komponentu tog para – glavu i rep liste. Takođe nam je potrebna funkcija koja proverava da li je lista prazna:

$$is\_nil = \lambda l.l\ (\lambda xyz.false)\ true$$

S obzirom da je neprazna lista predstavljena uređenim parom, mi ovde tom uređenom paru predajemo funkciju $\lambda xyz.false$. Kada se ova funkcija primeni na glavu i rep liste, dobija se funkcija $\lambda z.false$ koja uvek vraća $false$. Njenom primenom na $true$ dobijamo $false$:

$$is\_nil\ (H, T) = (\lambda l.l\ (\lambda xyz.false)\ true)\ (\lambda f.f\ H\ T) \Rightarrow_\beta^*$$
$$(\lambda f.f\ H\ T)\ (\lambda xyz.false)\ true \Rightarrow_\beta^*$$
$$\Rightarrow_\beta^* (\lambda xyz.false)\ H\ T\ true \Rightarrow_\beta^* false$$

Sa druge strane, za praznu listu imamo:

$$is\_nil\ nil = (\lambda l.l\ (\lambda xyz.false)\ true)\ (\lambda xy.y) \Rightarrow^*_\beta$$
$$(\lambda xy.y)\ (\lambda xyz.false)\ true \Rightarrow^*_\beta true$$

jer se prazna lista ponaša kao funkcija koja prosto vraća svoj drugi argument.

Primer 8.18. Konstruišimo funkciju koja vraća listu koja nastaje zamenom redosleda prva dva elementa, ukoliko lista ima bar dva elementa, a $false$ u suprotnom.

$$\begin{array}{rcll} xchg12 & = & \lambda l.cond & (or\ (is\_nil\ l)\ (is\_nil\ (tail\ l))) \\ & & & false \\ & & & (cons\ (head\ (tail\ l))\ (cons\ (head\ l)\ (tail\ (tail\ l)))) \end{array}$$

Dakle, funkcijom *cond* najpre ispitujemo da li je lista prazna ili je njen rep prazna lista. Ako jeste, vraćamo $false$, dok u suprotnom konstruišemo listu tako što ispred *tail* (*tail* l) najpre dopišemo *head* l, a zatim i *head* (*tail* l).

Liste ćemo na dalje zapisivati u formatu $[E_1, \ldots, E_n]$, a podrazumevaćemo da to označava $(E_1, (E_2, \ldots (E_n, nil)) \ldots)$.

8.3.3 Rekurzija u lambda računu

Na prvi pogled, λ-izračunljivost je prilično ograničena i svodi se na supstitucije promenljivih drugim izrazima u telu funkcije. Ne postoji ništa poput petlji, što bi λ-izračunljivost moglo staviti u istu ravan sa npr. URM programima ili Tjuringovim mašinama. Ovo ograničenje bi se, teorijski, moglo prevazići ako bismo u lambda izrazima imali mogućnost rekurzije. Na primer, bilo bi dobro kada bismo mogli da neki izraz f definišemo na sledeći način:

$$f = \lambda x.E[x, f]$$

gde je sa $E[x, f]$ označen neki lambda izraz koji u sebi sadrži slobodna pojavljivanja promenljivih x i f. Dakle, u pitanju je jedna *jednačina* po f čije bi rešenje upravo definisalo funkciju koja poziva samu sebe u svom telu. Na žalost, ovakva jednačina nema rešenja, jer bi takav izraz sadržao samog sebe kao podizraz, pa bi samim tim bio beskonačan, što nije moguće po definiciji lambda izraza kao konačnih objekata. Ipak, ispostavlja se da je ovako nešto moguće efektivno simulirati. Naime, iako gornja jednačina nema rešenja u smislu sintaksne jednakosti, ispostavi se da postoji rešenje jednačine:

$$f \equiv_{\beta\eta} \lambda x.E[x, f]$$

Dobijeno rešenje F je, prema teoremi 8.3, funkcionalno ekvivalentno sa $\lambda x.E[x, F]$, čime se efektivno simulira rekurzija.

Ono što je važno napomenuti je da se traženo rešenje F može efektivno konstruisati. Krenimo od jednačine:

$$f \equiv_{\beta\eta} \lambda x.E[x, f]$$

Ako promenljivu f u izrazu na desnoj strani apstrahujemo, dobijamo izraz:

$$F_f = \lambda f.\lambda x.E[x, f]$$

koji se, intuitivno, ponaša kao funkcija koja prihvata neku funkciju f i vraća drugu funkciju po x koja u svom telu koristi funkciju f. Dakle, F_f *transformiše* jednu funkciju u drugu. U tom tumačenju, rešenje gornje jednačine je zapravo *fiksna tačka* transformacije F_f – tražimo onu funkciju f takvu da je izraz $F_f\ f$ funkcionalno ekvivalentan sa f. Da bismo pronašli ovu fiksnu tačku, potrebno je da obezbedimo rekurzivno ponašanje funkcije f koju predajemo transformaciji F_f kao argument. Ovo postižemo tako što primenimo F_f na $(g\ g)$, pa opet apstrahujemo po g:

$$F_g = \lambda g.F_f\ (g\ g)$$

Na ovaj način obezbeđujemo da se funkcija koju predamo kao argument funkciji F_f unutar tela funkcije F_f primenjuje sama na sebe:

$$F_g = \lambda g.F_f\ (g\ g) = \lambda g.(\lambda f.\lambda x.E[x,f])\ (g\ g) \Rightarrow_\beta \lambda g.\lambda x.E[x,(g\ g)]$$

Sada se fiksna tačka transformacije F_f dobija kada se F_g primeni sama na sebe:

$$F = F_g\ F_g$$

Zaista, ovim se pokreće rekurzija, pa imamo:

$$F = F_g\ F_g = (\lambda g.F_f\ (g\ g))\ F_g \Rightarrow_\beta F_f\ (F_g\ F_g) = F_f\ F$$

odnosno:

$$F_f\ F \equiv_{\beta\eta} F$$

tj. važi:

$$F \equiv_{\beta\eta} \lambda x.E[x,F]$$

kao što smo i želeli. Uopšte, ako transformaciju F_f zamenimo proizvoljnom transformacijom h, pa apstrahujemo po h, dobijamo sledeći kombinator:

$$Y = \lambda h.(\lambda g.h\ (g\ g))\ (\lambda g.h\ (g\ g))$$

Ovaj kombinator je poznat i kao *Karijev kombinator fiksne tačke*. Njegovom primenom na proizvoljnu transformaciju dobijamo njenu fiksnu tačku. U našem primeru smo fiksnu tačku transformacije F_f dobili kao:

$$F = Y\ F_f$$

Napomena 8.8. Setimo se teoreme 3.9 o fiksnoj tački iz glave 3. U terminologiji URM programa, ona je tvrdila da svaka izračunljiva funkcija f koja transformiše kôd programa n u kôd nekog drugog programa $f(n)$ ima fiksnu tačku:

$$\phi_n = \phi_{f(n)}$$

U kontekstu lambda izraza, ulogu funkcije f igra transformacija F_f, a njena fiksna tačka predstavlja funkciju koja odgovara datoj rekurzivnoj definiciji.

U praksi, da bismo realizovali rekurziju u kontekstu lambda izraza, primenjujemo sledeći postupak:

- polazimo od željene rekurzivne definicije funkcije $f = \lambda x.E[x,f]$

- u telu rekurzivne funkcije zamenimo sve rekurzivne pozive pozivom neke druge funkcije koja se predaje kao prvi argument ($\lambda f.\lambda x.E[x, f]$), čime telo rekurzivne funkcije pretvaramo u *transformaciju* jedne funkcije u drugu
- za tako dobijenu transformaciju odredimo fiksnu tačku (Y kombinatorom)

Primer 8.19. Konstruišimo funkciju koja dodaje element na kraj liste. Ovu funkciju možemo opisati rekurzivno:

$$\begin{array}{lcll} append & = & \lambda x \lambda l.cond & (is\_nil\ l) \\ & & & (cons\ x\ nil) \\ & & & (cons\ (head\ l)\ (append\ x\ (tail\ l))) \end{array}$$

Da bismo rešili ovu jednačinu po *append* (u smislu $\beta\eta$-ekvivalencije), definišemo odgovarajuću transformaciju:

$$\begin{array}{lcll} append\_f & = & \lambda f.\lambda x.\lambda l.cond & (is\_nil\ l) \\ & & & (cons\ x\ nil) \\ & & & (cons\ (head\ l)\ (f\ x\ (tail\ l))) \end{array}$$

Dakle, samo smo dodali kao prvi parametar funkciju f koju sada pozivamo u telu funkcije umesto rekurzivnog poziva. Funkcija *append* će biti fiksna tačka transformacije $append\_f$, koju dobijamo primenom Y kombinatora na $append\_f$:

$$append = Y\ append\_f = append\_g\ append\_g$$

gde je:

$$append\_g = \lambda g.append\_f\ (g\ g)$$

Na primer, ako želimo da na kraj liste $[true, false]$ dopišemo $true$, imali bismo:

$$\begin{array}{l} append\ true\ [true, false] = Y\ append\_f\ true\ [true, false] \Rightarrow_\beta^* \\ append\_g\ append\_g\ true\ [true, false] \Rightarrow_\beta^* \\ (\lambda g.append\_f\ (g\ g))\ append\_g\ true\ [true, false] \Rightarrow_\beta^* \\ append\_f\ (append\_g\ append\_g)\ true\ [true, false] \end{array}$$

Kako lista nije prazna, operator *cond* u telu funkcije $append\_f$ se dalje redukuje u svoj treći argument, pa dobijamo:

$$\begin{array}{l} append\_f\ (append\_g\ append\_g)\ true\ [true, false] \Rightarrow_\beta^* \\ cons\ (head\ [true, false])\ ((append\_g\ append\_g)\ true\ (tail\ [true, false])) \Rightarrow_\beta^* \\ cons\ true\ (append\_g\ append\_g\ true\ [false]) \Rightarrow_\beta^* \\ cons\ true\ (append\_f\ (append\_g\ append\_g)\ true\ [false]) \end{array}$$

Opet, kako lista $[false]$ nije prazna, ponovo imamo sličnu redukciju:

$$\begin{array}{l} cons\ true\ (append\_f\ (append\_g\ append\_g)\ true\ [false]) \Rightarrow_\beta^* \\ cons\ true\ (cons\ (head\ [false])\ ((append\_g\ append\_g)\ true\ (tail\ [false]))) \Rightarrow_\beta^* \\ cons\ true\ (cons\ false\ (append\_g\ append\_g\ true\ nil)) \Rightarrow_\beta^* \\ cons\ true\ (cons\ false\ (append\_f\ (append\_g\ append\_g)\ true\ nil)) \end{array}$$

Ovog puta se *append_f* primenjuje na praznu listu, pa se *cond* u njenom telu redukuje u svoj drugi argument:

$$cons\ true\ (cons\ false\ (append\_f\ (append\_g\ append\_g)\ true\ nil)) \Rightarrow^*_\beta \\ cons\ true\ (cons\ false\ (cons\ true\ nil))$$

Ovde se izlazi iz rekurzije, jer smo izabrali granu u kojoj nema rekurzivnog poziva. Rezultat je upravo lista $[true, false, true]$ koju je i trebalo dobiti.

U nastavku ovog teksta ćemo podrazumevati da se funkcije mogu zadati rekurzivno, kao i da se odgovarajući izraz koji tu rekurziju realizuje može dobiti Y kombinatorom na gore opisani način, tako da to nećemo više posebno naglašavati.

Primer 8.20. Funkcija *reverse* koja obrće listu se može definisati rekurzivno na sledeći način:

$$\begin{array}{lcll} reverse & = & \lambda l.cond & (is\_nil\ l) \\ & & & nil \\ & & & (append\ (head\ l)\ (reverse\ (tail\ l))) \end{array}$$

8.3.4 Aritmetika u lambda računu

Da bismo predstavili aritmetiku u okviru lambda računa, najpre ćemo prirodne brojeve predstaviti *Čerčovim numeralima*:

$$\begin{array}{lcl} \overline{0} & = & \lambda f.\lambda x.x \\ \overline{1} & = & \lambda f.\lambda x.f\ x \\ \overline{2} & = & \lambda f.\lambda x.f\ (f\ x) \\ \ldots & & \ldots \\ \overline{n} & = & \lambda f.\lambda x.\underbrace{f\ (f\ \ldots\ (f}_{n}\ x)\ldots) \\ \ldots & & \ldots \end{array}$$

pri čemu je sa $\overline{n}$ predstavljan numeral kojim je predstavljen prirodan broj n. Inutuitivno, numeral $\overline{n}$ je funkcija koja prihvata neku funkciju f kao argument, a zatim tu funkciju n puta primenjuje na svoj drugi argument x (ovo ćemo kraće zapisivati sa $f^n\ x$). Skup svih Čerčovih numerala ćemo označavati sa $\mathbb{N}_C$. Lako možemo proveriti da li je dati numeral nula:

$$is\_zero = \lambda n.n\ (\lambda x.false)\ true$$

Zaista, imamo:

$$is\_zero\ \overline{0} = (\lambda n.n\ (\lambda x.false)\ true)\ (\lambda f.\lambda x.x) \Rightarrow_\beta \\ (\lambda f.\lambda x.x)\ (\lambda x.false)\ true \Rightarrow^*_\beta true$$

dok za $n > 0$ imamo:

$$is\_zero\ \overline{n} = (\lambda n.n\ (\lambda x.false)\ true)\ (\lambda f.\lambda x.f^n\ x) \Rightarrow_\beta \\ (\lambda f.\lambda x.f^n\ x)\ (\lambda x.false)\ true \Rightarrow_\beta \\ (\lambda x.(\lambda x.false)^n\ x)\ true \Rightarrow_\beta \\ (\lambda x.false)^n\ true \Rightarrow^*_\beta false$$

Funkcija *sledbenik* se može jednostavno definisati na sledeći način:

$$succ = \lambda n.\lambda f.\lambda x.f\ (n\ f\ x)$$

Zaista, za svaki numeral $\overline{n}$ imamo da je:

$$succ\ \overline{n} \Rightarrow_\beta \lambda f.\lambda x.f\ (\overline{n}\ f\ x) \Rightarrow_\beta \lambda f.\lambda x.f\ (f^n\ x) = \lambda f.\lambda x.f^{n+1}\ x = \overline{n+1}$$

Sa druge strane, predstavljanje funkcije koja izračunava *prethodnika* datog numerala zahteva malo više truda. Ideja je da najpre napravimo funkciju koja par numerala $(\overline{m}, \overline{n})$ transformiše u par $(\overline{n}, \overline{n+1})$:

$$C = \lambda p.mk\_pair\ (second\ p)\ (succ\ (second\ p))$$

Uzastopnom primenom ove funkcije na par $(\overline{0}, \overline{0})$ dobijamo parove:

$$\begin{array}{lcl} C\ (\overline{0},\overline{0}) & = & (\overline{0},\overline{1}) \\ C\ (\overline{0},\overline{1}) & = & (\overline{1},\overline{2}) \\ C\ (\overline{1},\overline{2}) & = & (\overline{2},\overline{3}) \\ \ldots & & \ldots \\ C\ (\overline{n-1},\overline{n}) & = & (\overline{n},\overline{n+1}) \end{array}$$

Dakle, kada funkciju C primenimo n puta na par $(\overline{0}, \overline{0})$, dobijamo par $(\overline{n-1}, \overline{n})$:

$$C^n\ (\overline{0},\overline{0}) = (\overline{n-1}, \overline{n})$$

Iz ovog para možemo izdvojiti prvu komponentu – to je upravo prethodnik numerala $\overline{n}$. Dakle, imamo:

$$prev = \lambda n.first\ (n\ C\ (\overline{0},\overline{0}))$$

Kako je numeral $\overline{n}$ predstavljen funkcijom koja datu funkciju primenjuje n puta na dati argument, imamo da je $\overline{n}\ C\ (\overline{0},\overline{0}) \Rightarrow_\beta^*\ C^n\ (\overline{0},\overline{0})$. Zato za numeral $\overline{n}$ imamo:

$$prev\ \overline{n} = first\ (\overline{n}\ C\ (\overline{0},\overline{0})) \Rightarrow_\beta^*\ first\ (C^n\ (\overline{0},\overline{0})) \Rightarrow_\beta^* \\ first\ (\overline{n-1}, \overline{n}) \Rightarrow_\beta^*\ \overline{n-1}$$

Takođe, važi da je $prev\ \overline{0} \Rightarrow_\beta^*\ \overline{0}$, jer je $\overline{0}\ C\ (\overline{0},\overline{0}) = (\overline{0},\overline{0})$. Ovaj način definisanja funkcije *prev* u okviru lambda računa pripisuje se Kliniju.

Ispostavlja se da je, koristeći ove osnovne funkcije is_zero, *succ* i *prev* i rekurziju, u terminima Čerčovih numerala moguće predstaviti bilo koju μ-izračunljivu funkciju. Formalno, neka je $f : \mathbb{N}^k \to \mathbb{N}$ proizvoljna parcijalna funkcija arnosti k. Neka je $\alpha : \mathbb{N} \to \mathbb{N}_C$ funkcija koja svakom prirodnom broju pridružuje odgovarajući Čerčov numeral. Funkciji f će u lambda računu odgovarati funkcija $f_\lambda = \alpha \circ f \circ \alpha^{-1} : \mathbb{N}_C^k \to \mathbb{N}_C$ za koju važi da je $f_\lambda(\overline{n_1}, \ldots, \overline{n_k}) = \overline{m}$ ako i samo ako je $f(n_1, \ldots, n_k) = m$. Ako je f μ-izračunljiva, tada je funkcija f_λ λ-izračunljiva, o čemu govori sledeća teorema.

Teorema 8.4. *Neka je $f : \mathbb{N}^k \to \mathbb{N}$ bilo koja parcijalna μ-izračunljiva funkcija arnosti k. Ako prirodne brojeve predstavimo Čerčovim numeralima, tada je odgovarajuća parcijalna funkcija $f_\lambda : \mathbb{N}_C^k \to \mathbb{N}_C$ λ-izračunljiva.*

Dokaz. Da bismo dokazali da je f_λ λ-izračunljiva, potrebno je pokazati da se f_λ može predstaviti lambda izrazom. Osnovne μ-rekurzivne funkcije (tačnije, njihovi ekvivalenti nad Čerčovim numeralima) se mogu u lambda računu predstaviti na sledeći način:

- za funkciju Z_k, odgovarajuća funkcija $Z_{k\lambda}$ se predstavlja izrazom $\lambda x_1 \dots x_k.\overline{0}$
- za funkciju S, odgovarajuća funkcija S_λ se predstavlja izrazom *succ*
- za funkciju P_k^i, odgovarajuća funkcija $P_{k\lambda}^i$ se predstavlja izrazom $\lambda x_1 \dots x_i \dots x_k.x_i$

Ako je h parcijalna funkcija arnosti m takva da je h_λ predstavljena lambda izrazom H, a $g_1, \dots, g_m$ su parcijalne funkcije arnosti k takve da su funkcije $g_{1\lambda}, \dots, g_{m\lambda}$ predstavljene redom izrazima $G_1, \dots, G_k$, tada se za funkciju $f = Sub(h, g_1, \dots, g_m)$ odgovarajuća funkcija f_λ može predstaviti izrazom:

$$\lambda x_1 \dots x_k.H\ (G_1\ x_1\ \dots\ x_k)\ \dots\ (G_m\ x_1\ \dots\ x_k)$$

Dalje, ako je za parcijalnu funkciju g arnosti k odgovarajuća funkcija g_λ predstavljena lambda izrazom G, a za funkciju h arnosti $k+2$ je odgovarajuća funkcija h_λ predstavljana izrazom H, tada se za funkciju $f = Rec(g, h)$ arnosti $k+1$ odgovarajuća funkcija f_λ može predstaviti sledećom rekurzivnom definicijom:

$$\begin{array}{rcll} F & = & \lambda y x_1 \dots x_k.cond & (is\_zero\ y) \\ & & & (G\ x_1\ \dots\ x_k) \\ & & & (H\ y\ x_1\ \dots\ x_k\ (F\ y\ x_1\ \dots\ x_k)) \end{array}$$

Najzad, ako je za parcijalnu funkciju g arnosti $k+1$ njena odgovarajuća funkcija g_λ predstavljena lambda izrazom G, tada se za funkciju $f = \mu(g)$ arnosti k odgovarajuća funkcija f_λ može predstaviti tako što najpre zadamo sledeću rekurzivnu definiciju:

$$\begin{array}{rcll} F' & = & \lambda z x_1 \dots x_k.cond & (is\_zero\ (G\ z\ x_1\ \dots\ x_k)) \\ & & & z \\ & & & (F'\ (succ\ z)\ x_1\ \dots\ x_k) \end{array}$$

Ovim smo definisali funkciju koja vraća najmanje $y \geq z$ takvo da je $g(y, x_1, \dots, x_k) = 0$, ako takvo y postoji. Sada se vrednost funkcije f dobija izračunavanjem ove funkcije za $z = 0$, tj. funkcija f_λ je predstavljena lambda izrazom:

$$F = F'\ \overline{0}$$

Kako se svaka μ-rekurzivna funkcija može dobiti od osnovnih μ-rekurzivnih funkcija konačnom primenom operatora *Sub*, *Rec* i μ, sledi da se za svaku μ-rekurzivnu funkciju f, njoj odgovarajuća funkcija f_λ može predstaviti lambda izrazom, pa je λ-izračunljiva. □

Iz prethodne teoreme sledi da se u okviru lambda računa može simulirati izračunavanje bilo koje μ-rekurzivne funkcije, pa samim tim i bilo kog URM programa, s obzirom da su URM programi i μ-rekurzivne funkcije ekvivalentni formalizmi. S obzirom na teoremu 7.1, sledi da se i svaka Tjuringova mašina može simulirati u okviru lambda računa. Otuda je lambda račun *Tjuring kompletan*.

Primer 8.21. Kao u dokazu prethodne teoreme, možemo definisati funkciju sabiranja pomoću primitivne rekurzije:

$$\begin{array}{rcll} add & = & \lambda xy.cond & (is\_zero\ y) \\ & & & x \\ & & & (succ\ (add\ x\ (prev\ y))) \end{array}$$

Slično, funkcija množenja se može definisati ovako:

$$\begin{array}{rcll} mul & = & \lambda xy.cond & (is\_zero\ y) \\ & & & \overline{0} \\ & & & (add\ (mul\ x\ (prev\ y))\ x) \end{array}$$

Čitaocu ostavljamo za vežbu da na sličan način definiše i ostale osnovne aritmetičke i relacione operatore.

Primer 8.22. Primetimo da se funkcije *add* i *mul* mogu definisati i jednostavnije, bez korišćenja rekurzije. Na primer:

$$add = \lambda m.\lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x)$$

Zaista, ako pozovemo $add\ \overline{m}\ \overline{n}$, dobićemo izraz $\lambda f.\lambda x.\overline{m}\ f\ (\overline{n}\ f\ x)$. Izraz $\overline{n}\ f\ x$ se redukuje u $f^n\ x$, pa se izraz $\overline{m}\ f\ (f^n\ x)$ redukuje u $f^m\ (f^n\ x)$, odnosno u $f^{m+n}\ x$. Otuda se ceo izraz $\lambda f.\lambda x.\overline{m}\ f\ (\overline{n}\ f\ x)$ redukuje u $\lambda f.\lambda x.f^{m+n}\ x$, što je upravo numeral $\overline{m+n}$. Na sličan način, važi da je:

$$mul = \lambda m.\lambda n.\lambda f.m\ (n\ f)$$

Kada pozovemo $mul\ \overline{m}\ \overline{n}$, dobićemo izraz $\lambda f.\overline{m}\ (\overline{n}\ f)$. Izraz $\overline{n}\ f$ se redukuje u $\lambda x.f^n\ x$, pa se izraz $\overline{m}\ (\lambda x.f^n\ x)$ redukuje u $\lambda y.(\lambda x.f^n\ x)^m\ y$, tj. potrebno je m puta primeniti funkciju $\lambda x.f^n\ x$ na y. Imamo $(\lambda x.f^n\ x)^m\ y \Rightarrow_\beta (\lambda x.f^n\ x)^{m-1}\ (f^n\ y) \Rightarrow_\beta (\lambda x.f^n\ x)^{m-2}\ (f^{2n}\ y) \Rightarrow_\beta \ldots \Rightarrow_\beta f^{m\cdot n}\ y$. Dakle, izraz $\lambda y.(\lambda x.f^n\ x)^m\ y$ se redukuje u $\lambda y.f^{m\cdot n}\ y$, pa se izraz $\lambda f.\overline{m}\ (\overline{n}\ f)$ redukuje u $\lambda f.\lambda y.f^{m\cdot n}\ y$, što je upravo numeral $\overline{m\cdot n}$.

Još zanimljiviji je primer eksponencijalne funkcije m^n, koja se može reprezentovati veoma jednostavno, bez upotrebe rekurzije:

$$exp = \lambda m.\lambda n.n\ m$$

Zaista, $exp\ \overline{m}\ \overline{n}$ se redukuje u $\overline{n}\ \overline{m}$. Kako je $\overline{n} = \lambda f.\lambda x.f^n\ x$, za svako g izraz $\overline{n}\ \overline{m}\ g$ se redukuje u $\overline{m^n}\ g = (\underbrace{\overline{m}\ (\overline{m}\ (\ldots\ (\overline{m}}_{n}\ g))\ldots)$. Kako se izraz $\overline{m}\ g\ x$ za proizvoljno x redukuje u $g^m\ x$, sledi da se primenom operatora $\overline{m}$ na g dobija funkcija g^m. Daljom primenom $\overline{m}\ g^m$ dobija se funkcija $(g^m)^m = g^{m^2}$. Slično, primenom $\overline{m}\ g^{m^2}$ dobijamo funkciju $(g^{m^2})^m = g^{m^3}$, itd. Nakon n primena operatora $\overline{m}$ dobijamo funkciju g^{m^n}. Dakle, izraz $\overline{n}\ \overline{m}\ g$ je ekvivalentan funkciji g^{m^n}. Otuda je $\overline{n}\ \overline{m}\ g\ x \Rightarrow_\beta^* g^{m^n}\ x$, što znači da je $\overline{n}\ \overline{m} \equiv_{\beta\eta} \lambda f.\lambda x.f^{m^n}\ x$, što odgovara numeralu $\overline{m^n}$.

Imajući u vidu poznate rezultate u teoriji izračunljivosti, slede neke zanimljive posledice prethodne teoreme.

Posledica 8.1. *Problem „Da li dati lambda izraz E ima normalnu formu?" je neodlučiv.*

Dokaz. Neka je P proizvoljan URM program. Ako je $f : \mathbb{N} \to \mathbb{N}$ parcijalna funkcija koju izračunava program P, tada znamo da je ona μ-rekurzivna (teorema 3.5), pa se prema prethodnoj teoremi 8.4 može predstaviti lambda izrazom. Neka je E takav lambda izraz. Izraz $E\ \overline{x}$ primenjuje E na Čerčov numeral koji odgovara broju x, a imaće normalnu formu ako i samo ako je $f(x)$ definisano, tj. ako i samo ako se program P zaustavlja za ulaz x. Ako bi problem postojanja normalne forme bio odlučiv, tada bismo odgovarajućom procedurom odlučivanja mogli da odlučujemo i o problemu zaustavljanja, što znamo da nije moguće. □

Posledica 8.2. *Problem ,,Da li su dva izraza E_1 i E_2 $\beta\eta$-ekvivalentni je neodlučiv.*

Dokaz. S obzirom na teoremu 8.3, ako bi ovaj problem bio odlučiv, to znači da bismo mogli da utvrdimo da li dva proizvoljna lambda izraza definišu istu funkciju. S obzirom da se funkcije koje URM programi izračunavaju mogu, prema teoremi 8.4, predstaviti lambda izrazima, sledi da bismo u tom slučaju mogli da odlučujemo i o ekvivalentnosti dva URM programa, što znamo da nije moguće. □

Teoremom 8.4 smo pokazali da se svaka μ-rekurzivna funkcija može predstaviti λ-izračunljivom funkcijom. Važi i obratno – svaka λ-izračunljiva funkcija se može predstaviti μ-rekurzivnom funkcijom. Formalno, neka je $f : \Lambda^k \to \Lambda$ bilo koja parcijalna funkcija arnosti k nad lambda izrazima. Ideja je da lambda izraze, kao hijerarhijske strukture, kodiramo prirodnim brojevima. Neka je $\gamma : \Lambda \to \mathbb{N}$ jedno takvo kodiranje. Funkciji f ćemo pridružiti parcijalnu funkciju $f_\mu = \gamma \circ f \circ \gamma^{-1} : \mathbb{N}^k \to \mathbb{N}$ koja radi sa kôdovima lambda izraza i za koju važi da je $f_\mu(\gamma(E_1), \ldots, \gamma(E_k)) = \gamma(F)$ ako i samo ako je $f(E_1, \ldots, E_k) = F$. Sada imamo sledeću teoremu.

Teorema 8.5. *Neka je $f : \Lambda^k \to \Lambda$ proizvoljna λ-izračunljiva funkcija arnosti k i neka je γ proizvoljno efektivno izračunljivo kodiranje lambda izraza prirodnim brojevima. Tada je odgovarajuća funkcija f_μ μ-izračunljiva.*

Dokaz. (skica) Imajući u vidu ekvivalentnost μ-izračunljivosti i URM-izračunljivosti, dovoljno je pokazati da postoji URM program koji izračunava funkciju f_μ. Kako je γ efektivno izračunljivo, tada je moguće URM programom vršiti kodiranje i dekodiranje. Otuda je na osnovu kôda izraza $(\lambda x.E)\ F$ moguće URM programom izračunati kôd izraza $E[x \mapsto F]$, tj. moguće je efektivno odrediti rezultat β-redukcije. Slično, na osnovu kôda izraza $\lambda x.E\ x$ moguće je URM programom izračunati kôd izraza E, tj. moguće je efektivno odrediti rezultat η-redukcije. Samim tim, moguće je simulirati redukciju lambda izraza u normalnom poretku i kao rezultat dobiti kôd odgovarajuće $\beta\eta$-normalne forme, ako ista postoji. Odavde sledi da je funkciju f_μ moguće izračunati URM programom, pa ju je moguće predstaviti i kao μ-rekurzivnu funkciju. □

Glava 9

Teorija tipova

Poznato je da algoritmi mogu raditi sa različitim tipovima podataka. To mogu biti prirodni, celi ili realni brojevi, tekstualni podaci, kao i složenije strukture podataka poput nizova, grafova, stabala, izraza, formula i sl. Pritom, u fiksiranom modelu izračunavanja, reprezentacija svih ovih podataka ima istu prirodu. Na primer, u formalizmu lambda izraza, svi podaci se predstavljaju upravo lambda izrazima. U formalizmu URM programa, svi podaci se kodiraju prirodnim brojevima. Tjuringove mašine reprezentuju podatke u obliku niski nad datom azbukom. Najzad, u realnim računarima, svi podaci se predstavljaju nizovima uzastopnih bajtova u memoriji. S obzirom da se svi podaci u fiksiranom modelu izračunavanja reprezentuju na istovetan način, sama reprezentacija podatka nema nikakvo unapred pridruženo značenje – njoj semantički smisao daje algoritam koji se na nju primenjuje.

Na primer, setimo se da su u lambda računu prirodni brojevi bili predstavljani lambda izrazima oblika $\lambda f.\lambda x.f^n\ x$, tzv. Čerčovim numeralima. Međutim, ova reprezentacija ni na koji način nije predodređena da predstavlja prirodne brojeve. Na primer, izraz $\lambda f.\lambda x.x$ koji predstavlja broj 0 je ujedno korišćen i za predstavljanje vrednosti $false$ (logičkog tipa), kao i za vrednost nil koja je predstavljala praznu listu. Dakle, ista reprezentacija može imati različit smisao, u zavisnosti od toga koji se algoritam na nju primenjuje.

Ključno pitanje je šta će se dogoditi ako neki algoritam primenimo na podatak „pogrešnog" tipa? Na primer, funkcija *succ* je dizajnirana tako da radi sa Čerčovim numeralima: za numeral $\overline{n}$ ova funkcija je vraćala numeral $\overline{n+1}$. Razmotrimo šta će se dogoditi ako ovu funkciju primenimo na vrednost $true = \lambda xy.x$:

$$succ\ true = (\lambda n.\lambda f.\lambda x.f\ (n\ f\ x))\ true \Rightarrow_\beta$$
$$\lambda f.\lambda x.f\ (true\ f\ x) \Rightarrow_\beta \lambda f.\lambda x.f\ f$$

Dobijeni izraz $\lambda f.\lambda x.f\ f$ nije Čerčov numeral i u kontekstu aritmetike ne predstavlja ništa. Dakle, kada primenimo funkciju na pogrešan tip, ona će nad tim podatkom uraditi *nešto*, ali to tipično neće imati nikakav smisao i dobijeni rezultat svakako neće biti ono što smo želeli.

Sličan fenomen se uočava i u slučaju realnih računara, kada programiramo na jezicima niskog nivoa, poput asemblera. Pretpostavimo da se na adresi x nalazi 4-bajtna reprezentacija koja predstavlja zapis podatka tipa $float$ – realnog broja

u pokretnom zarezu (zapis „znak-mantisa-eksponent"). Ako na ovaj podatak primenimo asemblersku instrukciju:

```
add eax, x
```

pokrenućemo *celobrojno* sabiranje ovog binarnog sadržaja sa sadržajem registra `eax`. S obzirom da binarni sadržaj na adresi x nije zapis celog broja, rezultat celobrojnog sabiranja nad ovim zapisom će biti potpuno besmislen binarni sadržaj koji u kontekstu realnih brojeva u pokretnom zarezu predstavlja potpuno pogrešnu vrednost. Ipak, ova instrukcija će se bez ikakvih problema izvršiti, ne dajući korisniku nikakvu naznaku da nešto nije u redu. Dakle, instrukcija će nad svakom binarnom reprezentacijom uraditi *nešto*, ali to izvesno neće biti ono što smo želeli.

Iz prethodnog razmatranja sledi jedan praktičan problem: kada pišemo programe, postoji opasnost da nehotice primenimo neku operaciju na podatak pogrešnog tipa. S obzirom da se svaka operacija može primeniti na bilo koju reprezentaciju, tako dobijeni program će raditi nešto, ali to neće biti ono što želimo. Dakle, tako napisani program će biti *sintaksno* ispravan, tj. biće formiran u skladu sa pravilima za konstrukciju programa u datom jeziku (ili formalizmu), ali će biti *semantički* neispravan, jer njegovo značenje neće biti ono što smo mi želeli da bude.

Na ovom mestu je zgodno setiti se Rajsove teoreme: za razliku od sintaksnih svojstava programa koja su po pravilu odlučiva, svako netrivijalno semantičko svojstvo programa je uvek neodlučivo. To znači da mi možemo efektivno proveriti da li je program sintaksno ispravan, ali ne i da li radi ono što mi želimo da radi. Zbog toga je otkrivanje semantičkih grešaka mnogo teže od pronalaženja sintaksnih grešaka (koje nam obično prijavljuje sam prevodilac). Pritom, greške vezane za pogrešnu upotrebu tipova podataka spadaju u najčešće semantičke greške koje se u praksi javljaju.

Jedan pristup rešavanju ovog problema je da se razmatranje upotrebe tipova podataka u izvesnoj meri prebaci sa terena semantike na teren sintakse: želimo da naš jezik za opis algoritama sintaksno obogatimo tako da uključuje mogućnost specifikacije tipova podataka, uz mogućnost provere (na sintaksnom nivou) da li se tipovi ispravno koriste. Za takve jezike (ili formalizme) kažemo da su *tipizirani*, dok jezici koju takvu sintaksnu mogućnost nemaju kažemo da su *netipizirani*. Na primer, asemblerski jezici su po pravilu netipizirani, dok je većina programskih jezika visokog nivoa tipizirano. Kada su u pitanju teorijski modeli izračunavanja, lambda račun koji smo opisali u prethodnoj glavi je netipiziran. Isto važi i za ostale formalizme koje smo upoznali u ovom tekstu (URM programi, μ-rekurzivne funkcije, Tjuringove mašine). U ovoj glavi upoznaćemo se sa *tipiziranim lambda računom*, kao formalnim modelom izračunavanja koji uključuje podršku za tipove u svojoj sintaksi. Tipizirani lambda račun je teorijski osnov za sve moderne tipizirane programske jezike, a naročito za funkcionalne programske jezike koji su poznati po veoma moćnim sistemima tipova.

9.1 Prosti tipovi

U ovom poglavlju se upoznajemo sa najjednostavnijom teorijom tipova koja postoji – teorijom *prostih tipova* (engl. *simple type theory*). Ova teorija je do-

voljna da opišemo osnovne tipove podataka, kao i tipove funkcija koje se na njih primenjuju.

Definicija 9.1. Neka je dat najviše prebrojiv skup *atomičkih tipova* Σ. *Prost tip* dobijamo konačnom primenom sledećih pravila:

- svaki atomički tip $\sigma \in \Sigma$ je prost tip
- ako su σ i τ prosti tipovi, tada je i $\sigma \to \tau$ prost tip.

Skup svih prostih tipova označavaćemo sa $\mathcal{T}_S$.

Napomena 9.1. Operator $\to$ je desno asocijativan. To znači da $\sigma \to \tau \to \rho$ znači $\sigma \to (\tau \to \rho)$, a *ne* $(\sigma \to \tau) \to \rho$.

Intuitivno, operator $\to$ formira *funkcijske tipove*. Tip $\sigma \to \tau$ predstavlja funkciju koja prihvata argument tipa σ i vraća vrednost tipa τ. Pritom, funkcije uvek prihvataju samo jedan argument. Funkcije veće arnosti se predstavljaju kariranjem. Na primer, tip funkcije koja prihvata argumente tipa σ_1 i σ_2 i vraća tip τ se predstavlja tipom $\sigma_1 \to \sigma_2 \to \tau$. Imajući u vidu prethodnu primedbu o desnoj asocijativnosti operatora $\to$, ovo je zapravo funkcija koja prihvata argument tipa σ_1 i vraća *funkciju* tipa $\sigma_2 \to \tau$. Ova funkcija se sada primenjuje na argument tipa σ_2 i vraća vrednost tipa τ.

Primer 9.1. Pretpostavimo da imamo atomičke tipove `Int` i `Bool`. Tip `Int` $\to$ `Bool` predstavlja tip funkcije koja prihvata argument tipa `Int` i vraća vrednost tipa `Bool`. Tip `Int` $\to$ `Int` $\to$ `Int` (tj. `Int` $\to$ (`Int` $\to$ `Int`)) predstavlja tip funkcije koja prihvata `Int` i vraća funkciju koja prihvata `Int` i vraća `Int`. Imajući u vidu kariranje, ovim tipom ćemo predstavljati i funkcije koje prihvataju dva argumenta tipa `Int` i vraćaju rezultat tipa `Int`. Sa druge strane, tip (`Int` $\to$ `Int`) $\to$ `Bool` predstavlja funkciju koja prihvata kao argument funkciju tipa `Int` $\to$ `Int`, a vraća vrednost tipa `Bool`.

Na prostim tipovima je zasnovan *prosto tipizirani lambda račun*, kao jedan od osnovnih tipiziranih formalizama za opis algoritama.

Definicija 9.2. Neka je dat skup atomičkih tipova Σ i skup *promenljivih* V. *Kontekst* Γ je zadat konačnim skupom parova $x_1 : \sigma_1, \ldots, x_n : \sigma_n$, gde su x_i promenljive, a σ_i prosti tipovi pridruženi tim promenljivama. *Prosto tipizirani lambda izrazi nad kontekstom* Γ nastaju konačnom primenom sledećih pravila:

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma}\ var$$

$$\frac{\Gamma, x : \sigma \vdash E : \tau}{\Gamma \vdash (\lambda x : \sigma . E) : \sigma \to \tau}\ abs$$

$$\frac{\Gamma \vdash E : \sigma \to \tau \qquad \Gamma \vdash F : \sigma}{\Gamma \vdash (E\ F) : \tau}\ app$$

pri čemu oznaka $\Gamma \vdash E : \sigma$ znači da je E izraz koji u kontekstu Γ ima tip σ.

Intuitivno, pravilo *var* kaže da je svaka promenljiva x kojoj je u datom kontekstu pridružen tip σ sintaksno ispravan izraz tipa σ u tom kontekstu. Pravilo *abs* kaže da ako na izraz E tipa τ primenimo apstrakciju po promenljivoj x tipa σ, tada dobijamo izraz tipa $\sigma \to \tau$ (dakle, dobijamo funkciju koja prihvata

argument tipa σ i vraća vrednost tipa τ). Najzad, pravilo *app* kaže da ako izraz E funkcijskog tipa $\sigma \to \tau$ primenimo na izraz F tipa σ, dobijamo izraz tipa τ (funkcija se primenjuje na argument tipa σ i vraća rezultat tipa τ).

Primetimo da sintaksno ispravne tipizirane lambda izraze nad zadatim kontekstom Γ moguće dobiti samo primenom gornjih pravila. Otuda izraz može biti sintaksno ispravan samo ako je *ispravno tipiziran* (engl. *well-typed*), tj. ako mu se može pridružiti tip u datom kontekstu u skladu sa gore navedenim pravilima. Između ostalog, ovo znači da izraz može sadržati samo one slobodne promenljive kojima je u kontekstu pridružen tip. Intuitivno, kontekst se može razumeti kao skup *deklaracija promenljivih* koje se mogu koristiti u izrazu izrazu. Ono što nije deklarisano, ne može se koristiti, jer nije poznato kog je tipa. Ovo odgovara uobičajenoj praksi u programskim jezicima koji zahtevaju eksplicitnu deklaraciju promenljivih pre njihove upotrebe. Takođe, izraz se može dobiti apstrakcijom po nekoj promenljivoj x kojoj je u kontekstu pridružen tip σ, pri čemu se tom apstrakcijom par $x : \sigma$ *briše* iz konteksta, ali se zato informacija o njenom tipu sintaksno ugrađuje u sam izraz (pišemo $\lambda x : \sigma$, a ne samo λx, kao ranije). Intuitivno, apstrahovanjem promenljive postaju *lokalni parametri* funkcija, te se deklarišu u sklopu apstrakcije, gde se navodi i njihov tip. Nakon što izvršimo apstrakciju, mi izlazimo iz tela funkcije, te njeni lokalni parametri prestaju da budu vidljivi, pa ih zato brišemo iz konteksta. Dakle, uloga konteksta je da sadrži informacije o tipovima slobodnih promenljivih koje su vidljive i mogu se koristiti u izrazu na datom nivou. Posledično, ako izraz ne sadrži slobodne promenljive, onda kontekst može biti i prazan, tj. možemo imati da je:

$$\vdash E : \sigma$$

Ovo znači da je E zatvoren izraz tipa σ, nezavisno od konteksta. U takvim situacijama ćemo često pisati i samo $E : \sigma$.

Najzad, primetimo da je u skladu sa pravilom *app* izraz E moguće primeniti na izraz F samo ako je izraz E nekog funkcijskog tipa $\sigma \to \tau$, a izraz F je upravo argument odgovarajućeg tipa σ koji funkcija i očekuje. U suprotnom, izraz $E\ F$ će biti pogrešno tipiziran, pa će samim tim biti sintaksno neispravan izraz. Dakle, uveli smo sintaksno ograničenje kojim sprečavamo primenu funkcije na argument pogrešnog tipa. Dakle, više nije moguće primenjivati bilo koju operaciju na bilo koji podatak, već samo na podatke odgovarajućeg tipa, čime rešavamo polazni problem zbog koga smo i uveli tipizaciju.

Primer 9.2. Neka je dat kontekst $\Gamma = \{x : \sigma, y : \tau, f : \sigma \to \tau \to \rho\}$. Odredimo tip izraza $f\ x\ y$ u kontekstu Γ. Imamo sledeće izvođenje:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash f : \sigma \to \tau \to \rho}\,var \qquad \dfrac{}{\Gamma \vdash x : \sigma}\,var}{\Gamma \vdash f\ x : \tau \to \rho}\,app \qquad \dfrac{}{\Gamma \vdash y : \tau}\,var}{\Gamma \vdash f\ x\ y : \rho}\,app$$

Dakle, izraz $f\ x\ y$ je ispravno tipiziran u kontekstu Γ i ima tip ρ. Dalje, odredimo tip izraza $\lambda g : \sigma \to \sigma . f\ (g\ x)\ y$ u kontekstu Γ:

$$\dfrac{\dfrac{\dfrac{}{\Gamma' \vdash f : \sigma \to \tau \to \rho}\,var \quad \dfrac{\dfrac{}{\Gamma' \vdash g : \sigma \to \sigma}\,var \quad \dfrac{}{\Gamma' \vdash x : \sigma}\,var}{\Gamma' \vdash g\ x : \sigma}\,app}{\Gamma' \vdash f\ (g\ x) : \tau \to \rho}\,app \quad \dfrac{}{\Gamma' \vdash y : \tau}\,var}{\dfrac{\Gamma' \vdash f\ (g\ x)\ y : \rho}{\Gamma \vdash \lambda g. : \sigma \to \sigma . f\ (g\ x)\ y : (\sigma \to \sigma) \to \rho}\,abs}\,app$$

gde je $\Gamma' = \Gamma \cup \{g : \sigma \to \sigma\}$. Dakle, i ovaj izraz je ispravno tipiziran u kontekstu Γ, a ima tip $(\sigma \to \sigma) \to \tau$. U ovom izrazu, promenljiva g je lokalni parametar tipa $\sigma \to \sigma$, dok su ostale promenljive zadate u kontekstu.

Primer 9.3. Razmotrimo izraz $\lambda f : \sigma \to \tau . \lambda x : \sigma . f\ x$. Ovaj izraz je zatvoren, pa njegov tip ne zavisi od konteksta. Odredimo njegov tip. Imamo sledeće izvođenje:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f : \sigma \to \tau, x : \sigma \vdash f : \sigma \to \tau}\,var \quad \dfrac{}{f : \sigma \to \tau, x : \sigma \vdash x : \sigma}\,var}{f : \sigma \to \tau, x : \sigma \vdash f\ x : \tau}\,app}{f : \sigma \to \tau \vdash \lambda x : \sigma . f\ x : \sigma \to \tau}\,abs}{\vdash \lambda f : \sigma \to \tau . \lambda x : \sigma . f\ x : (\sigma \to \tau) \to \sigma \to \tau}\,abs$$

Dakle, imamo da je tip izraza $\lambda f : \sigma \to \tau . \lambda x : \sigma . f\ x$ jednak $(\sigma \to \tau) \to \sigma \to \tau$. Samim tim, izraz je ispravno tipiziran.

U prethodna dva primera smo demonstrirali jedan od osnovnih postupaka u teoriji tipova – *određivanje tipa izraza* (engl. *type inference*). U pitanju je deduktivni postupak u kome se pravila tipizacije primenjuju kao *pravila izvođenja*, a ceo sistem tipova se razmatra kao deduktivni sistem u kome konstruišemo *izvođenje tipa* (engl. *type derivation*) u obliku stabla. U slučaju prosto tipiziranih lambda izraza, postupak određivanje tipa datog izraza E u datom kontekstu Γ je efektivan i sprovodi se rekurzivno na sledeći način:

- ako je E promenljiva x, tada je potrebno proveriti da li u kontekstu Γ postoji par $x : \sigma$. Ako postoji, tada je E tipa σ, dok se u suprotnom izrazu E ne može pridružiti tip
- ako je E oblika $\lambda x : \sigma . E'$, tada se rekurzivno određuje tip izraza E' u kontekstu $\Gamma \cup \{x : \sigma\}$. Ako u tom kontekstu izraz E' ima tip τ, tada izraz $E = \lambda x : \sigma . E'$ u kontekstu Γ ima tip $\sigma \to \tau$. U suprotnom, izrazu E se ne može pridružiti tip
- ako je E oblika $E_1\ E_2$, tada je potrebno rekurzivno utvrditi tipove izraza E_1 i E_2 u kontekstu Γ. Ako izrazi E_1 i E_2 imaju tipove σ_1 i σ_2 u kontekstu Γ, tada je neophodno da tip σ_1 bude oblika $\sigma_2 \to \tau$, u kom slučaju će izraz $E = E_1\ E_2$ imati tip τ u kontekstu Γ. U suprotnom, izrazu $E_1\ E_2$ nije moguće pridružiti tip.

Dakle, ovim rekurzivnim postupkom se stablo izvođenja konstruiše od korena ka listovima. Rezultat postupka može biti ili tip koji je pridružen izrazu u datom kontekstu, ili informacija da izrazu nije moguće pridružiti tip.

S obzirom da je izraz ispravno tipiziran ako i samo ako mu se može pridružiti tip, opisanim postupkom određivanja tipa se ujedno odgovara i na pitanje ,,Da li je dati izraz E u kontekstu Γ ispravno tipiziran?". Samim tim, u teoriji prostih tipova, ovaj problem je *odlučiv*.

Primer 9.4. Neka je dat izraz $\lambda x : \sigma.x\ x$. Ovaj izraz nije ispravno tipiziran. Zaista, da bismo odredili tip ovog izraza, potrebno je rekurzivno odrediti tip izraza $x\ x$ u kontekstu $\Gamma = \{x : \sigma\}$. Međutim, ovom izrazu nije moguće pridružiti tip, jer bi levo x moralo da ima tip oblika $\sigma \to \tau$, da bi moglo da se primeni na desno x koje je tipa σ. Međutim, i levo i desno x su istog tipa (σ), tako da ovakva primena nije ispravna.

Uopšte, izraz $E\ E$ nikada nije ispravno tipiziran izraz, tj. ni jedan izraz se u prosto tipiziranom lambda računu se ne može primeniti na samog sebe.

Primer 9.5. Razmotrimo izraz $\lambda f : \sigma \to \tau.\lambda g : \sigma \to \tau.\lambda x : \sigma.f\ (g\ x)$. Da bismo odredili tip ovog izraza, potrebno je odrediti tip izraza $f\ (g\ x)$ u kontekstu $\Gamma = \{f : \sigma \to \tau, g : \sigma \to \tau, x : \sigma\}$. Da bi izraz $f\ (g\ x)$ bio ispravno tipiziran u Γ potrebno je da izraz $g\ x$ ima tip σ, kako bi funkcija f tipa $\sigma \to \tau$ mogla da bude primenjena na njega. Međutim, kako je g tipa $\sigma \to \tau$, tada je rezultat funkcije g tipa τ, pa izraz $g\ x$ nije tipa σ. Otuda ceo izraz nije ispravno tipiziran.

U prethodnom primeru, videli smo da je za proveru ispravnosti tipizacije izraza potrebno proveriti da li se funkcije primenjuju na argumente odgovarajućih tipova. Ovo se svodi na pitanje „Da li dati izraz E u kontekstu Γ ima tip σ?". Ovaj problem poznat je i kao problem *provere tipova* (engl. *type checking*). U slučaju prostih tipova, i ovaj problem je odlučiv, a može se svesti na određivanje tipa izraza, a zatim na sintaksno upoređivanje dobijenog tipa sa datim tipom.

Primer 9.6. Neka je $E = \lambda f : \sigma \to \tau.\lambda x : \sigma.f\ x$ izraz iz primera 9.3 za koji smo utvrdili da je tipa $(\sigma \to \tau) \to \sigma \to \tau$. Neka je dat kontekst $\Gamma = \{u : \tau\}$. Ispitajmo da li je izraz $F = E\ (\lambda v : \sigma.u)$ sintaksno ispravan u kontekstu Γ. S obzirom da znamo da je E sintaksno ispravan u bilo kom kontekstu, dovoljno je proveriti da li je aplikacija izraza E na izraz $\lambda v : \sigma.u$ sintaksno ispravna u kontekstu Γ. Za to je dovoljno da izraz $\lambda v : \sigma.u$ u kontekstu Γ ima tip $\sigma \to \tau$ (dakle, imamo problem provere tipa). Kako je $\Gamma \vdash u : \tau$, sledi da je $\Gamma \vdash \lambda v : \sigma.u : \sigma \to \tau$, pa je izraz F sintaksno ispravan u Γ i ima tip $\sigma \to \tau$.

Napomena 9.2. Primetimo da se u opisanom postupku određivanja tipa problem provere tipa javlja pri svakoj primeni operatora aplikacije. Stablo izvođenja tipa se konstruiše od listova ka korenu, polazeći od promenljivih deklarisanih u odgovarajućem kontekstu, a zatim se za svaku primenu operatora aplikacije proverava da li je tip argumenta sintaksno identičan tipu koji funkcija očekuje. Na ovaj način se obezbeđuje da je stablo izvođenja tipa ispravno konstruisano, te da za ceo izraz znamo da je ispravno tipiziran i da ima tip koji je određen tim izvođenjem.

U realnim programskim jezicima, provera tipa se vrši za argumente svakog poziva funkcije, kao i za podizraze na koje se primenjuju operatori. Ovo se radi u dubinu, do najjednostavnijih izraza – promenljivih, za koje se na osnovu deklarisanog tipa utvrđuje da li su odgovarajućeg tipa ili ne. Ovo upravo odgovara konstrukciji stabla izvođenja tipa u tipiziranom lambda računu.

9.1.1 Redukcija tipiziranih lambda izraza

U ovom odeljku razmatramo pitanje ponašanja uobičajenih operacija redukcije (β i η redukcija) u kontekstu tipiziranih lambda izraza. U tom smislu, ključno pitanje je da li se redukcijom ispravno tipiziranog lambda izraza ponovo dobija ispravno tipizirani lambda izraz, kao i koji tip se tako redukovanom izrazu

može pridružiti. Naredna teorema tvrdi da se β redukcijom ispravno tipiziranog izraza tipa σ ponovo dobija ispravno tipizirani izraz tipa σ.

Teorema 9.1. *Ako za neki izraz E važi $\Gamma \vdash E : \sigma$ i ako je $E \Rightarrow_\beta E'$, tada je i $\Gamma \vdash E' : \sigma$.*

Dokaz. Dokaz možemo sprovesti indukcijom po dubini redex čvora u sintaksnom stablu izraza na koji se primenjuje β-redukcija.

Razmotrimo najpre slučaj kada se redukcija vrši nad čvorom dobine 0, tj. nad korenim čvorom stabla. Drugim rečima, neka je dati izraz oblika $(\lambda x : \tau.E)\ F$ i potrebno je izvršiti β-redukciju nad tim redex-om na najvišem nivou. Pretpostavimo da je ovaj izraz ispravno tipiziran u datom kontekstu Γ. Ako je E izraz tipa σ u kontekstu $\Gamma \cup \{x : \tau\}$, tada je funkcija $\lambda x : \tau.E$ tipa $\tau \to \sigma$ u kontekstu Γ. Izraz F tada mora biti tipa τ, pa će ceo izraz biti tipa σ u kontekstu Γ. Primenom β-redukcije na ovaj redex na najvišem nivou dobijamo izraz $E[x \mapsto F]$. Kako se slobodna pojavljivanja promenljive x tipa τ zamenjuju izrazom F istog tipa τ, nijedno pravilo tipizacije unutar izraza E neće biti narušeno, pa će i ovaj izraz biti ispravno tipiziran u kontekstu Γ i imaće isti tip kao i izraz E – biće tipa σ.

Pretpostavimo sada da tvrđenje važi za redex-e na dubini manjoj od k i pretpostavimo da želimo da redukujemo neki redex na dubini k. Potrebno je razmotriti dva slučaja:

- izraz je oblika $E_1\ E_2$. Ako je ovaj izraz ispravno tipiziran u kontekstu Γ, to znači da je tip izraza E_1 oblika $\tau \to \sigma$, dok je tip izraza E_2 jednak τ. Pretpostavimo najpre da se redex koji redukujemo nalazi u izrazu E_1. Unutar stabla izraza E_1 taj redex se nalazi na dubini $k-1$, pa prema induktivnoj pretpostavci sledi da njegovim redukovanjem dobija izraz E_1' koji je ispravno tipiziran i istog tipa $\tau \to \sigma$. Otuda i ceo izraz $E_1'\ E_2$ ostaje ispravno tipiziran i tipa σ. Slično važi i u slučaju da se redex nalazi u izrazu E_2.

- izraz je oblika $\lambda x : \tau.E$. Ako je ovaj izraz ispravno tipiziran u kontekstu Γ, onda je njegov tip oblika $\tau \to \sigma$, gde je σ tip izraza E u kontekstu $\Gamma \cup \{x : \tau\}$. Redex koji redukujemo se mora nalaziti u izrazu E, na dubini $k-1$. Po induktivnoj pretpostavci, njegovom redukcijom dobija se izraz E' koji je ispravno tipiziran i ima isti tip σ u kontekstu $\Gamma \cup \{x : \tau\}$. Otuda i izraz $\lambda x : \tau.E'$ ostaje ispravno tipiziran i ima tip $\tau \to \sigma$ u kontekstu Γ.

Odavde, prema principu indukcije, sledi da se prilikom primene β-redukcije nad redex-om na bilo kojoj dubini u stablu izraza dobija ispravno tipiziran lambda izraz koji ostaje istog tipa kao i polazni izraz. □

Dakle, β-redukcija je ,,bezbedna" operacija, jer ne narušava pravila tipizacije, pa se može sprovoditi nad prosto tipiziranim izrazima na isti način kao i nad netipiziranim izrazima. Slično važi i za η-redukciju, o čemu govori sledeća teorema.

Teorema 9.2. *Ako je E ispravno tipiziran izraz tipa σ u kontekstu Γ i ako važi $E \Rightarrow_\eta E'$, tada je i E' ispravno tipiziran izrazu u kontekstu Γ i ima tip σ.*

Dokaz. Slično kao i u dokazu prethodne teoreme, možemo primeniti indukciju po dubini čvora nad kojim se vrši redukcija. U slučaju da je dubina $k = 0$, to znači da imamo izraz oblika $\lambda x : \tau.E\ x$, pri čemu E ne sadrži slobodna pojavljivanja promenljive x. Ako je ovaj izraz ispravno tipiziran u kontekstu Γ, tada je E ispravno tipiziran izraz nekog tipa oblika $\tau \to \sigma$ u kontekstu Γ, pa je izraz $E\ x$ tipa σ u kontekstu $\Gamma \cup \{x : \tau\}$, dok je ceo izraz $\lambda x : \tau.E\ x$ tipa $\tau \to \sigma$ u kontekstu Γ. Primenom η redukcije nad ovim izrazom na najvišem nivou dobijamo izraz E koji je ispravno tipiziran i tipa $\tau \to \sigma$ u kontekstu Γ. Induktivni korak se sprovodi na potpuno isti način kao u dokazu prethodne teoreme. □

Posledica 9.1. *Ako je E ispravno tipiziran izraz tipa σ u kontekstu Γ i ako važi $E \Rightarrow^*_{\beta\eta} E'$, tada je i E' ispravno tipiziran izraz tipa σ u kontekstu Γ.*

Dokaz. Indukcijom po dužini izvođenja. □

9.2 Implicitna tipizacija

U prethodnom poglavlju smo razmatrali tipizirane lambda izraze kod kojih se tip svake promenljive zapisuje eksplicitno prilikom apstrakcije. Na primer, izraz $I_\sigma = \lambda x : \sigma.x$ predstavlja identičku funkciju tipa $\sigma \to \sigma$, a informacija o tipu parametra funkcije je eksplicitno navedena u samom izrazu. Ovakvu tipizaciju nazivamo *eksplicitnom tipizacijom*, a pripisujemo je Čerču (engl. *Church-style typing*). Ona je karakteristična za većinu klasičnih programskih jezika, kao što su *C*, *C++*, *Java*, *C#*, *Pascal*, *Fortran* i sl. U ovim programskim jezicima programer mora eksplicitno da deklariše promenljive koje će koristiti, pri čemu navodi i njihove tipove (tj. zadaje kontekst). Takođe, programer mora eksplicitno navesti tipove parametara i povratnih vrednosti funkcija koje definiše. Ovim posao programskog prevodioca postaje relativno lak – potrebno je samo vršiti proveru tipova u izrazima u kojima se te promenljive i funkcije koriste. Sa druge strane, teret tipizacije promenljivih se prebacuje na programera, koji mora ispravno da odredi i eksplicitno zapiše tipove svih promenljivih, kao i funkcija u svom programu.

Alternativni pristup je da programer ne navodi tipove funkcija i promenljivih, već da kompletnu rekonstrukciju tipova prepusti prevodiocu, koji će tipove odrediti iz konteksta u kojima se te promenljive i funkcije koriste. Ovakva tipizacija se naziva *implicitna tipizacija*, a pripisuje se Kariju (engl. *Curry-style typing*). Karakteristična je za moderne funkcionalne programske jezike, poput jezika *Haskell*.

U ovom poglavlju ćemo razmotriti implicitnu tipizaciju u okviru teorije prostih tipova. Uvedimo najpre sledeću definiciju, veoma sličnu definiciji 9.2, ali ovoga puta, bez eksplicitnog navođenja tipova u apstrakcijama.

Definicija 9.3. Neka je dat skup atomičkih tipova Σ i skup *promenljivih* V. *Kontekst* Γ je zadat konačnim skupom parova $x_1 : \sigma_1, \ldots, x_n : \sigma_n$, gde su x_i promenljive, a σ_i prosti tipovi pridruženi tim promenljivama. *Implicitno prosto tipizirani lambda izrazi nad kontekstom* Γ nastaju konačnom primenom sledećih pravila:

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma}\ var$$

$$\frac{\Gamma, x : \sigma \vdash E : \tau}{\Gamma \vdash (\lambda x.E) : \sigma \to \tau}\ abs$$

$$\frac{\Gamma \vdash E : \sigma \to \tau \qquad \Gamma \vdash F : \sigma}{\Gamma \vdash (E\ F) : \tau}\ app$$

pri čemu oznaka $\Gamma \vdash E : \sigma$ znači da je E izraz kome u kontekstu Γ može biti pridružen tip σ.

Ono što primećujemo u prethodnoj definiciji je da je sada sintaksa lambda izraza potpuno ista kao i u netipiziranom lambda računu. Na primer, umesto izraza $I_\sigma = \lambda x : \sigma.x$ imamo izraz $I = \lambda x.x$, baš kao i u netipiziranom lambda računu. Takođe, u prethodnoj definiciji stoji da oznaka $\Gamma \vdash E : \sigma$ znači da tip σ *može* biti pridružen izrazu E, a ne da E obavezno ima taj tip. Naime, s obzirom da se sada tipovi promenljivih ne deklarišu eksplicitno, isti izraz može biti tipiziran različitim tipovima. Na primer, izraz $I_\sigma = \lambda x : \sigma.x$ je za svaki fiksirani tip σ imao tačno jedan tip – $\sigma \to \sigma$. Zato smo za svaki tip σ morali da imamo po jednu identičku funkciju I_σ koja se primenjuje na taj tip. Kod implicitne tipizacije imamo samo jedan izraz $I = \lambda x.x$ za koji *za svaki* tip σ imamo jedno ovakvo izvođenje:

$$\frac{\dfrac{}{x : \sigma \vdash x : \sigma}\ var}{\vdash \lambda x.x : \sigma \to \sigma}\ abs$$

Otuda izrazu I može biti pridružen bilo koji tip oblika $\sigma \to \sigma$, za svaki mogući prost tip σ. Dakle, više ne govorimo o *jednom* tipu koji je pridružen datom izrazu, već o *familiji* tipova koji mogu biti pridruženi tom izrazu, u zavisnosti od konteksta u kome se nađe. Svojstvo izraza da mu se mogu pridružiti različiti tipovi naziva se *polimorfizam*.

Definicija 9.4. Za netipizirani lambda izraz kažemo da je *tipizibilan* (engl. *typable*) ako može biti implicitno tipiziran, tj. ako mu pravilima implicitne tipizacije može biti pridružen bar jedan tip.

Dakle, kod implicitne tipizacije ne govorimo o *ispravno tipiziranim* lambda izrazima, već o *tipizibilnim* lambda izrazima, tj. mi razmatramo netipizirane izraze i postavljamo pitanje da li se oni mogu tipizirati.

Između eksplicitno tipiziranih lambda izraza i tipizibilnosti netipiziranih lambda izraza postoji jednostavna veza koja sledi direktno iz strukture odgovarajućih pravila izvođenja u netipiziranom i tipiziranom slučaju. Ova veza je iskazana sledećom teoremom.

Teorema 9.3. *Neka je E bilo koji eksplicitno tipizirani lambda izraz za koji važi $\Gamma \vdash E : \sigma$. Ako je E^{nt} netipizirani izraz nastao od E brisanjem tipova svih promenljivih, tada važi $\Gamma \vdash E^{nt} : \sigma$.*

Dakle, ako je izraz E ispravno tipiziran izraz tipa σ, tada je njegov netipizirani ekvivalent E^{nt} tipizibilan i može mu se pridružiti isti tip σ. Ipak, za razliku od eksplicitno tipiziranog izraza E koji ima samo jedan tip, njegovom netipiziranom ekvivalentu E^{nt} se mogu pridružiti i drugi tipovi, s obzirom na prethodnu diskusiju.

Ako je izraz tipizibilan, tada se postavlja pitanje na koji način možemo opisati familiju tipova koji se mogu pridružiti tom izrazu? Poželjno bi bilo da za dati izraz postoji *najopštiji tip*, tj. neki tip takav da su svi tipovi datog izraza *instance* tog tipa. Da bismo ovo formalizovali, uvedimo najpre sledeću definiciju.

Definicija 9.5. Neka je dat skup *atomičkih* tipova Σ, kao i skup *tipskih promenljivih* (ili *tipskih parametara*) T. *Parametarski* (*šablonski, šematski*) prosti tip se dobija konačnom primenom sledećih pravila:

- atomički tip $\sigma \in \Sigma$ je parametarski prost tip
- tipska promenljiva $t \in T$ je parametarski prost tip
- ako su σ i τ parametarski prosti tipovi, tada je i $\sigma \to \tau$ parametarski prost tip.

Iz definicije 9.5 sledi da je svaki prost tip (u smislu definicije 9.1) ujedno i parametarski prost tip. Dodatno, parametarski prosti tipovi mogu sadržati i tipske promenljive (parametre) umesto atomičkih tipova. Za parametre je karakteristično da se oni mogu instancirati proizvoljnim tipovima. Ovo formalizujemo sledećom definicijom.

Definicija 9.6. *Zamenom* tipskih parametara $t_1, \ldots, t_k$ tipovima $\sigma_1, \ldots, \sigma_k$ u tipu τ (u oznaci $\tau[t_1 \mapsto \sigma_1, \ldots, t_k \mapsto \sigma_k]$) dobija se tip koji nastaje tako što se sva pojavljivanja tipske promenljive t_i u tipu τ zamene tipom σ_i. Za tip τ' kažemo da je *instanca* tipa τ, ako postoji zamena $s = [t_1 \mapsto \sigma_1, \ldots, t_k \mapsto \sigma_k]$ takva da je $\tau' = \tau s = \tau[t_1 \mapsto \sigma_1, \ldots, t_k \mapsto \sigma_k]$. Kažemo i da je τ *opštiji tip* od tipa τ', i to zapisujemo sa $\tau \preceq \tau'$.

Sada možemo definisati najopštiji tip tipizibilnog izraza. Ipak, s obzirom da izraz može sadržati i slobodne promenljive, potrebno je odrediti i *najopštiji kontekst* u kome je izraz moguće tipizirati. Formalno, imamo sledeću definiciju.

Definicija 9.7. Neka je E proizvoljni tipizibilni lambda izraz. Za par (Γ, τ), gde je Γ parametarski kontekst (tj. skup parova $x_1 : \sigma_1, \ldots, x_k : \sigma_k$, gde su σ_i prosti parametarski tipovi), a τ je parametarski tip, kažemo da je *najopštiji par* izraza E ako za svaki kontekst Γ' i tip τ' važi $\Gamma' \vdash E : \tau'$ ako i samo ako postoji zamena s takva da je $\Gamma' = \Gamma s$ i $\tau' = \tau s$. Pritom, za Γ kažemo da je *najopštiji kontekst* u kome se izraz E može tipizirati, a τ je *najopštiji tip* koji mu u tom kontekstu može biti dodeljen.

U skladu sa ovom definicijom, ako je (Γ, τ) najopštiji par pridružen izrazu E, tada svaki tip τ' koji se može pridružiti izrazu E predstavlja instancu tipa τ, a kontekst Γ' koji odgovara tipu τ' je instanca konteksta Γ, pri istoj instancijaciji. Dakle, najopštiji tip predstavlja *šematski tip* koji u sebi sadrži tipske parametre koji se mogu instancirati proizvoljnim prostim tipovima, i na taj način se mogu dobiti *svi* tipovi koji se datom izrazu mogu pridružiti. U slučaju da izraz sadrži slobodne promenljive, biće potrebno odrediti i najopštije tipove tih promenljivih takve da je izraz moguće tipizirati, što upravo predstavlja najopštiji kontekst. Ako izraz ne sadrži slobodne promenljive, taj kontekst će biti prazan, pa govorimo samo o najopštijem tipu izraza.

Napomena 9.3. Primetimo da ako je (Γ, τ) najopštiji par pridružen izrazu E, tada svakako postoji zamena s takva da je $\Gamma s = \Gamma$ i $\tau s = \tau$ (npr. možemo uzeti zamenu $[t \mapsto t]$ za proizvoljnu tipsku promenljivu t). To znači da, u skladu sa prethodnom definicijom, važi $\Gamma \vdash E : \tau$, tj. najopštiji tip τ je i sam jedan od tipova koji se mogu pridružiti izrazu E, a odgovarajući kontekst je upravo najopštiji kontekst Γ.

Postavlja se pitanje da li svaki tipizibilni lambda izraz E mora imati najopštiji par? S obzirom da za svaki kontekst Γ' i tip τ' za koji važi $\Gamma' \vdash E : \tau'$ mora postojati stablo izvođenja D' formirano prema pravilima implicitne tipizacije koje u korenu sadrži $\Gamma' \vdash E : \tau'$, sledi da bi u slučaju postojanja najopštijeg para (Γ, τ) izraza E postojalo i stablo izvođenja D koje izvodi $\Gamma \vdash E : \tau$, s obzirom na prethodnu napomenu. Pritom, stablo izvođenja D' za proizvoljno $\Gamma' \vdash E : \tau'$ bi bilo *instanca* stabla D, tj. postojala bi zamena s koju kada primenimo na sve čvorove stabla D dobijamo stablo D'. Ovakvo izvođenje nazivaćemo *najopštije izvođenje*. Dakle, najopštiji par će postojati ako i samo ako postoji najopštije izvođenje.

Primer 9.7. Razmotrimo izraz $\lambda x.f\ x$. Ovaj izraz sadrži slobodnu promenljivu f. Može se pokazati da je najopštije izvođenje koje se može pridružiti ovom izrazu dato u nastavku:

$$\dfrac{\dfrac{\dfrac{}{f : t_1 \to t_2, x : t_1 \vdash f : t_1 \to t_2}\,var \quad \dfrac{}{f : t_1 \to t_2, x : t_1 \vdash x : t_1}\,var}{f : t_1 \to t_2, x : t_1 \vdash f\ x : t_2}\,app}{f : t_1 \to t_2 \vdash \lambda x.f\ x : t_1 \to t_2}\,abs$$

Dakle, najopštiji kontekst u kome se izraz $\lambda x.f\ x$ može tipizirati je $f : t_1 \to t_2$, i u tom kontekstu ovaj izraz ima najopštiji tip $t_1 \to t_2$. Svaki konkretan tip koji ovaj izraz može imati je neka instanca ovog tipa, tj. neki tip oblika $\sigma_1 \to \sigma_2$ za neke proizvoljne tipove σ_1 i σ_2, pri čemu pretpostavljamo da je slobodnoj promenljivoj f u kontekstu dodeljen tip $\sigma_1 \to \sigma_2$. Izvođenje tvrđenja $f : \sigma_1 \to \sigma_2 \vdash \lambda x.f\ x : \sigma_1 \to \sigma_2$ se može dobiti od gornjeg najopštijeg izvođenja, primenom zamene $s = [t_1 \mapsto \sigma_1, t_2 \mapsto \sigma_2]$ na sve čvorove stabla:

$$\dfrac{\dfrac{\dfrac{}{f : \sigma_1 \to \sigma_2, x : \sigma_1 \vdash f : \sigma_1 \to \sigma_2}\,var \quad \dfrac{}{f : \sigma_1 \to \sigma_2, x : \sigma_1 \vdash x : \sigma_1}\,var}{f : \sigma_1 \to \sigma_2, x : t_1 \vdash f\ x : \sigma_2}\,app}{f : \sigma_1 \to \sigma_2 \vdash \lambda x.f\ x : \sigma_1 \to \sigma_2}\,abs$$

Napomena 9.4. Primetimo da smo u prethodnom primeru umesto tipskih promenljivih t_1 i t_2 mogli koristiti bilo koje druge tipske promenljive. Drugim rečima, preimenovanjem promenljivih u najopštijem paru ponovo dobijamo najopštiji par. Dakle, formalno posmatrano, najopštiji par nije jedinstven. Ipak, ako su dva najopštija para isti do na preimenovanje promenljivih, smatraćemo da se oni suštinski ne razlikuju.

Primer 9.8. Razmotrimo izraz $K = \lambda xy.x$. Ovaj izraz je zatvoren (nema slobodnih promenljivih), pa nam je potreban samo najopštiji tip. Imamo sledeće najopštije izvođenje:

$$\dfrac{\dfrac{\dfrac{}{x : t_1, y : t_2 \vdash x : t_1}\,var}{x : t_1 \vdash \lambda y.x : t_2 \to t_1}\,abs}{\vdash \lambda xy.x : t_1 \to (t_2 \to t_1)}\,abs$$

Dakle, najopštiji tip ovog izraza je $t_1 \to (t_2 \to t_1)$, pa se svaki tip koji se može pridružiti ovom izrazu može dobiti kao instanca ovog tipa. Na primer, jedan takav tip može biti tip $(\sigma \to \sigma) \to (\sigma \to (\sigma \to \sigma))$ koji se dobija primenom zamene $s = [t_1 \mapsto \sigma \to \sigma, t_2 \mapsto \sigma]$ na najopštiji tip. Odgovarajuće izvođenje se dobija primenom zamene s na sve čvorove stabla najopštijeg izvođenja:

$$\dfrac{\dfrac{\dfrac{}{x:\sigma\to\sigma, y:\sigma \vdash x:\sigma\to\sigma}\,var}{x:\sigma\to\sigma \vdash \lambda y.x:\sigma\to(\sigma\to\sigma)}\,abs}{\vdash \lambda xy.x:(\sigma\to\sigma)\to(\sigma\to(\sigma\to\sigma))}\,abs$$

Postavlja se pitanje na koji način se za proizvoljan netipizirani lambda izraz može konstruisati odgovarajuće najopštije izvođenje, ako isto postoji. U slučaju eksplicitne tipizacije, konstrukciju izvođenja i određivanje tipa je bilo jednostavno obaviti: ako imamo, na primer, izraz $\lambda f:\sigma\to\tau.\lambda x:\sigma.f\ x$, tada se određivanje njegovog tipa svodilo na rekurzivno određivanje tipa podizraza $\lambda x:\sigma.f\ x$ u kontekstu $f:\sigma\to\tau$. Dakle, s obzirom da je tip promenljive f eksplicitno zadat, kontekst u kome određujemo tip podizraza nam je poznat. Sa druge strane, da bismo u implicitnoj tipizaciji odredili najopštiji tip izraza $\lambda f.\lambda x.f\ x$, možemo na sličan način pokušati da rekurzivno odredimo tip podizraza $\lambda x.f\ x$, ali se postavlja pitanje u kom kontekstu? S obzirom da nam tip promenljive f nije eksplicitno zadat, moramo pretpostaviti da je njen tip neki najopštiji mogući tip. Drugim rečima, uvešćemo novu tipsku promenljivu t_1 i pretpostavićemo da je tip promenljive f upravo t_1. Na ovaj način, omogućavamo da se tokom postupka određivanja tipa ova tipska promenljiva instancira na odgovarajući način, u zavisnosti od upotrebe ove promenljive u izrazu. Sada možemo pokušati da rekurzivno odredimo tip izraza $\lambda x.f\ x$ u kontekstu $f:t_1$. S obzirom da je ponovo u pitanju apstrakcija, ovaj tip određujemo na sličan način – tako što uvodimo novu tipsku promenljivu t_2 i pretpostavljamo da je x tipa t_2, pa određujemo rekurzivno tip podizraza $f\ x$ u kontekstu $f:t_1, x:t_2$. Ovog puta imamo izraz $f\ x$ koji je dobijen operatorom aplikacije. Da bi izraz $f\ x$ mogao da se tipizira, neophodno je da tip promenljive f bude oblika $t_2\to\sigma$ za neki tip σ, da bi f mogla da se primeni na x. Otuda uvodimo novu tipsku promenljivu t_3 i pretpostavljamo da je f tipa $t_2\to t_3$. Međutim, mi već imamo pretpostavku da je f tipa t_1. Zbog toga pokušavamo da nekako *izjednačimo* tip t_1 sa tipom $t_2\to t_3$. Ovo *izjednačavanje* se vrši *unifikacijom* – primenom neke pogodno odabrane zamene s na oba tipa tako da postanu sintaksno jednaki.

Definicija 9.8. Neka su τ i ρ dva prosta parametarska tipa. Za zamenu s kažemo da je *unifikator* tipova τ i ρ ako važi $\tau s=\rho s$. Za tipove τ i ρ kažemo da su *unifikabilni* ako imaju bar jedan unifikator. Za instancu $\tau s\ (=\rho s)$ kažemo da je *zajednička instanca* tipova τ i ρ dobijena unifikatorom s.

Dakle, provera tipova (*type-checking*) se u slučaju implicitne tipizacije ne svodi na utvrđivanje *sintaksne jednakosti* tipa parametra funkcije i tipa argumenta, već na ispitivanje njihove *unifikabilnosti*. Ukoliko se ispostavi da jesu unifikabilni, tada se može odrediti njihov unifikator, kao i odgovarajuća zajednička instanca koja će nam odrediti neophodnu strukturu odgovarajućeg tipa. Dakle, postupkom unifikacije mi *rekonstruišemo* tipove promenljivih na osnovu konteksta u kome se primenjuju, tako što instanciramo tipske promenljive koje smo inicijalno uveli kao apstrakcije tipova u trenutku kada o njima nismo ništa znali.

Da bismo bolje razumeli pojam unifikacije tipova, ilustrujmo je najpre kroz nekoliko primera.

Primer 9.9. Razmotrimo tipove $t_1\to\sigma_1\to\sigma_2$ i $\sigma_2\to t_2$, gde su t_1 i t_2 tipski parametri, a σ_1 i σ_2 konkretni (neparametarski) prosti tipovi. Jedan mogući unifikator ova dva tipa je zamena $s=[t_1\mapsto\sigma_2, t_2\mapsto\sigma_1\to\sigma_2]$. Zaista, primenom zamene s na oba tipa dobijamo isti tip $\sigma_2\to\sigma_1\to\sigma_2$.

Primer 9.10. Razmotrimo tipove $\sigma \to (t_1 \to \sigma)$ i $t_2 \to t_1$, gde je σ neparametarski prosti tip, a t_1 i t_2 su tipske promenljive. Da bismo unifikovali ova dva tipa, promenljiva t_2 bi se morala zameniti tipom σ, kako bi se unifikovale leve strane operatora $\to$. Sa druge strane, da bismo unifikovali desne strane, potrebno je zameniti t_1 nekim tipom tako da tipovi $t_1 \to \sigma$ i t_1 postanu sintaksno identični. Lako se vidi da ovo nije moguće, jer kojim god tipom σ' da zamenimo t_1 dobićemo tipove $\sigma' \to \sigma$ i σ' koji nisu sintaksno identični. Dakle, tipovi $\sigma \to (t_1 \to \sigma)$ i $t_2 \to t_1$ nisu unifikabilni.

Primer 9.11. Neka su dati tipovi $t_1 \to (t_2 \to \sigma)$ i $t_3 \to t_4$. Jedan mogući unifikator ova dva tipa je zamena $s = [t_1 \mapsto t_3, t_4 \mapsto t_2 \to \sigma]$. Primenom ove zamene na oba tipa dobijamo tip $t_3 \to (t_2 \to \sigma)$.

Ipak, ovo nije jedini unifikator ova dva izraza. Ako primenimo zamenu $s' = [t_1 \mapsto \sigma, t_3 \to \sigma, t_4 \mapsto t_2 \to \sigma]$ na oba tipa dobijamo tip $\sigma \to (t_2 \to \sigma)$. Dakle, i zamena s' je jedan unifikator ova dva tipa.

Iz poslednjeg primera vidimo da unifikator dva tipa ne mora biti jedinstven. Da bismo bolje razumeli strukturu i međusobni odnos različitih unifikatora, uvedimo najpre pojam *kompozicije* zamena.

Definicija 9.9. *Kompozicija* zamena $s_1 = [t_1 \mapsto \sigma_1, \ldots, t_n \mapsto \sigma_n]$ i $s_2 = [u_1 \mapsto \rho_1, \ldots, u_m \mapsto \rho_m]$ (u oznaci $s_1 s_2$) je zamena $[t_1 \mapsto \sigma_1 s_2, \ldots t_n \mapsto \sigma_n s_2, u_1 \mapsto \rho_1, \ldots, u_m \mapsto \rho_m]$, pri čemu su izbačene eventualne zamene oblika $t_i \mapsto t_i$, kao i zamane $u_j \mapsto \rho_j$ gde je $u_j = t_i$ za neko i.

Može se pokazati da je kompozicija zamena asocijativna (tj. važi $(s_1 s_2)s_3 = s_1(s_2 s_3)$), kao i da za svaki tip τ važi $\tau(s_1 s_2) = (\tau s_1)s_2$. Dakle, efekat kompozicije zamena na neki tip odgovara uzastopnoj primeni pojedinačnih zamena koje se komponuju.

Za zamenu s_1 kažemo da je *opštija* od zamene s_2 ako postoji zamena s' takva da je $s_2 = s_1 s'$. Ovo ćemo zapisivati sa $s_1 \preceq s_2$.

Primer 9.12. U primeru 9.11, za unifikatore $s = [t_1 \mapsto t_3, t_4 \mapsto t_2 \to \sigma]$ i $s' = [t_1 \mapsto \sigma, t_3 \to \sigma, t_4 \mapsto t_2 \to \sigma]$ važi da je $s' = s[t_3 \mapsto \sigma]$, pa je $s \preceq s'$. Dakle, unifikator s je opštiji unifikator tipova $t_1 \to (t_2 \to \sigma)$ i $t_3 \to t_4$ od unifikatora s'. Pritom, imali smo da je $(t_1 \to (t_2 \to \sigma))s' = (t_1 \to (t_2 \to \sigma))(s[t_3 \mapsto \sigma]) = ((t_1 \to (t_2 \to \sigma))s)[t_3 \mapsto \sigma] = (t_3 \to (t_2 \to \sigma))[t_3 \mapsto \sigma] = \sigma \to (t_2 \to \sigma)$.

Koristeći pojam kompozicije zamena možemo definisati pojam *najopštijeg unifikatora* tipova. Ovaj unifikator će nam biti neophodan za određivanje najopštijeg izvođenja, kao i najopštijeg para datog izraza.

Definicija 9.10. Za zamenu s kažemo da je *najopštiji unifikator* tipova τ i ρ ako važi $\tau s = \rho s$ i za svaki drugi unifikator s' ova dva tipa važi da postoji zamena s'' takva da je $s' = ss''$.

Dakle, unifikator s je najopštiji unifikator za τ i ρ, ako se svaki drugi unifikator ta dva tipa dobija kompozicijom s i neke druge zamene. Pritom, kako je $\tau s' = \tau(ss'') = (\tau s)s''$, sledi da je zajednička instanca tipova τ i ρ koja odgovara unifikatoru s' instanca zajedničke instance τs koja odgovara najopštijem unifikatoru s. Dakle, važi $\tau s \preceq \tau s'$. Za instancu τs kažemo da je *najopštija zajednička instanca* tipova τ i ρ.

Može se pokazati sledeća teorema koju navodimo bez dokaza.

Teorema 9.4. *Svaka dva unifikabilna tipa τ i ρ imaju najopštiji unifikator i on je jedinstven do na preimenovanje promenljivih.*

Najopštiji unifikator dva tipa τ i ρ možemo odrediti primenom postupka koji opisujemo u nastavku. Najpre ćemo inicijalizovati *listu parova za unifikaciju* – ova lista je oblika $[(\tau_1, \rho_1), \ldots, (\tau_k, \rho_k)]$, pri čemu su (τ_i, ρ_i) parovi tipova koje je potrebno *istovremeno* unifikovati, tj. pronaći zamenu s za koju važi $\tau_i s = \rho_i s$ za svako i. Inicijalno, ova lista je jednočlana i sastoji se samo iz para $[(\tau, \rho)]$, a zatim je tokom postupka modifikujemo primenom sledećih pravila (u svakom koraku primenjujemo prvo od navedenih pravila koje je moguće primeniti):

- ako lista sadrži par oblika (τ, τ), tada taj par brišemo iz liste
- ako lista sadrži par (σ_1, σ_2), gde su σ_1 i σ_2 različiti atomički tipovi, tada lista nije unifikabilna, pa prekidamo postupak i prijavljujemo *neuspeh*
- ako lista sadrži par $(\sigma, \tau \to \rho)$, gde je σ neki atomički tip, tada lista takođe nije unifikabilna, pa ponovo prekidamo postupak i prijavljujemo *neuspeh*
- ako lista sadrži par oblika (t, σ) (ili (σ, t)) gde je t tipska promenljiva, a σ je neki prost parametarski tip, tada:
  - ako tip σ ne sadrži promenljivu t, a promenljiva t se pojavljuje u nekom od ostalih parova u listi, tada na sve ostale parove u listi treba primeniti zamenu $[t \mapsto \sigma]$
  - ako tip σ sadrži promenljivu t, tada tekuća lista parova nije unifikabilna, pa prekidamo postupak i prijavljujemo *neuspeh*
- ako lista sadrži par oblika $(\tau_1 \to \tau_2, \rho_1 \to \rho_2)$, tada se takav par *zamenjuje* parovima (τ_1, ρ_1) i (τ_2, ρ_2)

Ako nije više moguće primeniti ni jedno od prethodnih pravila, tada se postupak zaustavlja i prijavljujemo *uspeh* – polazni tipovi su unifikabilni. Najopštiji unifikator tipova τ i ρ se može očitati iz preostalih parova u listi – oni su svi oblika (t, σ) (ili (σ, t)), pa se ovi parovi tumače kao zamene $t \mapsto \sigma$.

Primer 9.13. U primeru 9.11, bilo je potrebno unifikovati tipove $t_1 \to (t_2 \to \sigma)$ i $t_3 \to t_4$. Primenom prethodnog postupka, imamo:

- inicijalno, imamo listu $[(t_1 \to (t_2 \to \sigma), t_3 \to t_4)]$
- ovaj par dekomponujemo i dobijamo listu $[(t_1, t_3), ((t_2 \to \sigma), t_4)]$.

Dalje nije moguće primeniti ni jedan od koraka postupka, pa je najopštiji unifikator dobijen iz finalne liste: $s = [t_1 \mapsto t_3, t_4 \mapsto (t_2 \to \sigma)]$.

Primer 9.14. Razmotrimo unifikaciju tipova iz primera 9.10: $\sigma \to (t_1 \to \sigma)$ i $t_2 \to t_1$, gde je σ bio neparametarski prost tip, a t_1 i t_2 tipske promenljive. Primenom gornjeg postupka krećemo od liste $[(\sigma \to (t_1 \to \sigma), t_2 \to t_1)]$ i primenjujemo dekompoziciju, čime dobijamo listu $[(\sigma, t_2), (t_1 \to \sigma, t_1)]$. Kako u paru $(t_1 \to \sigma, t_1)$ imamo da tip $t_1 \to \sigma$ sadrži pojavljivanje promenljive t_1, sledi da unifikacija nije moguća, pa postupak prijavljuje *neuspeh*.

Primer 9.15. Neka su dati tipovi $(t_1 \to t_2) \to (t_2 \to t_3)$ i $t_3 \to (\sigma \to t_4)$, gde je σ neparametarski prost tip. Da bismo odredili najopštiji unifikator ova dva tipa, polazimo od liste $[((t_1 \to t_2) \to (t_2 \to t_3), t_3 \to (\sigma \to t_4))]$. Dekompozicijom ovih tipova dobijamo listu $[(t_1 \to t_2, t_3), (t_2 \to t_3, \sigma \to t_4)]$. Kako u prvom paru tip $t_1 \to t_2$ ne sadrži tipsku promenljivu t_3, vršimo zamenu $[t_3 \mapsto t_1 \to t_2]$ u drugom paru u listi, te dobijamo listu $[(t_1 \to t_2, t_3), (t_2 \to (t_1 \to t_2), \sigma \to t_4)]$. Dalje se dekompozicijom drugog para dobija lista $[(t_1 \to t_2, t_3), (t_2, \sigma), (t_1 \to t_2, t_4)]$. Na osnovu drugog para vršimo zamenu $[t_2 \mapsto \sigma]$ u ostalim parovima, te dobijamo $[(t_1 \to \sigma, t_3), (t_2, \sigma), (t_1 \to \sigma, t_4)]$. Dalje nije moguće primeniti ni jedno pravilo, pa prijavljujemo *uspeh*: polazni tipovi su unifikabilni, a njihov najopštiji unifikator je $s = [t_3 \mapsto t_1 \to \sigma, t_2 \mapsto \sigma, t_4 \mapsto t_1 \to \sigma]$. Najopštija zajednička instanca ova dva tipa se dobija primenom zamene s na bilo koji od ta dva tipa, na primer, $(t_3 \to (\sigma \to t_4))s = (t_1 \to \sigma) \to (\sigma \to (t_1 \to \sigma))$.

Vratimo se sada na određivanje najopštijeg izvođenja, kao i najopštijeg para datog izraza. Postupak koji ćemo primenjivati radi određivanja najopštijeg para (engl. *type inference*) je veoma sličan postupku koji smo primenjivali za određivanje tipa u slučaju eksplicitne tipizacije. Jedina razlika je u tome što za promenljive inicijalno pretpostavljamo da imaju tipove zadate nekim tipskim promenljivama, a zatim prilikom aplikacije tipove izjednačavamo unifikacijom. Postupak preciznije opisujemo u nastavku. U pitanju je rekurzivna procedura kod koje svakom rekurzivnom pozivu predajemo neki tekući parametarski kontekst Γ i neki netipizirani lambda izraz E. Rekurzivni poziv vraća najopštiji par (Γ', σ) izraza E *saglasan* sa pretpostavkama iz Γ – ovo znači da postoji zamena s takva da je $\Gamma s \subseteq \Gamma'$. Inicijalni poziv na ulazu ima prazan kontekst (tj. nemamo nikakvih polaznih pretpostavki), kao i dati netipizirani lambda izraz čiji je najopštiji par potrebno odrediti (tj. čije je najopštije izvođenje potrebno konstruisati). U svakom rekurzivnom pozivu koji je pozvan za neki izraz E i neki kontekst Γ radimo sledeće:

- ako je E promenljiva x, tada je potrebno proveriti da li u tekućem kontekstu Γ postoji par $x : \sigma$. Ako postoji, tada rekurzivni poziv vraća najopštiji par (Γ, σ). U suprotnom, potrebno je u kontekst dodati par $x : t$, gde je t novouvedena tipska promenljiva. Sada rekurzivni poziv vraća par $(\Gamma \cup \{x : t\}, t)$

- ako je E oblika $\lambda x.E'$, tada se rekurzivno poziva postupak za izraz E' i kontekst $\Gamma \cup \{x : t\}$, gde je t nova tipska promenljiva. Ako rekurzivni poziv utvrdi da izraz E' u kontekstu $\Gamma \cup \{x : t\}$ nije tipizibilan, tada to nije ni izraz E u kontekstu Γ. U suprotnom, rekurzivni poziv vraća par oblika $(\tau, \Gamma' \cup \{x : \sigma\})$, gde je σ tip dobijen instanciranjem promenljive t tokom rekurzivnog poziva. Sada tekući poziv vraća par $(\Gamma', \sigma \to \tau)$ kao najopštiji par izraza $\lambda x.E$ saglasan sa kontekstom Γ

- ako je E oblika $E_1\ E_2$, tada najpre rekurzivnim pozivom određujemo najopšiji par izraza E_1 saglasan sa datim kontekstom Γ. Ako se ispostavi da takva tipizacija nije moguća, tada ni izraz E nije tipizibilan u kontekstu Γ. U suprotnom, rekurzivni poziv vraća najopštiji par (Γ_1, τ_1) izraza E_1 koji je saglasan sa Γ. Sada pozivamo novi rekurzivni poziv za izraz E_2 i kontekst Γ_1. Ako izraz E_2 nije tipizibilan u kontekstu Γ_1, tada ni izraz E nije tipizibilan u kontekstu Γ. U suprotnom, ovaj rekurzivni poziv

vraća najopštiji par (Γ_2, τ_2) izraza E_2 saglasan sa kontekstom Γ_1. Sada sprovodimo proveru tipova: da bi izraz $E = E_1\ E_2$ mogao biti tipiziran, neophodno je da tip τ_1 bude *unifikabilan* sa tipom $\tau_2 \to t$, gde je t nova tipska promenljiva. Ukoliko unifikacija uspe, dobijamo najopštiji unifikator s. U tom slučaju će tekući poziv vratiti par $(\Gamma_2 s, ts)$ kao najopštiji par izraza $E = E_1\ E_2$ saglasan sa kontekstom Γ. U suprotnom, izraz E nije tipizibilan u kontekstu Γ.

Na kraju postupka, par (Γ, τ) koji vrati inicijalni poziv će biti upravo najopštiji par pridružen polaznom izrazu E, s obzirom da je polazni kontekst bio prazan. Primetimo da se i ovde prilikom utvrđivanja tipa aplikacije $E_1\ E_2$ vrši provera tipa, ali se ona sada svodi na unifikabilnost, a ne na sintaksnu jednakost. Pritom, vrlo je važno da upotrebimo najopštiji unifikator, kako bismo bili sigurni da ćemo na kraju dobiti najopštije izvođenje.

Primer 9.16. Vratimo se na primer izraza $\lambda f.\lambda x.f\ x$ iz prethodnog razmatranja. Inicijalno krećemo od praznog konteksta i pokušavamo da odredimo najopštiji par izraza $\lambda f.\lambda x.f\ x$ saglasan sa praznim kontekstom (što je upravo najopštiji par datog izraza). S obzirom na dve apstrakcije, problem se svodi na rekurzivno određivanje najopštijeg tipa izraza $f\ x$ u kontekstu $\Gamma = \{f : t_1, x : t_2\}$, kao što smo ranije razmatrali. Sada je potrebno unifikovati tip t_1 sa tipom $t_2 \to t_3$, gde je t_3 nova tipska promenljiva. Najopštiji unifikator ova dva tipa je $s = [t_1 \mapsto t_2 \to t_3]$, odakle će izrazu $f\ x$ biti pridružen tip $t_3 s = t_3$. Ista zamena s se primenjuje i na kontekst, pa dobijamo kontekst $\Gamma s = \{f : t_2 \to t_3, x : t_2\}$. Dakle, najopštiji par izraza $f\ x$ saglasan sa kontekstom $\Gamma = \{f : t_1, x : t_2\}$ je $(\{f : t_2 \to t_3, x : t_2\}, t_3)$. Sada će izrazu $\lambda x.f\ x$ biti pridružen najopštiji par $(\{f : t_2 \to t_3\}, t_2 \to t_3)$ (saglasan sa kontekstom $\{f : t_1\}$), dok će izrazu $\lambda f.\lambda x.f\ x$ biti pridružen najopštiji par $(\{\}, (t_2 \to t_3) \to (t_2 \to t_3))$. Dakle, najopštiji tip izraza $\lambda f.\lambda x.f\ x$ je $(t_2 \to t_3) \to (t_2 \to t_3)$, dok je najopštiji kontekst prazan. Imamo sledeće najopštije izvođenje:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f : t_2 \to t_3, x : t_2 \vdash f : t_2 \to t_3}\,var \quad \dfrac{}{f : t_2 \to t_3, x : t_2 \vdash x : t_2}\,var}{f : t_2 \to t_3, x : t_2 \vdash f\ x : t_3}\,app}{f : t_2 \to t_3 \vdash \lambda x.f\ x : t_2 \to t_3}\,abs}{\vdash \lambda f.\lambda x.f\ x : (t_2 \to t_3) \to (t_2 \to t_3)}\,abs$$

Važi sledeća teorema koju navodimo bez dokaza.

Teorema 9.5. *U prosto tipiziranom lambda računu, za svaki tipizibilni izraz E postoji najopštiji par (Γ, τ) i on je jedinstven do na preimenovanje tipskih promenljivih.*

Može se pokazati da je se za svaki izraz E primenom prethodnog postupka zaista dobija najopštiji par, ukoliko isti postoji. Ovo za posledicu ima sledeću teoremu.

Teorema 9.6. *Problem tipizibilnosti lambda izraza u teoriji prostih tipova je odlučiv.*

Primer 9.17. Pokažimo da izraz $\lambda x.x\ x$ nije tipizibilan u teoriji prostih tipova. Primenimo prethodni postupak na dati izraz i prazan kontekst. U tu svrhu, potrebno je rekurzivno odrediti najopštiji par izraza $x\ x$ u kontekstu $\{x : t\}$,

gde je t nova tipska promenljiva. Sada je potrebno unifikovati tipove t i $t \to t'$ za neku novu tipsku promenljivu t'. Ovo nije moguće, pa unifikacija ne uspeva. Otuda ovaj izraz nije tipizibilan.

Primer 9.18. Odredimo najopštiji tip izraza $\lambda f.\lambda x.f\ (f\ x)$. Problem se svodi na određivanje najopštijeg para izraza $f\ (f\ x)$ koji je saglasan sa kontekstom $f : t_1, x : t_2$, gde su t_1 i t_2 novouvedene tipske promenljive. Dalje rekurzivno određujemo najopštiji par izraza $f\ x$ koji je saglasan sa kontekstom $f : t_1, x : t_2$. Ovo radimo tako što unifikujemo tip t_1 sa tipom $t_2 \to t_3$, gde je t_3 novouvedena tipska promenljiva. Najopštiji unifikator ova dva tipa je $[t_1 \mapsto t_2 \to t_3]$. Otuda za izraz $f\ x$ dobijamo najopštiji par $(\{f : t_2 \to t_3, x : t_2\}, t_3)$. U odnosu na dobijeni kontekst $\{f : t_2 \to t_3, x : t_2\}$ sada je potrebno odrediti najopštiji par izraza $f\ (f\ x)$. U tu svrhu unifikujemo tip $t_2 \to t_3$ promenljive f sa tipom $t_3 \to t_4$, gde je t_4 novouvedena tipska promenljiva. Najopštiji unifikator ova dva tipa je $[t_2 \mapsto t_4, t_3 \mapsto t_4]$, pa dobijamo najopštiji par $(\{f : t_4 \to t_4, x : t_4\}, t_4)$ izraza $f\ (f\ x)$. Sada će izrazu $\lambda x.f\ (f\ x)$ biti pridružen najopštiji par $(\{f : t_4 \to t_4\}, t_4 \to t_4)$, a izrazu $\lambda f.\lambda x.f\ (f\ x)$ najopštiji par $(\{\}, (t_4 \to t_4) \to (t_4 \to t_4))$. Dakle, najopštiji tip izraza $\lambda f.\lambda x.f\ (f\ x)$ je $(t_4 \to t_4) \to (t_4 \to t_4)$, a odgovarajuće najopštije izvođenje je:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash f : t_4 \to t_4}\,var \quad \dfrac{\dfrac{}{\Gamma \vdash f : t_4 \to t_4}\,var \quad \dfrac{}{\Gamma \vdash x : t_4}\,var}{f : \Gamma \vdash f\ x : t_4}\,app}{\Gamma \vdash f\ (f\ x) : t_4}\,app}{f : t_4 \to t_4 \vdash \lambda x.f\ (f\ x) : t_4 \to t_4}\,abs}{\vdash \lambda f.\lambda x.f\ (f\ x) : (t_4 \to t_4) \to (t_4 \to t_4)}\,abs$$

gde je $\Gamma = \{f : t_4 \to t_4, x : t_4\}$.

9.2.1 Redukcija i implicitna tipizacija

Kao i kod eksplicitno tipiziranih lambda izraza, i u slučaju implicitne tipizacije imamo da se β-redukcija dobro ponaša, u smislu očuvanja tipova i tipizibilnosti.

Teorema 9.7. *Neka je E implicitno tipizirani lambda izraz za koji važi $\Gamma \vdash E : \sigma$. Ako je $E \Rightarrow_\beta E'$, tada važi $\Gamma \vdash E' : \sigma$.*

Slično važi i za η-redukciju.

Teorema 9.8. *Neka je E implicitno tipizirani lambda izraz za koji važi $\Gamma \vdash E : \sigma$. Ako je $E \Rightarrow_\eta E'$, tada je i $\Gamma \vdash E' : \sigma$.*

Dokazi prethodnih teorema se izvode na sličan način kao i u slučaju eksplicitne tipizacije.

Posledica 9.2. *Za proizvoljni implicitno tipizirani izraz E ako je $E \Rightarrow^*_{\beta\eta} E'$, tada iz $\Gamma \vdash E : \sigma$ sledi $\Gamma \vdash E' : \sigma$.*

Primetimo da obratno ne mora da važi, tj. može se desiti da za izraze E i E' za koje je $E \Rightarrow^*_{\beta\eta} E'$ važi $\Gamma \vdash E' : \sigma$, iako ne važi $\Gamma \vdash E : \sigma$. O ovome govori sledeći primer.

Primer 9.19. Razmotrimo izraz $E = \lambda x.\lambda y.(\lambda u.\lambda v.u)\ y\ (x\ y)$. Primenom postupka određivanja najopštijeg tipa, dobijamo najopštiji tip $(t_1 \to t_2) \to (t_1 \to t_1)$ (proveriti za vežbu!). Sa druge strane, β-redukcijom ovog izraza dobijamo izraz $E' = \lambda x.\lambda y.y$ čiji je najopštiji tip $t_3 \to t_4 \to t_4$. Dakle, najopštiji tip redukovanog izraza je opštiji od najopštijeg tipa polaznog izraza (primenom zamene $[t_3 \mapsto t_1 \to t_2, t_4 \mapsto t_1]$ na tip $t_3 \to t_4 \to t_4$ dobijamo tip $(t_1 \to t_2) \to (t_1 \to t_1)$). Zbog toga je redukovanom izrazu moguće pridružiti širi skup tipova nego polaznom izrazu. Na primer, ako su σ i τ proizvoljni *atomički* tipovi, tada za redukovani izraz E' važi $\vdash E' : \sigma \to \tau \to \tau$, dok za polazni izraz E ne važi $\vdash E : \sigma \to \tau \to \tau$.

Dakle, redukovanom izrazu mogu, u opštem slučaju, biti pridruženi i neki tipovi koji ne mogu biti pridruženi polaznom izrazu. Specijalno, može se dogoditi da polazni izraz uopšte nije tipizibilan, a da redukovani to jeste.

Primer 9.20. Razmotrimo izraz $(\lambda xy.y)\ (\lambda u.u\ u)$. Ovaj izraz nije tipizibilan, zato što to nije argument $(\lambda u.u\ u)$. Međutim, kako funkcija $\lambda xy.y$ ne koristi svoj prvi argument u telu, redukcijom dobijamo izraz $\lambda y.y$ koji ima najopštiji tip $t \to t$.

9.3 Izračunljivost u teoriji prostih tipova

U prethodnom izlaganju smo ustanovili da, na primer, izraz $E = \lambda x.x\ x$ nije tipizibilan u teoriji prostih tipova. Ovo nije bilo slučajno – teorija prostih tipova ne dopušta mogućnost da funkcija prihvati samu sebe kao argument. To znači da ni izraz $\Omega = E\ E$ takođe nije tipizibilan. Da se podsetimo, to je upravo onaj izraz za koji smo imali beskonačni lanac redukcija:

$$\Omega \Rightarrow_\beta \Omega \Rightarrow_\beta \Omega \Rightarrow_\beta \ldots$$

Razlog za nepostojanje normalne forme ovog izraza je upravo u činjenici da u sebi sadrži primenu funkcije na samu sebe. Kako ovako nešto u teoriji prostih tipova nije moguće tipizirati, ispostavlja se da smo prostom tipizacijom osigurali *zaustavljanje* procesa redukcije. Zaista, važi sledeća teorema (koju navodimo bez dokaza).

Teorema 9.9 (Teorema o jakoj normalizaciji)**.** *Ako je izraz E tipizibilan u teoriji prostih tipova, tada on ima $\beta\eta$-normalnu formu do koje se garantovano dolazi bez obzira na odabrani redosled redukcija.*

Na prvi pogled, ovo je dobra vest – ako razmatranje ograničimo samo na tipizibilne izraze, onda nema više bojazni od beskonačnog izračunavanja, niti moramo da vodimo računa o redosledu redukcija, garantovano nas svaki redosled vodi do normalne forme. Ipak, ova činjenica krije i jednu lošu vest – formalizam prosto tipiziranih lambda izraza više nije Tjuring kompletan. Naime, Tjuringove mašine (kao i ostali formalizmi koje smo izučavali) mogu izračunavati i funkcije koje nisu totalne, tj. mogu postojati ulazi za koje se data Tjuringova mašina ne zaustavlja. Jasno je da se izračunavanje takvih funkcija ne može simulirati u prosto tipiziranom lambda računu, s obzirom da se redukcija prosto tipiziranih lambda izraza uvek zaustavlja.

Dublji razlog za ovaj fenomen se ogleda u činjenici da se u teoriji prostih tipova ne može tipizirati ni kombinator fiksne tačke Y:

$$Y = \lambda h.(\lambda g.h\ (g\ g))\ (\lambda g.h\ (g\ g))$$

jer u sebi sadrži izraz $g\ g$ koji nije tipizibilan. Ovo za posledicu ima da pomoću prosto tipiziranih lambda izraza ne možemo simulirati rekurziju, što značajno smanjuje njihovu moć izračunavanja. Ne samo što nije moguće reprezentovati izračunljive funkcije koje nisu totalne, već je i veoma mali deo primitivno rekurzivnih funkcija (koje jesu totalne) moguće izraziti u prosto tipiziranom lambda računu. Ilustrujmo ovo kroz par primera.

Primer 9.21. Razmotrimo tipizaciju lambda izraza koje smo u odeljku 8.3.1 koristili za predstavljanje logičkih vrednosti i funkcija:

- izraz $true = \lambda xy.x$ se može tipizirati najopštijim tipom $a \to b \to a$, dok se izraz $false = \lambda xy.y$ može tipizirati najopštijim tipom $c \to d \to d$. Očito, ova dva tipa nisu ista, kao što bismo očekivali za dve konstante koje bi intuitivno trebalo da budu istog tipa. Najopštija zajednička instanca ova dva tipa je $t \to t \to t$, tako da bi neki tip tog oblika mogao da reprezentuje logički tip. Na primer, mogli bismo uvesti oznaku $bool = \sigma \to \sigma \to \sigma$, za neki unapred fiksirani atomički tip σ. Može se pokazati da su za tako definisani tip *bool* izrazi $true$ i $false$ jedini zatvoreni izrazi u normalnoj formi tog tipa. Otuda se na ovaj način u potpunosti postiže osnovni cilj tipizacije – da se spreči da funkcija koja očekuje argument tipa *bool* prihvati bilo šta drugo osim $true$ ili $false$.

- uslovni izraz $cond = \lambda u.\lambda x.\lambda y.u\ x\ y$ ima najopštiji tip $(a \to b \to c) \to a \to b \to c$. S obzirom na intuitivno značenje operatora *cond*, u bi trebalo da bude logičkog tipa, dok bi x i y trebali da budu istog tipa, kao i da povratna vrednost bude tog tipa. Ovo bismo mogli postići tako što tipske promenljive a, b, c instanciramo atomičkim tipom σ, odakle dobijamo tip $(\sigma \to \sigma \to \sigma) \to \sigma \to \sigma \to \sigma$ kao jedan od tipova koji je moguće pridružiti operatoru *cond*. Ako sada kao i malopre uvedemo oznaku $bool = \sigma \to \sigma \to \sigma$, gornji tip možemo zapisati kao $bool \to \sigma \to \sigma \to \sigma$, što odgovara našoj intuiciji za tip operatora *cond*.[1] Ipak, ovakva tipizacije krije jedno suptilno ograničenje – tip σ je *ugrađen* u sam tip *bool*, što znači da se sada operator *cond* može primenjivati samo za izbor između vrednosti tipa σ, čime se gubi njegova opštost.

- operator $and = \lambda b_1.\lambda b_2.cond\ b_1\ b_2\ false$ se takođe može tipizirati, ali njegov najopštiji tip postaje prilično neprirodan i neintuitivan. Naime, s obzirom da je tip prvog argumenta operatora *cond* oblika $a \to b \to c$, a tip drugog argumenta onda mora biti a, sledi da b_1 i b_2 ne mogu biti istog tipa, kao što bismo intuitivno očekivali. Treći operand mora biti tipa b, a kako je najopštiji tip izraza $false$ oblika $d \to e \to e$, unifikacijom dobijamo da mora biti $b \mapsto (d \to e \to e)$, pa je najopštiji tip operatora *and*:

  $$(a \to (d \to e \to e) \to c) \to a \to c$$

  S obzirom da se tip prvog i drugog argumenta u gornjem najopštijem tipu ne mogu nikako unifikovati, jasno je da nije moguće koristiti isti tip za oba

[1] Da bismo fiksirali da tip operatora *cond* bude baš $bool \to \sigma \to \sigma \to \sigma$, možemo koristiti eksplicitnu tipizaciju: $cond = \lambda u : bool.\lambda x : \sigma.\lambda y : \sigma.u\ x\ y$.

argumenta b_1 i b_2, kao što bi bilo prirodno. Specijalno, nije moguće da b_1 i b_2 budu tipa $\sigma \to \sigma \to \sigma$, kako smo ranije definisali tip *bool*.

U gornjim primerima tipizacija je uspevala, ali po veoma visokoj ceni – dobijeni najopštiji tipovi nisu bili intuitivni i nije bilo moguće definisati jedinstveni *bool* tip koji bi mogao biti uniformno korišćen u svim situacijama. Jasno je da ovako nešto nije samo ,,kozmetički" problem, već da će pre ili kasnije isplivati ozbiljna ograničenja prilikom pokušaja tipizacije različitih logičkih izraza. Na primer, pokušajmo da tipiziramo izraz $\lambda b.and\ b\ b$. U pitanju je krajnje jednostavan izraz – funkcija koja računa vrednost $b \wedge b$. Ipak, ovaj izraz nije moguće tipizirati. Naime, s obzirom da su oba argumenta operatora *and* jednaki b, oni moraju biti istog tipa. Sa druge strane, gore navedeni najopštiji tip operatora *and* to isključuje. Dakle, ovo je tačka u kojoj ,,udaramo u zid" – čak i vrlo jednostavni logički izrazi ne mogu biti tipizirani u teoriji prostih tipova.

Primer 9.22. Razmotrimo sada tipizaciju Čerčovih numerala i aritmetičkih funkcija, onako kako smo ih definisali u odeljku 8.3.4:

- Numeral $\overline{0}$ je bio predstavljen izrazom $\lambda f x.x$ čiji je najopštiji tip $a \to b \to b$. Numeral $\overline{1}$ je bio predstavljen izrazom $\lambda f x.f\ x$ čiji je najopštiji tip $(c \to d) \to c \to d$. Numeral $\overline{2}$ je bio predstavljen izrazom $\lambda f x.f\ (f\ x)$ čiji je najopštiji tip $(t \to t) \to t \to t$. Lako se može videti da se isti najopštiji tip može pridružiti i svim ostalim numeralima oblika $\lambda f x.f^n\ x$. Najopštija zajednička instanca svih navedenih tipova je upravo $(t \to t) \to t \to t$, pa bismo za tip prirodnih brojeva mogli uvesti tip $nat = (\sigma \to \sigma) \to \sigma \to \sigma$ za neki konkretan atomički tip σ. Može se pokazati da će svi zatvoreni i redukovani lambda izrazi kojima se može pridružiti ovaj tip biti upravo Čerčovi numerali, čime se ponovo postiže željeni cilj – da funkcije koje očekuju tip *nat* kao svoj argument mogu prihvatiti samo Čerčove numerale.

- Odredimo najopštiji tip operatora $succ = \lambda n.\lambda f.\lambda x.f\ (n\ f\ x)$. Kako se n primenjuje na f i x, ako pretpostavimo da je f tipa a, a x tipa b, tada n mora biti tipa $a \to b \to c$. Međutim, kako se f primenjuje na izraz $n\ f\ x$ (koji je tipa c), tada f mora biti tipa oblika $c \to d$. Odatle moramo izvršiti unifikaciju zamenom $[a \mapsto (c \to d)]$, pa će n biti tipa $(c \to d) \to b \to c$. Sada će tip operatora *succ* biti:

  $$((c \to d) \to b \to c) \to (c \to d) \to b \to d$$

  Ako sve tipske promenljive u gornjem parametarskom tipu instanciramo atomičkim tipom σ, dobijamo tip:

  $$((\sigma \to \sigma) \to \sigma \to \sigma) \to (\sigma \to \sigma) \to \sigma \to \sigma$$

  Ako, kao i ranije, uzmemo da je $nat = (\sigma \to \sigma) \to \sigma \to \sigma$, gornji tip se može zapisati u obliku:

  $$nat \to nat$$

  što odgovara intuiciji da *succ* preslikava *nat* u *nat*.

- Kako nemamo mogućnost reprezentacije primitivne rekurzije, naše razmatranje je unapred ograničeno na neke elementarne aritmetičke operacije

koje se mogu definisati bez upotrebe primitivne rekurzije. U primeru 8.22 videli smo da se funkcija *add* može predstaviti sledećim lambda izrazom:

$$add = \lambda m.\lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x)$$

Tip promenljive n mora biti oblika $a \to b \to c$, gde su a i b tipovi promenljivih f i x, respektivno. Sada će izraz $n\ f\ x$ imati tip c, pa m mora biti tipa $a \to c \to d$. Odavde izraz $m\ f\ (n\ f\ x)$ ima tip d, pa je najopštiji tip izraza *add*:

$$(a \to c \to d) \to (a \to b \to c) \to a \to b \to d$$

Primenom zamene $[a \mapsto (\sigma \to \sigma), b \mapsto \sigma, c \mapsto \sigma, d \mapsto \sigma]$ dobijamo tip:

$$((\sigma \to \sigma) \to \sigma \to \sigma) \to ((\sigma \to \sigma) \to \sigma \to \sigma) \to ((\sigma \to \sigma) \to \sigma \to \sigma)$$

odnosno:

$$nat \to nat \to nat$$

pri čemu smo uzeli da je $nat = (\sigma \to \sigma) \to (\sigma \to \sigma)$, kao i ranije.

- Slično možemo uraditi i sa definicijom funkcije *mul*, koja se, bez korišćenja rekurzije, može zadati ovako:

$$mul = \lambda m.\lambda n.\lambda f.m\ (n\ f)$$

Ako je a tip promenljive f, tada je tip promenljive n oblika $a \to b$, pa će izraz $n\ f$ biti tipa b. Promenljiva m mora biti tipa $b \to c$, pa će izraz $m\ (n\ f)$ biti tipa c. Otuda izraz *mul* ima najopštiji tip:

$$(b \to c) \to (a \to b) \to a \to c$$

Primenom zamene $[a \mapsto (\sigma \to \sigma), b \mapsto (\sigma \to \sigma), c \mapsto (\sigma \to \sigma)]$ dobijamo tip:

$$((\sigma \to \sigma) \to (\sigma \to \sigma)) \to ((\sigma \to \sigma) \to (\sigma \to \sigma)) \to ((\sigma \to \sigma) \to (\sigma \to \sigma))$$

odnosno:

$$nat \to nat \to nat$$

pri čemu je ponovo $nat = (\sigma \to \sigma) \to (\sigma \to \sigma)$.

Iz ovih primera sledi da se svi aritmetički izrazi koji uključuju sabiranje i množenje mogu konzistentno tipizirati. Otuda se, na primer, izračunavanje vrednosti polinoma može obaviti u okviru prosto tipiziranog lambda računa.

Ipak, postoje i vrlo jednostavne aritmetičke funkcije koje nije moguće tipizirati. Razmotrimo, na primer, funkciju stepenovanja:

$$exp = \lambda m.\lambda n.n\ m$$

koja je za dva numerala $\overline{m}$ i $\overline{n}$ izračunavala numeral $\overline{m^n}$. Najopštiji tip izraza *exp* je:

$$a \to (a \to b) \to b$$

Sada ponovo imamo problem da tipovi prvog i drugog argumenta ne mogu biti isti tip, jer tipovi a i $a \to b$ nisu unifikabilni. Specijalno, očekivani tip $nat \to nat \to nat$ neće biti instanca ovog tipa, pa neće moće da bude pridružen funkciji *exp*. Ovo otežava mogućnost tipizacije aritmetičkih izraza koji uključuju funkciju stepenovanja. Na primer, izraz:

$$\lambda n.exp\ n\ n$$

koji izračunava n^n neće biti tipizibilan u teoriji prostih tipova.

U oba prethodna primera, suštinski problem je bio u tome što nije postojala mogućnost da se tipovi *bool* i *nat* fiksiraju tako da omoguće konzistentnu tipizaciju odgovarajućih logičkih i aritmetičkih funkcija. Kod logičkih funkcija, problem je nastajao već kod elementarnih iskaznih veznika, poput funkcije *and*. U slučaju aritmetike, imali smo mogućnost da donekle konzistentno gradimo aritmetičke funkcije (sledbenik, sabiranje, množenje), ali se čitava konstrukcija raspala kod operacije stepenovanja. Ovo znači da je domet izračunljivosti u teoriji prostih tipova mnogo manji od primitivne rekurzije i opštih izračunljivih funkcija.